

Laboratório de Estrutura de Dados

Relatório Roteiro Sala

Correção de Programa

Nome: Arthur Oliveira Praxedes. Matrícula: 2023208510017.

Nome: Davi Roberto Pereira Barbosa. Matrícula: 2023208510028.

Nome: Leonardo Istamilo Silva Ferreira. Matrícula: 2023208510023.

Curso: Ciência da Computação

Data: 06/06/2025

1. Introdução

A estrutura de dados conhecida como lista duplamente encadeada é amplamente utilizada em diversos contextos da Ciência da Computação por sua eficiência na manipulação de elementos em sequências dinâmicas. Ao contrário das listas simplesmente encadeadas, cada nó de uma lista duplamente encadeada mantém referências tanto para o seu sucessor quanto para seu antecessor, permitindo percursos bidirecionais e operações de inserção e remoção mais flexíveis.

O presente relatório tem como objetivo avaliar, diagnosticar e corrigir uma implementação básica de lista duplamente encadeada em Java, identificando erros de lógica, falhas na implementação dos pilares da Programação Orientada a Objetos — especialmente o encapsulamento —, e comportamentos inesperados.

Além disso, foram utilizados testes automatizados com JUnit 5 para validar os métodos da classe `DoublyLinkedList`, como `insertOrdered`, `remove`, `printForward` e `printBackward`. A abordagem prática permitiu não apenas a identificação de bugs, mas também a implementação de melhorias com base em princípios de software seguro e confiável.

Este documento apresenta os problemas encontrados, as soluções aplicadas e os resultados obtidos após as correções, demonstrando o funcionamento adequado da estrutura e o aprendizado prático de conceitos fundamentais de estruturas de dados e boas práticas em programação.

2. Descrição geral sobre o método utilizado

A metodologia adotada neste trabalho envolveu as seguintes etapas:

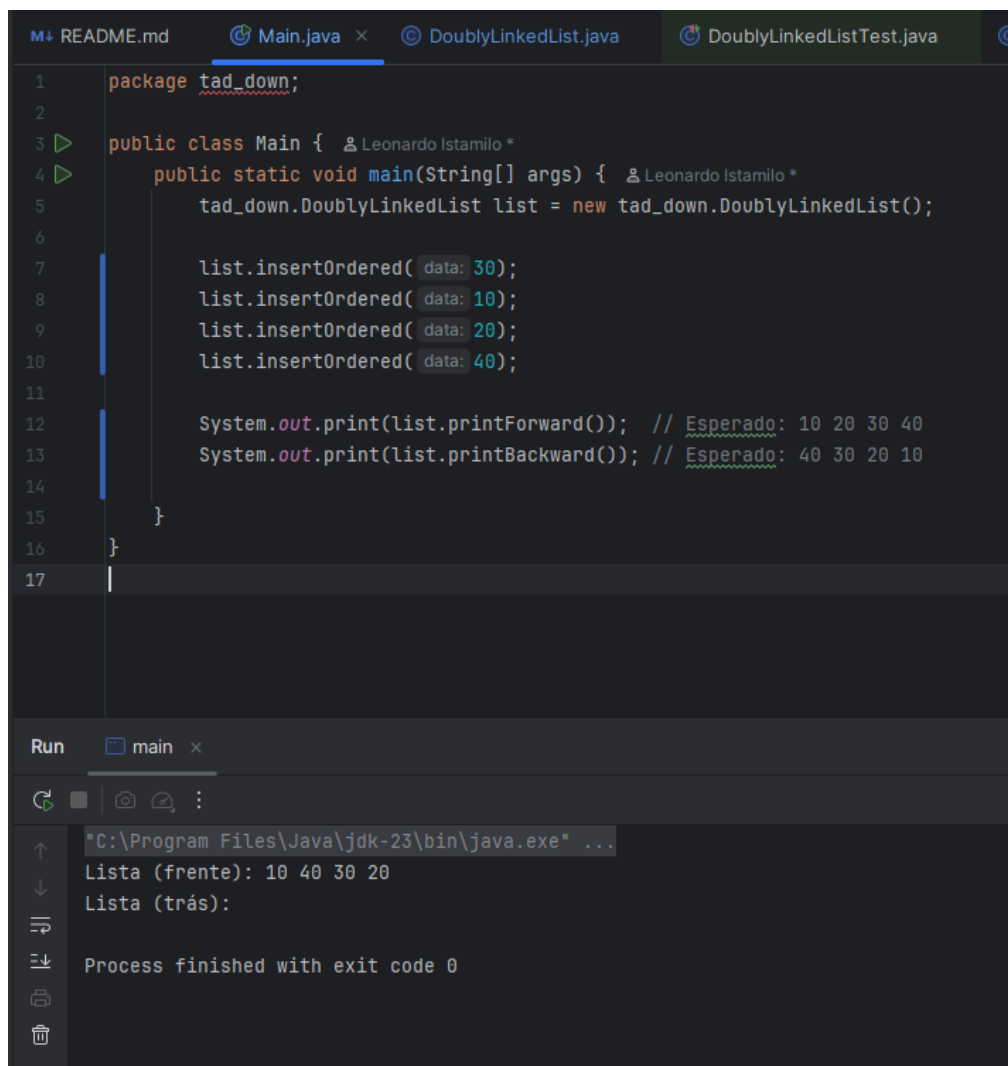
1. **Execução e análise inicial** da implementação original para observar comportamentos incorretos na inserção, remoção e exibição dos dados na lista.
2. **Aplicação de testes unitários** com JUnit 5, focando na verificação dos principais métodos da classe, permitindo a reprodução sistemática dos problemas encontrados.
3. **Depuração do código-fonte**, análise lógica das funções e verificação das interações entre os nós da estrutura.
4. **Correções pontuais e incrementais**, com base nos resultados dos testes, visando restaurar o comportamento esperado da lista em cada operação.
5. **Refatoração do código**, utilizando os princípios da POO, principalmente o **encapsulamento**, com a privatização dos atributos da classe Node, criação de métodos de acesso (getters e setters) e melhoria da legibilidade.
6. **Validação final por meio de novos testes**, incluindo casos especiais como listas com apenas um elemento, remoção de elementos inexistentes e manipulação de listas vazias.

Essa abordagem sistemática possibilitou uma compreensão prática mais aprofundada da estrutura de dados estudada, além de reforçar boas práticas de desenvolvimento, como modularidade, reutilização e testes automatizados.

3. Resultados e Análise

Ao executar o teste inicial na classe main, notou-se que o método `printBackward()` não apresentou retorno, e o método `printForward()` também não retornou a lista ordenada como esperado. (Imagem 1).

Imagem 1:



The screenshot shows an IDE with four tabs: README.md, Main.java, DoublyLinkedList.java, and DoublyLinkedListTest.java. The Main.java tab is active, displaying the following code:

```
1 package tad_down;
2
3 public class Main {
4     public static void main(String[] args) {
5         tad_down.DoublyLinkedList list = new tad_down.DoublyLinkedList();
6
7         list.insertOrdered(data: 30);
8         list.insertOrdered(data: 10);
9         list.insertOrdered(data: 20);
10        list.insertOrdered(data: 40);
11
12        System.out.print(list.printForward()); // Esperado: 10 20 30 40
13        System.out.print(list.printBackward()); // Esperado: 40 30 20 10
14
15    }
16 }
17
```

Below the code editor, the 'Run' tab is active, showing the execution output:

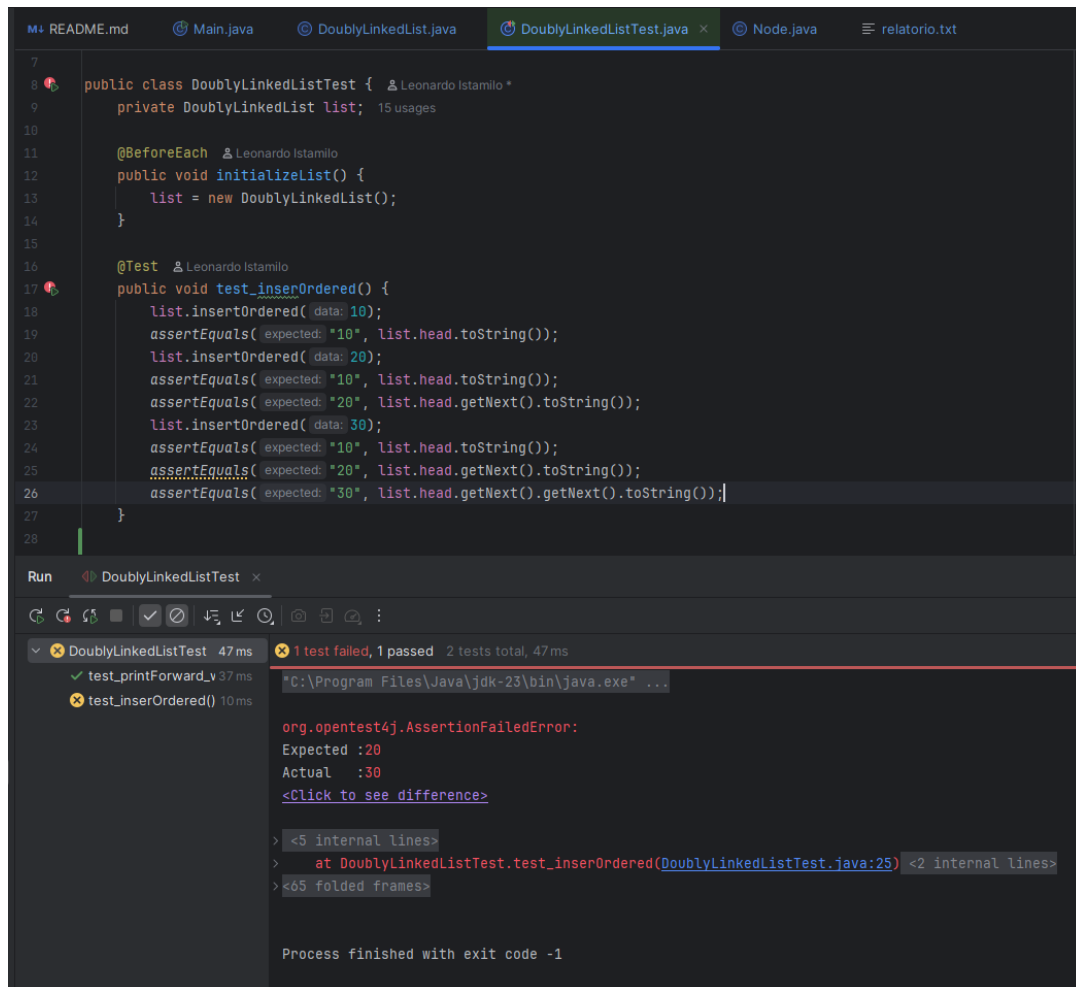
```
Run main
"C:\Program Files\Java\jdk-23\bin\java.exe" ...
Lista (frente): 10 40 30 20
Lista (trás):
Process finished with exit code 0
```

Elaboramos duas hipóteses: O método `insertOrdered()` não estava inserindo os elementos em ordem crescente ou, o método `printForward()` estava imprimindo incorretamente a lista.

Correção do método InsertOrdered()

Para testar as hipóteses anteriores, partimos para os testes unitário usando o Junit 5.8.1, onde encontramos um erro no método `insertOrdered()`. (Imagem 2).

Imagem 2:



```
7
8 public class DoublyLinkedListTest { Leonardo Istamilio *
9     private DoublyLinkedList list; 15 usages
10
11     @BeforeEach Leonardo Istamilio
12     public void initializeList() {
13         list = new DoublyLinkedList();
14     }
15
16     @Test Leonardo Istamilio
17     public void test_inserOrdered() {
18         list.insertOrdered( data: 10);
19         assertEquals( expected: "10", list.head.toString());
20         list.insertOrdered( data: 20);
21         assertEquals( expected: "10", list.head.toString());
22         assertEquals( expected: "20", list.head.getNext().toString());
23         list.insertOrdered( data: 30);
24         assertEquals( expected: "10", list.head.toString());
25         assertEquals( expected: "20", list.head.getNext().toString());
26         assertEquals( expected: "30", list.head.getNext().getNext().toString());
27     }
28 }
```

Run DoublyLinkedListTest x

1 test failed, 1 passed 2 tests total, 47ms

test_printForward_v 37 ms

test_inserOrdered() 10 ms

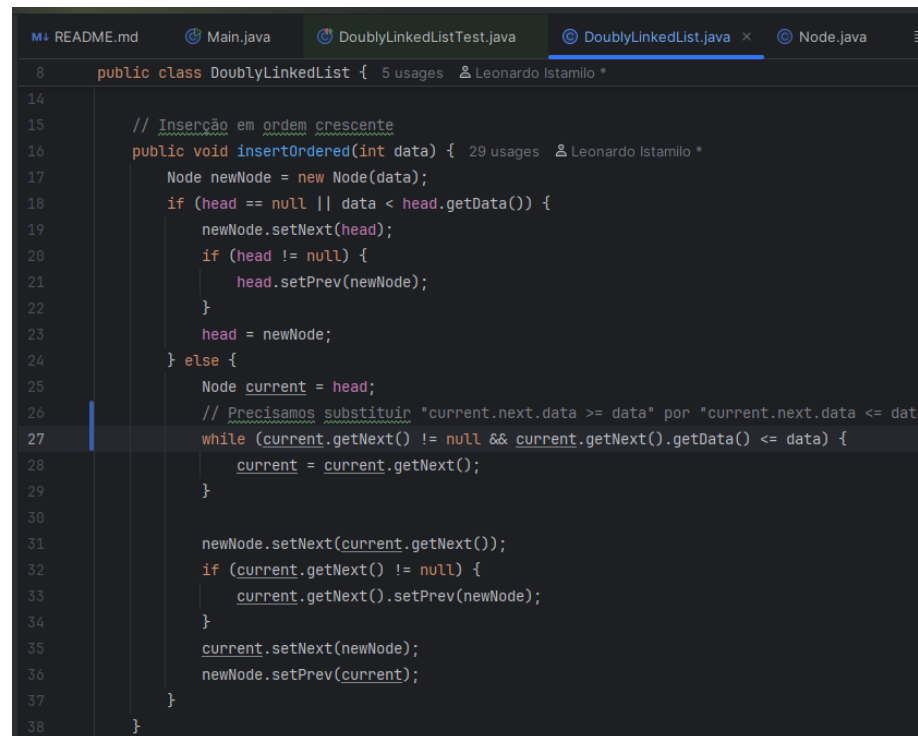
org.opentest4j.AssertionFailedError:
Expected :20
Actual :30
<Click to see difference>

> <5 internal lines>
> at DoublyLinkedListTest.test_inserOrdered(DoublyLinkedListTest.java:25) <2 internal lines>
> <65 folded frames>

Process finished with exit code -1

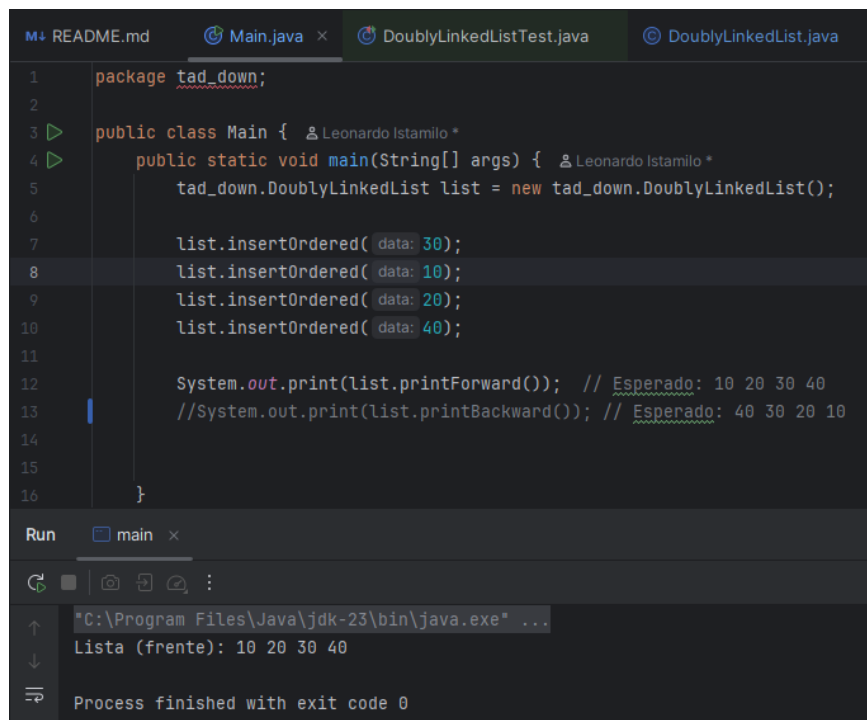
A substituição da condição `current.next.data >= data` por `current.next.data <= data` corrigiu a ordenação. (Imagens 3 e 4).

Imagem 3:



```
8 public class DoublyLinkedList { 5 usages Leonardo Istamilo *
14
15 // Inserção em ordem crescente
16 public void insertOrdered(int data) { 29 usages Leonardo Istamilo *
17     Node newNode = new Node(data);
18     if (head == null || data < head.getData()) {
19         newNode.setNext(head);
20         if (head != null) {
21             head.setPrev(newNode);
22         }
23         head = newNode;
24     } else {
25         Node current = head;
26         // Precisamos substituir "current.next.data >= data" por "current.next.data <= data"
27         while (current.getNext() != null && current.getNext().getData() <= data) {
28             current = current.getNext();
29         }
30
31         newNode.setNext(current.getNext());
32         if (current.getNext() != null) {
33             current.getNext().setPrev(newNode);
34         }
35         current.setNext(newNode);
36         newNode.setPrev(current);
37     }
38 }
```

Imagem 4:



```
1 package tad_down;
2
3 public class Main { Leonardo Istamilo *
4     public static void main(String[] args) { Leonardo Istamilo *
5         tad_down.DoublyLinkedList list = new tad_down.DoublyLinkedList();
6
7         list.insertOrdered( data: 30);
8         list.insertOrdered( data: 10);
9         list.insertOrdered( data: 20);
10        list.insertOrdered( data: 40);
11
12        System.out.print(list.printForward()); // Esperado: 10 20 30 40
13        //System.out.print(list.printBackward()); // Esperado: 40 30 20 10
14
15    }
16 }
```

Run main x

*C:\Program Files\Java\jdk-23\bin\java.exe" ...
Lista (frente): 10 20 30 40
Process finished with exit code 0

Correção do método printBackward()

Foi necessário substituir a condição `current != null` por `current.next != null` no primeiro while. Essa modificação garante que a iteração avance até o último nó, impedindo que o nó `current` seja nulo e permitindo que o próximo while seja executado corretamente. (Imagem 5).

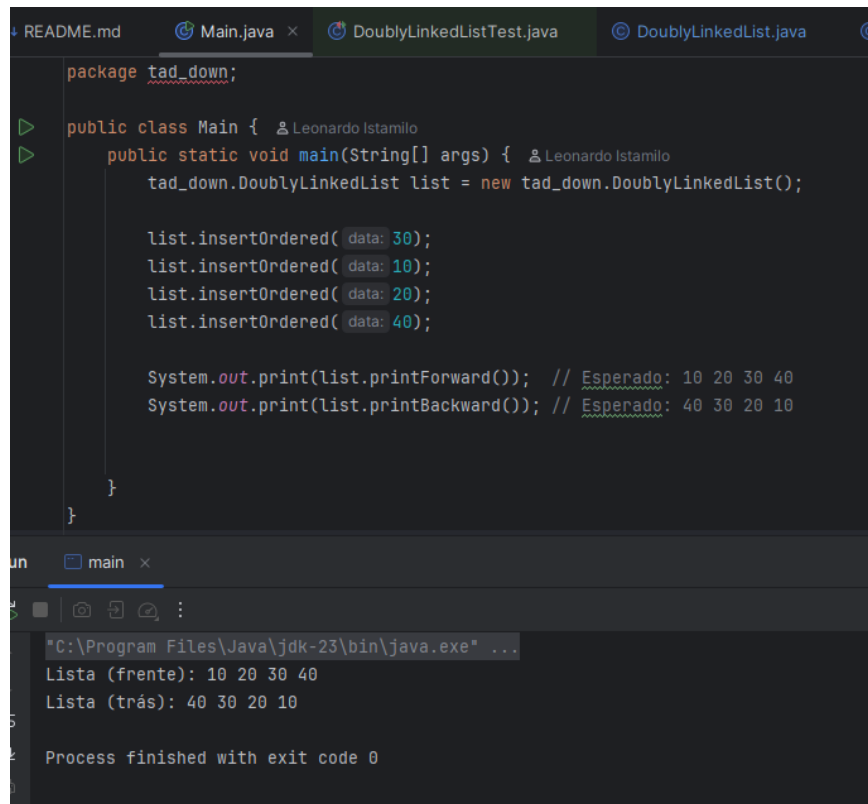
Imagem 5:



```
8 public class DoublyLinkedList { 5 usages  Leonardo Istamilo *
50
51     public String printBackward() { 6 usages  Leonardo Istamilo *
52         //int count = 0;
53         Node current = head;
54         String output = "Lista (trás): ";
55
56         //adicionada condicional para lista vazia
57         if (current == null) {
58             return output + "\n";
59         }
60
61         //precisamos substituir "current != null" por "current.getNext() != null" para
62         while (current.getNext() != null) {
63             current = current.getNext();
64         }
65
66         while (current != null) {
67             output += (current.getData() + " ");
68             current = current.getPrev();
69         }
70         return output + "\n";
71     }
72 }
```

Após essa correção, o método funcionou corretamente e retornou os valores em ordem reversa. (Imagem 6).

Imagem 6:



```
package tad_down;

public class Main {
    public static void main(String[] args) {
        tad_down.DoublyLinkedList list = new tad_down.DoublyLinkedList();

        list.insertOrdered(data: 30);
        list.insertOrdered(data: 10);
        list.insertOrdered(data: 20);
        list.insertOrdered(data: 40);

        System.out.print(list.printForward()); // Esperado: 10 20 30 40
        System.out.print(list.printBackward()); // Esperado: 40 30 20 10
    }
}
```

main

```
"C:\Program Files\Java\jdk-23\bin\java.exe" ...
Lista (frente): 10 20 30 40
Lista (trás): 40 30 20 10

Process finished with exit code 0
```

Além disso, um problema de loop infinito foi notado ao usar o `printBackward()`. Foi constatado que `head.prev` apontava para ele mesmo devido a `this.prev = this` no construtor do Node. (imagem 7).

Imagem 7:



```
package tad_down;

public class Node {
    int data;
    Node next;
    Node prev;

    public Node(int data) {
        this.data = data;
        this.next = null;
        this.prev = null; // Substituímos "this.prev = this" por "this.prev = null"
    }
}
```

Aplicação de Encapsulamento (POO)

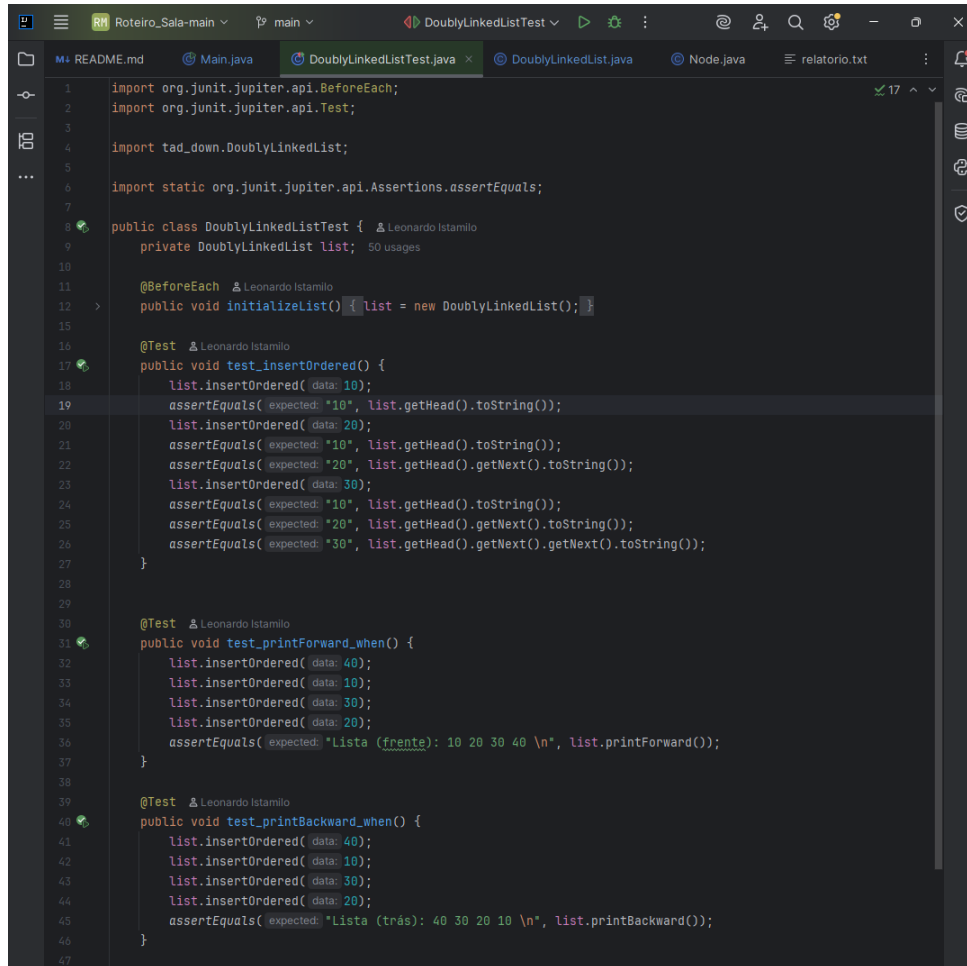
- Atributos da classe Node.java tornaram-se privados.;
- Adicionados os métodos: getData(), getNext(), getPrev(), setNext(), setPrev();
- O uso de head.data, head.next e head.prev foi substituído por chamadas apropriadas via métodos públicos;
- Implementação dos métodos toString(), hashCode() e equals() em Node.java.

Novos testes adicionados

Foram criados testes adicionais para:

- Remoção em lista com apenas 1 elemento;
- Remoção de elemento inexistente;
- Impressão de uma lista vazia;
- Inserção de elementos duplicados;
- Inserção e remoção em diferentes posições (início, meio, fim);
(Imagens 7 e 8).

Imagem 8:



The image shows a screenshot of an IDE window titled "Roteiro_Sala-main". The active file is "DoublyLinkedListTest.java". The code is a Java test class for a "DoublyLinkedList". It includes imports for JUnit and the "DoublyLinkedList" class. The class "DoublyLinkedListTest" has a private "DoublyLinkedList list" and three test methods: "initializeList()", "test_insertOrdered()", and "test_printForward_when()", "test_printBackward_when()". The "initializeList()" method initializes the list. The "test_insertOrdered()" method tests inserting elements 10, 20, and 30 in order, with assertions for the head and next pointers. The "test_printForward_when()" method tests printing the list forward after inserting 40, 10, 30, and 20. The "test_printBackward_when()" method tests printing the list backward after inserting 40, 10, 30, and 20. The IDE interface includes a sidebar with a file explorer, a top toolbar with various icons, and a right sidebar with a search and other utility icons.

```
1 import org.junit.jupiter.api.BeforeEach;
2 import org.junit.jupiter.api.Test;
3
4 import tad_down.DoublyLinkedList;
5
6 import static org.junit.jupiter.api.Assertions.assertEquals;
7
8 public class DoublyLinkedListTest {
9     private DoublyLinkedList list;
10
11     @BeforeEach
12     public void initializeList() { list = new DoublyLinkedList(); }
13
14     @Test
15     public void test_insertOrdered() {
16         list.insertOrdered(10);
17         assertEquals("10", list.getHead().toString());
18         list.insertOrdered(20);
19         assertEquals("10", list.getHead().toString());
20         assertEquals("20", list.getHead().getNext().toString());
21         list.insertOrdered(30);
22         assertEquals("10", list.getHead().toString());
23         assertEquals("20", list.getHead().getNext().toString());
24         assertEquals("30", list.getHead().getNext().getNext().toString());
25     }
26
27     @Test
28     public void test_printForward_when() {
29         list.insertOrdered(40);
30         list.insertOrdered(10);
31         list.insertOrdered(30);
32         list.insertOrdered(20);
33         assertEquals("Lista (frente): 10 20 30 40 \n", list.printForward());
34     }
35
36     @Test
37     public void test_printBackward_when() {
38         list.insertOrdered(40);
39         list.insertOrdered(10);
40         list.insertOrdered(30);
41         list.insertOrdered(20);
42         assertEquals("Lista (trás): 40 30 20 10 \n", list.printBackward());
43     }
44 }
```

Imagem 9:

```
47
48 @Test @Leonardo Istamilio
49 public void test_remove() {
50     list.insertOrdered( data: 40);
51     list.insertOrdered( data: 10);
52     list.insertOrdered( data: 30);
53     list.insertOrdered( data: 20);
54
55     list.remove( data: 10);
56     assertEquals( expected: "Lista (frente): 20 30 40 \n", list.printForward()); // remoção do primeiro elemento
57     list.insertOrdered( data: 10);
58
59     list.remove( data: 20);
60     assertEquals( expected: "Lista (frente): 10 30 40 \n", list.printForward()); // remoção de um elemento do meio
61     list.insertOrdered( data: 20);
62
63     list.remove( data: 40);
64     assertEquals( expected: "Lista (frente): 10 20 30 \n", list.printForward()); // remoção do ultimo elemento
65 }
66
67 @Test @Leonardo Istamilio
68 public void test_remove_singleElement() {
69     list.insertOrdered( data: 10);
70     list.remove( data: 10);
71     assertEquals( expected: "Lista (frente): \n", list.printForward());
72     assertEquals( expected: "Lista (trás): \n", list.printBackward());
73 }
74
75 @Test @Leonardo Istamilio
76 public void test_remove_nonExistentElement() {
77     list.insertOrdered( data: 10);
78     list.insertOrdered( data: 20);
79     list.insertOrdered( data: 30);
80     list.remove( data: 100); // 100 não existe
81     assertEquals( expected: "Lista (frente): 10 20 30 \n", list.printForward());
82     assertEquals( expected: "Lista (trás): 30 20 10 \n", list.printBackward());
83 }
84
85 @Test @Leonardo Istamilio
86 public void test_print_onEmptyList() {
87     assertEquals( expected: "Lista (frente): \n", list.printForward());
88     assertEquals( expected: "Lista (trás): \n", list.printBackward());
89 }
90
91 @Test @Leonardo Istamilio
92 public void test_insertOrdered_duplicates() {
93     list.insertOrdered( data: 20);
94     list.insertOrdered( data: 10);
95     list.insertOrdered( data: 20);
96     list.insertOrdered( data: 10);
97     assertEquals( expected: "Lista (frente): 10 10 20 20 \n", list.printForward());
98     assertEquals( expected: "Lista (trás): 20 20 10 10 \n", list.printBackward());
99 }
100 }
101
```

Run DoublyLinkedListTest

Test	Duration
✓ DoublyLinkedListTest	53 ms
✓ test_remove_nonExistentElement()	42 ms
✓ test_remove_singleElement()	2 ms
✓ test_printBackward_when()	2 ms
✓ test_printForward_when()	2 ms
✓ test_remove()	1 ms
✓ test_insertOrdered_duplicates()	1 ms
✓ test_print_onEmptyList()	1 ms
✓ test_insertOrdered()	2 ms

✓ 8 tests passed 8 tests total, 53ms

"C:\Program Files\Java\jdk-23\bin\java.exe" . . .

Process finished with exit code 0

Roteiro_Sala-main > tests > DoublyLinkedListTest 19:55 LF UTF-8 4 spaces