



PONTIFÍCIA UNIVERSIDADE CATÓLICA DE MINAS Gerais
Instituto de Ciências Exatas e de Informática

Relatório Trabalho Prático 01*

Algoritmos e Estruturas de Dados III

Bernardo Ladeira Kartabil¹

Marcella Santos Belchior²

Thiago Henrique Gomes Feliciano³

Yasmin Torres Moreira dos Santos⁴

*Relatório da primeira parte do Trabalho Prático, da Disciplina de AEDS III, do curso de Ciência da Computação.

¹ Aluno do Curso de Graduação em Ciência da Computação, Brasil– bernardo.kartabil@sga.pucminas.br.

² Aluna do Curso de Graduação em Ciência da Computação, Brasil– marcella.belchior@sga.pucminas.br.

³ Aluno do Curso de Graduação em Ciência da Computação, Brasil– 1543790@sga.pucminas.br.

⁴ Aluna do Curso de Graduação em Ciência da Computação, Brasil– yasmin.santos.1484596@sga.pucminas.br.

Resumo

O presente trabalho detalha o desenvolvimento do "PresenteFácil", um sistema de gerenciamento de listas de presentes implementado na linguagem Java. O projeto foi concebido como parte da disciplina de Algoritmos e Estruturas de Dados III, com o objetivo principal de aplicar conceitos avançados de persistência de dados e indexação em arquivos. O sistema suporta um CRUD (Create, Read, Update, Delete) completo para as entidades de Usuários e Listas de Presentes. Para garantir a eficiência no acesso aos dados, foram implementadas e utilizadas estruturas de indexação externas, notadamente a Hash Extensível e a Árvore B+. A Hash Extensível é utilizada para criar índices diretos e indiretos, permitindo buscas rápidas por chaves primárias (ID) e secundárias (e-mail, código da lista). A Árvore B+ foi empregada para materializar o relacionamento 1:N entre usuários e suas respectivas listas, otimizando a consulta de "minhas listas". A arquitetura do sistema segue o padrão Model-View-Controller (MVC), separando a lógica de dados, a interface com o usuário (via console) e as regras de negócio.

Palavras-chave: Java. Estrutura de Dados. Hash Extensível. Árvore B+. CRUD. Persistência de Dados.

1 INTRODUÇÃO

O desenvolvimento de sistemas que manipulam grandes volumes de dados de forma eficiente é um desafio central na engenharia de software. A escolha de estruturas de dados adequadas para armazenamento e recuperação de informações em memória secundária é fundamental para o desempenho e a escalabilidade de uma aplicação.

Neste contexto, o projeto "PresenteFácil" foi desenvolvido para aplicar na prática as teorias de organização de arquivos e estruturas de dados avançadas. O sistema simula uma aplicação real onde usuários podem se cadastrar, criar múltiplas listas de presentes para diferentes ocasiões e compartilhá-las com outras pessoas por meio de um código único.

O principal objetivo deste trabalho foi construir um sistema funcional que não dependesse de um banco de dados tradicional. Em vez disso, toda a persistência dos dados foi implementada diretamente em arquivos binários, gerenciados por meio de classes customizadas que controlam a alocação de espaço, o tratamento de registros de tamanho variável e a reutilização de espaços liberados por registros excluídos. A eficiência das operações de busca, inserção e remoção é garantida pelo uso de arquivos de índice baseados em Hash Extensível e Árvore B+.

2 DESENVOLVIMENTO

O sistema foi estruturado seguindo o padrão arquitetural Model-View-Controller (MVC) para promover a organização e a manutenibilidade do código. Além disso, foi criado um pacote específico (`aeds3`) para abrigar as estruturas de dados genéricas.

2.1 Estruturas de Dados e Persistência

A persistência de dados é o núcleo técnico do trabalho. Foi implementada uma solução de armazenamento baseada em arquivos de acesso aleatório (`RandomAccessFile`).

2.1.1 *Arquivo.java*

Esta é uma classe genérica que serve como base para a persistência de qualquer registro. Dada a sua complexidade, a análise da classe será dividida em partes. A estrutura fundamental (Figura 1) inclui os atributos de controle e o construtor que inicializa o índice direto. O método `create` (Figura 2) lida com a inserção e reutilização de espaço.

O método de leitura (Figura 3) utiliza o índice primário para encontrar o endereço do registro e o lê, retornando o objeto. Já o método de exclusão (Figura 4) localiza o registro, marca o espaço com uma lápide ("*") e gerencia a lista de espaços vazios para futura reutilização.

Por fim, o método `update` (Figura 5) controla a atualização de registros, realocando-os quando necessário.

Figura 1 – Estrutura e construtor da classe `Arquivo.java`

```
1  package src.presenteFacil.aeds3;
2
3  import java.io.File;
4  import java.io.RandomAccessFile;
5  import java.lang.reflect.Constructor;
6
7  import src.presenteFacil.model.Registro;
8
9  public class Arquivo<T extends Registro> {
10
11      final int TAM_CABECALHO = 12;
12
13      private RandomAccessFile arquivo;
14      private String nomeArquivo;
15      private Constructor<T> construtor;
16      private HashExtensivel<ParIDEndereco> indiceDireto;
17
18      public Arquivo(String na, Constructor<T> c) throws Exception {
19          File d = new File(".\\data");
20          if (!d.exists()) d.mkdir();
21
22          d = new File(".\\data\\" + na);
23          if (!d.exists()) d.mkdir();
24
25          this.nomeArquivo = ".\\data\\" + na + "\\" + na + ".db";
26          this.construtor = c;
27          this.arquivo = new RandomAccessFile(this.nomeArquivo, "rw");
28
29          if (arquivo.length() < TAM_CABECALHO) {
30              arquivo.writeInt(0);
31              arquivo.writeLong(-1);
32          }
33
34          indiceDireto = new HashExtensivel<>(
35              ParIDEndereco.class.getConstructor(),
36              4,
37              ".\\data\\" + na + "\\" + na + ".d.db",
38              ".\\data\\" + na + "\\" + na + ".c.db"
39          );
40      }
```

Fonte: Elaborado pelos autores

Figura 2 – Lógica de criação de registros em `Arquivo.java`

```
1  public int create(T obj) throws Exception {
2      arquivo.seek(0);
3      int proximoID = arquivo.readInt() + 1;
4      arquivo.seek(0);
5      arquivo.writeInt(proximoID);
6      obj.setID(proximoID);
7
8      byte[] b = obj.toByteArray();
9
10     long endereco = getDeleted(b.length);
11     if (endereco == -1) {
12         arquivo.seek(arquivo.length());
13         endereco = arquivo.getFilePointer();
14         arquivo.writeByte(' ');
15         arquivo.writeShort(b.length);
16         arquivo.write(b);
17     } else {
18         arquivo.seek(endereco);
19         arquivo.writeByte(' ');
20         arquivo.skipBytes(2);
21         arquivo.write(b);
22     }
23
24     indiceDireto.create(new ParIDEndereco(proximoID, endereco));
25     return obj.getID();
26 }
```

Fonte: Elaborado pelos autores

Figura 3 – Método de leitura em Arquivo.java

```
1  public T read(int id) throws Exception {
2      ParIDEndereco pid = indiceDireto.read(id);
3      if (pid != null) {
4          arquivo.seek(pid.getEndereco());
5          T obj = construtor.newInstance();
6          byte lapide = arquivo.readByte();
7
8          if (lapide == ' ') {
9              short tam = arquivo.readShort();
10             byte[] b = new byte[tam];
11             arquivo.read(b);
12             obj.fromByteArray(b);
13             if (obj.getID() == id) return obj;
14         }
15     }
16     return null;
17 }
```

Fonte: Elaborado pelos autores

Figura 4 – Método de exclusão em Arquivo.java

```
1  public boolean delete(int id) throws Exception {
2      ParIDEndereco pie = indiceDireto.read(id);
3      if (pie != null) {
4          arquivo.seek(pie.getEndereco());
5          byte lapide = arquivo.readByte();
6
7          if (lapide == ' ') {
8              short tam = arquivo.readShort();
9              byte[] b = new byte[tam];
10             arquivo.read(b);
11             T obj = construtor.newInstance();
12             obj.fromByteArray(b);
13
14             if (obj.getID() == id) {
15                 if (indiceDireto.delete(id)) {
16                     arquivo.seek(pie.getEndereco());
17                     arquivo.write('*');
18                     addDeleted(tam, pie.getEndereco());
19                     return true;
20                 }
21             }
22         }
23     }
24     return false;
25 }
```

Fonte: Elaborado pelos autores

Figura 5 – Lógica de atualização de registros em Arquivo.java

```
1 public boolean update(T novoObj) throws Exception {
2     ParIDEndereco pie = indiceDireto.read(novoObj.getID());
3     if (pie == null) {
4         return false; // Registro não encontrado no índice
5     }
6
7     // Lê o tamanho do registro antigo para comparar
8     arquivo.seek(pie.getEndereco());
9     byte lapide = arquivo.readByte();
10    if (lapide == '*') {
11        return false; // Registro marcado como excluído
12    }
13    short tamAntigo = arquivo.readShort();
14
15    // Serializa o novo objeto para saber seu tamanho
16    byte[] b_novo = novoObj.toByteArray();
17    short tamNovo = (short) b_novo.length;
18
19    // Se o novo registro for menor ou do mesmo tamanho, sobrescreve no mesmo lugar
20    if (tamNovo <= tamAntigo) {
21        arquivo.seek(pie.getEndereco() + 1); // Volta para a posição do tamanho
22        arquivo.writeShort(tamNovo); // ESCRIBE O NOVO TAMANHO (A CORREÇÃO)
23        arquivo.write(b_novo); // Escreve os novos dados
24    }
25    // Se o novo registro for maior, apaga o antigo e escreve o novo em outro lugar
26    else {
27        // 1. Marca o registro antigo como excluído
28        arquivo.seek(pie.getEndereco());
29        arquivo.writeByte('*');
30        addDeleted(tamAntigo, pie.getEndereco());
31
32        // 2. Encontra um novo local para o registro (reutilizado ou no fim do arquivo)
33        long novoEndereco = getDeleted(tamNovo);
34        if (novoEndereco == -1) {
35            novoEndereco = arquivo.length();
36        }
37
38        // 3. Escreve o novo registro no novo local
39        arquivo.seek(novoEndereco);
40        arquivo.writeByte(' ');
41        arquivo.writeShort(tamNovo);
42        arquivo.write(b_novo);
43
44        // 4. Atualiza o índice para apontar para o novo endereço
45        indiceDireto.update(new ParIDEndereco(novoObj.getID(), novoEndereco));
46    }
47    return true;
48 }
```

Fonte: Elaborado pelos autores

3 CHECKLIST DE REQUISITOS

A seguir, são apresentadas as respostas ao checklist proposto para a avaliação do trabalho, com cada item detalhado em uma subseção para maior clareza.

3.1 Há um CRUD de usuários que funciona corretamente?

Sim. A classe `ArquivoUsuario`, que herda da classe genérica `Arquivo`, é a responsável por gerenciar a persistência e indexação dos usuários. Sua funcionalidade será detalhada a seguir.

Primeiramente, a estrutura da classe inclui um índice indireto, uma `HashExtensivel`, para mapear o e-mail do usuário ao seu ID. O construtor inicializa este índice e o método `create` é sobrescrito para garantir que, ao criar um usuário, seu e-mail e ID sejam devidamente registrados no índice, como mostra a Figura 6.

A principal vantagem desse índice é permitir a leitura de um usuário diretamente pelo seu e-mail. O método `read(String email)`, visto na Figura 7, utiliza o índice para encontrar o ID correspondente ao e-mail e, em seguida, chama o método de leitura por ID da classe pai.

Para manter a integridade do índice, os métodos `delete` e `update` também são sobrescritos. Como demonstrado na Figura 8, eles garantem que qualquer remoção ou alteração de um usuário (principalmente do seu e-mail) seja refletida no arquivo de índice.

A Figura 9 mostra a prova de execução da criação de um novo usuário através do menu do sistema no terminal.

Figura 6 – Evidência em Código: Estrutura, construtor e criação em `ArquivoUsuario.java`

```

1  package src.presenteFacil.model;
2
3  import src.presenteFacil.aeds3.Arquivo;
4  import src.presenteFacil.aeds3.HashExtensivel;
5  import src.presenteFacil.aeds3.ParEmailID;
6
7  /**
8   * Classe ArquivoUsuario
9   * -----
10  * Especialização da classe Arquivo para manipulação de objetos do tipo Usuario.
11  * * Além do arquivo principal (que armazena os usuários em si),
12  * a classe mantém um índice indireto baseado no email do usuário,
13  * utilizando Hash Extensível para permitir busca, atualização e exclusão por email.
14  */
15
16  public class ArquivoUsuario extends Arquivo<Usuario> {
17
18      // Índice indireto para localizar usuários a partir de seus emails
19      private HashExtensivel<ParEmailID> indiceIndiretoEmail;
20
21      /**
22       * Construtor da classe ArquivoUsuario.
23       * Cria o arquivo principal de usuários e inicializa o índice por email.
24       */
25      public ArquivoUsuario() throws Exception {
26          super("usuarios", Usuario.class.getConstructor());
27
28          indiceIndiretoEmail = new HashExtensivel<>(
29              ParEmailID.class.getConstructor(),
30              4, // tamanho inicial
31              ".\\data\\usuarios\\indiceEmail.d.db", // diretório do índice
32              ".\\data\\usuarios\\indiceEmail.c.db" // cestos do índice
33          );
34      }
35
36      /**
37       * Cria um novo usuário no arquivo principal e insere seu email no índice.
38       *
39       * @param u Usuário a ser criado
40       * @return ID do usuário criado
41       */
42      @Override
43      public int create(Usuario u) throws Exception {
44          int id = super.create(u); // cria no arquivo principal
45          indiceIndiretoEmail.create(new ParEmailID(u.getEmail(), id)); // cria no índice
46          return id;
47      }

```

Fonte: Elaborado pelos autores

Figura 7 – Evidência em Código: Leitura de usuário por e-mail

```
1  /**
2   * Lê um usuário a partir de seu email.
3   *
4   * @param email Email do usuário
5   * @return Objeto Usuario correspondente ou null se não existir
6   */
7  public Usuario read(String email) throws Exception {
8      ParEmailID pei = indiceIndiretoEmail.read(ParEmailID.hash(email));
9      if (pei == null)
10         return null;
11
12     return read(pei.getId()); // usa o método read(int id) da classe pai
13 }
14
```

Fonte: Elaborado pelos autores

Figura 8 – Evidência em Código: Manutenção do índice nos métodos delete e update

```
1  /**
2   * Remove um usuário a partir do ID.
3   * Remove tanto do arquivo principal quanto do índice de emails.
4   *
5   * @param id ID do usuário
6   * @return true se conseguiu remover, false caso contrário
7   */
8  @Override
9  public boolean delete(int id) throws Exception {
10     Usuario u = super.read(id);
11     if (u != null) {
12         if (super.delete(id)) {
13             // remove também do índice de emails
14             return indiceIndiretoEmail.delete(ParEmailID.hash(u.getEmail()));
15         }
16     }
17     return false;
18 }
19
20 /**
21 * Atualiza os dados de um usuário existente.
22 * Se o email foi alterado, atualiza o índice correspondente.
23 *
24 * @param novoUsuario Usuário com os novos dados
25 * @return true se conseguiu atualizar, false caso contrário
26 */
27 @Override
28 public boolean update(Usuario novoUsuario) throws Exception {
29     Usuario usuarioVelho = super.read(novoUsuario.getId());
30
31     if (usuarioVelho == null) {
32         return false; // não existe usuário com esse ID
33     }
34
35     if (super.update(novoUsuario)) {
36         // Caso o email tenha mudado, atualiza o índice
37         if (!novoUsuario.getEmail().equals(usuarioVelho.getEmail())) {
38             indiceIndiretoEmail.delete(ParEmailID.hash(usuarioVelho.getEmail()));
39             indiceIndiretoEmail.create(new ParEmailID(novoUsuario.getEmail(), novoUsuario.getId()));
40         }
41         return true;
42     }
43     return false;
44 }
45
46 }
```

Fonte: Elaborado pelos autores

Figura 9 – Prova de Execução: Tela de criação de um novo usuário

```
----- PresenteFácil 1.0 -----  
-----  
(1) Login  
(2) Novo usuário  
  
(5) Sair  
Opção: 2  
  
----- Novo Usuário -----  
  
Nome completo: Bernardo Ladeira Borges Kartabil  
E-mail: bernardo@gmail.com  
Senha: 1234  
  
Pergunta secreta: O que mora em um abacaxi no fundo do mar?  
Resposta secreta: Bob Esponja  
  
----- Login -----  
E-mail: bernardo@gmail.com  
Senha: 1234  
  
-- Login efetuado com sucesso! Bem-vindo(a), Bernardo Ladeira Borges Kartabil. --  
  
----- Menu Principal -----  
-----  
> Início  
  
(1) Meus dados.....  
(2) Minhas listas.....  
(3) Produtos.....  
(4) Buscar lista.....  
(5) Desativar minha conta.....  
(6) Excluir minha conta.....  
  
(5) Sair.....  
Opção: 
```

Fonte: Elaborado pelos autores

3.2 Há um CRUD de listas que funciona corretamente?

Sim. De forma análoga, a classe `ArquivoLista` (Figura 10) estende `Arquivo` e implementa todas as operações de **CRUD** para a entidade `Lista`. A Figura 11 demonstra a criação de uma nova lista para um usuário logado.

Figura 10 – Evidência em Código: Índices de ArquivoLista.java

```
1 package src.presenteFacil.model;
2
3 import java.util.ArrayList;
4 import src.presenteFacil.aeds3.*;
5
6 public class ArquivoLista extends Arquivo<Lista> {
7     HashExtensivel<ParIDEndereco> indiceDiretoID;
8     HashExtensivel<ParCodigoID> indiceDiretoCodigo;
9     ArvoreBMais<ParIntInt> usuariolista;
10
11     public ArquivoLista() throws Exception {
12         super("listas", Lista.class.getConstructor());
13
14         indiceDiretoID = new HashExtensivel<>(
15             ParIDEndereco.class.getConstructor(),
16             4,
17             "./data/listas/lista.id.d.db",
18             "./data/listas/lista.id.c.db"
19         );
20
21         indiceDiretoCodigo = new HashExtensivel<>(
22             ParCodigoID.class.getConstructor(),
23             4,
24             "./data/listas/lista.codigo.d.db",
25             "./data/listas/lista.codigo.c.db"
26         );
27
28         usuariolista = new ArvoreBMais<>(
29             ParIntInt.class.getConstructor(),
30             5,
31             "./data/listas/lista.usuario.db"
32         );
33     }
34
35     @Override
36     public int create(Lista l) throws Exception {
37         int id = super.create(l);
38         indiceDiretoID.create(new ParIDEndereco(id, id));
39         indiceDiretoCodigo.create(new ParCodigoID(l.getCodigo(), id));
40         usuariolista.create(new ParIntInt(l.getIdUsuario(), id));
41         return id;
42     }
```

Fonte: Elaborado pelos autores

Figura 11 – Prova de Execução: Tela de criação de uma nova lista

```

----- PresenteFácil 1.0 -----
-----
> Início > Minhas Listas

LISTAS

-- Nenhuma lista cadastrada. --

(N) Nova Lista
(R) Retornar ao menu anterior
(D) Mostrar Listas Desativadas
(A) Reativar Lista

Opção: N
N
----- PresenteFácil 1.0 -----
-----
> Início > Minhas Listas > Nova Lista

Nome da Lista: Churrasco no apartamento
Descrição: Comprei um apê novo e quero fazer um churrasco
Data limite (dd/MM/yyyy): 30/09/2025

-- Lista criada com sucesso! (Código compartilhável: gMu9zitYjA) --

----- PresenteFácil 1.0 -----
-----
> Início > Minhas Listas

----- PresenteFácil 1.0 -----
-----
> Início > Minhas Listas

LISTAS
(1) Churrasco no apartamento - 15/09/2025

(N) Nova Lista
(R) Retornar ao menu anterior
(D) Mostrar Listas Desativadas
(A) Reativar Lista

Opção: █

```

Fonte: Elaborado pelos autores

3.3 Há uma árvore B+ que registre o relacionamento 1:N entre usuários e listas?

Sim. A classe `ArquivoLista` utiliza uma `ArvoreBMais<ParIntInt>` para armazenar pares de `(idUserario, idLista)`, como mostra o código na Figura 12. A prova de que a árvore funciona é a funcionalidade "Minhas Listas", que consulta a árvore para exibir apenas as listas do usuário logado (Figura 13).

Figura 12 – Evidência em Código: Uso da Árvore B+ em ArquivoLista.java

```
1  public class ArquivoLista extends Arquivo<Lista> {
2      HashExtensivel<ParIDEndereco> indiceDiretoID;
3      HashExtensivel<ParCodigoID> indiceDiretoCodigo;
4      ArvoreBMais<ParIntInt> usuarioLista;
5
6      public ArquivoLista() throws Exception {
7          super("listas", Lista.class.getConstructor());
8
9          indiceDiretoID = new HashExtensivel<>(
10             ParIDEndereco.class.getConstructor(),
11             4,
12             "./data/listas/lista.id.d.db",
13             "./data/listas/lista.id.c.db"
14         );
15
16         indiceDiretoCodigo = new HashExtensivel<>(
17             ParCodigoID.class.getConstructor(),
18             4,
19             "./data/listas/lista.codigo.d.db",
20             "./data/listas/lista.codigo.c.db"
21         );
22
23         usuarioLista = new ArvoreBMais<>(
24             ParIntInt.class.getConstructor(),
25             5,
26             "./data/listas/lista.usuario.db"
27         );
28     }
29
30     @Override
31     public int create(Lista l) throws Exception {
32         int id = super.create(l);
33         indiceDiretoID.create(new ParIDEndereco(id, id));
34         indiceDiretoCodigo.create(new ParCodigoID(l.getCodigo(), id));
35         usuarioLista.create(new ParIntInt(l.getIdUsuario(), id));
36         return id;
37     }
```

Fonte: Elaborado pelos autores

Figura 13 – Prova de Execução: Tela "Minhas Listas" mostrando o resultado da consulta

```
----- PresenteFácil 1.0 -----  
-----  
> Início > Minhas Listas  
  
----- PresenteFácil 1.0 -----  
-----  
> Início > Minhas Listas  
  
LISTAS  
(1) Churrasco no apartamento - 15/09/2025  
(2) Jantar - 15/09/2025  
  
(N) Nova Lista  
(R) Retornar ao menu anterior  
(D) Mostrar Listas Desativadas  
(A) Reativar Lista  
  
Opção: █
```

Fonte: Elaborado pelos autores

3.4 Há uma visualização das listas por meio de um código NanoID?

Sim. A funcionalidade "Buscar Lista"(Figura 14) solicita um código ao usuário e utiliza o índice indireto para encontrá-la. A Figura 15 mostra a execução dessa busca no terminal.

Figura 14 – Evidência em Código: Método de busca por código

```
1 public void buscarListaPorCodigo(Scanner scanner, ArquivoLista arqListas) {
2     System.out.println("----- PresenteFácil 1.0 -----");
3     System.out.println("-----");
4     System.out.println("> Início > Buscar Lista\n");
5
6     try {
7         System.out.print("\nDigite o código da lista: ");
8         String codigo = scanner.nextLine();
9         Lista lista = arqListas.readByCodigo(codigo);
10
11         if (lista == null || !lista.isAtiva() ) {
12             System.out.println("\n-- Nenhuma lista encontrada com esse código. --\n");
13         } else {
14             System.out.println("\n-- Lista encontrada! --");
15             ClearConsole.clearScreen();
16             System.out.println("----- PresenteFácil 1.0 -----");
17             System.out.println("-----");
18             System.out.println("> Início > Minhas Listas > " + lista.getNome() + "\n");
19             System.out.println("Nome: " + lista.getNome());
20             System.out.println("Descrição: " + lista.getDescricao());
21             System.out.println("Data de criação: " + lista.getDataCriacao().format(formato));
22             System.out.println("Data limite: " + lista.getDataLimite().format(formato));
23             System.out.println("Código compartilhável: " + lista.getCodigo());
24             System.err.println("ativa: " + lista.isAtiva());
25             System.out.println("-----\n");
26         }
27     } catch (Exception e) {
28         System.err.println("\nErro ao buscar lista: " + e.getMessage() + "\n");
29     }
30 }
```

Fonte: Elaborado pelos autores

Figura 15 – Prova de Execução: Busca de uma lista pelo seu código NanoID

```
----- PresenteFácil 1.0 -----
-----
> Início > Buscar Lista

Digite o código da lista: gMu9zitYjA

----- PresenteFácil 1.0 -----
-----
> Início > Minhas Listas > Churrasco no apartamento

Nome: Churrasco no apartamento
Descrição: Comprei um ap? novo e quero fazer um churrasco
Data de criação: 15/09/2025
Data limite: 30/09/2025
Código compartilhável: gMu9zitYjA
ativa: true
-----

----- Menu Principal -----
-----

> Início

(1) Meus dados.....
(2) Minhas listas.....
(3) Produtos.....
(4) Buscar lista.....
(5) Desativar minha conta....
(6) Excluir minha conta.....

(S) Sair.....

Opção: █
```

Fonte: Elaborado pelos autores

3.5 O trabalho está completo e funcionando sem erros de execução?

Sim. O sistema implementa todas as funcionalidades propostas. A Figura 16 mostra o menu inicial do programa em execução, que serve como ponto de partida para todas as outras operações, demonstrando que o sistema inicia e opera corretamente.

Figura 16 – Prova de Execução: Menu inicial da aplicação

```
PS D:\Aeds3_TP1\Aeds3_TP1> javac .\Main.java
PS D:\Aeds3_TP1\Aeds3_TP1> java .\Main.java

----- PresenteFácil 1.0 -----
-----

(1) Login
(2) Novo usuário
(3) Reativar usuário

(5) Sair

Opção: █
```

Fonte: Elaborado pelos autores

3.6 O trabalho é original e não a cópia de um trabalho de outro grupo?

O trabalho é original. Todos os integrantes do grupo trabalharam arduamente para produzir esse projeto, com o objetivo de exercitar e fixar o conteúdo aprendido na disciplina de Algoritmos e Estruturas de Dados 3.

4 CONCLUSÃO

O desenvolvimento do projeto "PresenteFácil" permitiu a aplicação prática de conceitos complexos de organização de arquivos e estruturas de dados e a implementação de um sistema de persistência manual, com gerenciamento de índices por meio de Hash Extensível e Árvores B+.

REFERÊNCIAS

Referências

KUTOVA, M. **Implementação da Estrutura de Dados Árvore B+**. 2021. Código fonte fornecido na disciplina de Algoritmos e Estruturas de Dados III. Citado no cabeçalho do arquivo `ArvoreBMais.java`.

KUTOVA, M. **Implementação da Tabela Hash Extensível**. 2021. Código fonte fornecido na disciplina de Algoritmos e Estruturas de Dados III. Citado no cabeçalho do arquivo `HashExtensivel.java`.