Algoritmos e Estrutura de Dados

Relatório Trabalho Prático 3

Gabriel Moreira Silva - 203

- **Enunciado**
  - Esse trabalho consiste em implementar uma estrutura de dados recursiva em Java usando herança. A estrutura a ser representada será um conjunto
  - É necessário a implementação de 6 classes, que estendem a classe abstrata Conjunto<T> (documentação 1), onde cada classe representa uma operação a ser realizada sobre os conjuntos
    - ConjuntoVazio (conjunto sem nenhum elemento)
    - ConjuntoElemento (adicionar um elemento ao conjunto)
    - Uniao (união de dois conjuntos)
    - Intersecao (interseção de dois conjuntos)
    - Diferenca (diferença entre dois conjuntos)
    - Complemento (complemento de um conjunto)
  - A classe Conjunto<T> é genérica, ou seja, o conjunto pode ser de qualquer tipo T. Ela já usa os construtores de cada classe, então eles também precisam ser implementados
  - O único método abstrato é o contemElemento, que também deve ser implementado em cada classe derivada.
  - Enunciado Completo

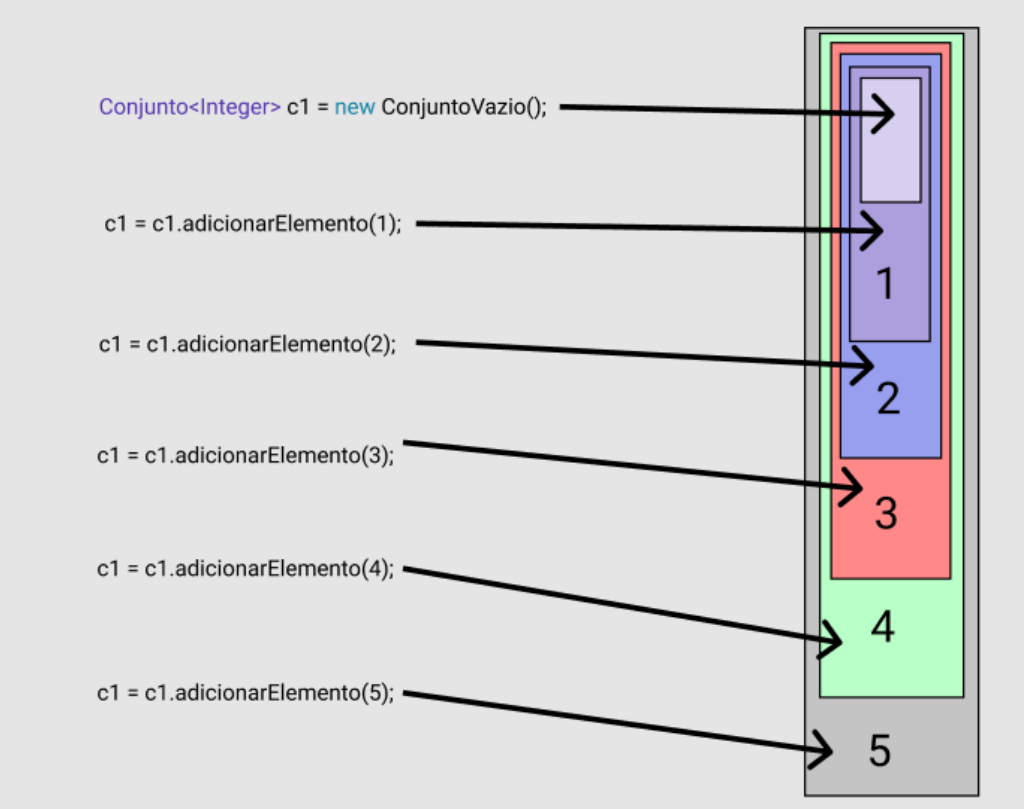
- **Desenvolvimento**
  - Para fins de compreensão do código, o nome da classe “ConjuntoElemento” foi alterado para “AdicionarElemento”
  - O projeto completo está totalmente disponível: Repositório no github
  - Foram criadas 8 classes no total
    - **Conjunto<T>**
      - ➢ Classe genérica, ou seja, um conjunto pode ser de qualquer tipo <T>.
      - ➢ Possui um único método abstrato (contemElemento(T elemento)) que é implementado em cada um das classes que estendem Conjunto<T>.
      - ➢ Possui outros 5 métodos que representam as possibilidades de operação com os conjuntos.

```
public abstract class Conjunto<T>
{
    public abstract Boolean contemElemento(T elemento);

    Conjunto<T> adicionarElemento(T elemento)
    {
        return new AdicionarElemento<>(elemento, this);
    }

    Conjunto<T> uniao(Conjunto<T> conjunto)
    {
        return new Uniao<>(this, conjunto);
    }

    Conjunto<T> intersecao(Conjunto<T> conjunto)
    {
        return new Intersecao<>(this, conjunto);
    }

    Conjunto<T> diferenca(Conjunto<T> conjunto)
    {
        return new Diferenca<>(this, conjunto);
    }

    Conjunto<T> complemento()
    {
        return new Complemento<>(this);
    }
}
```

- **ConjuntoVazio<T>**
  - ➢ Como o seu nome diz, é um conjunto vazio.
  - ➢ O método contemElemento sempre retorna false já que não há nenhum elemento.

```
class ConjuntoVazio<T> extends Conjunto<T>
{
    @Override
    public Boolean contemElemento(T elemento)
    {
        return false;
    }
}
```

- **AdicionarElemento<T>**
  - ➢ Responsável por adicionar um elemento do tipo T no conjunto
  - ➢ É importante ressaltar que quando “adicionamos” um elemento ao conjunto estamos, na verdade, criando um novo objeto que recebe como conteúdo:
    - Um conjunto: como definido na classe Conjunto, ele sempre recebe o próprio objeto.
    - Um elemento: um valor do tipo T
    - Dessa forma, o conjunto passa a ser a união desses dois valores, (conjunto U valor), como mostra a imagem abaixo



- Nesse caso, o método contemElemento() é implementado de forma que retorna true quando o valor buscado é encontrado no próprio objeto ou em algum dos subconjuntos deste objeto

```
class AdicionarElemento<T> extends Conjunto<T>
{
    private final T valor;

    private final Conjunto<T> conjunto;

    public AdicionarElemento(T valor, Conjunto<T> conjunto)
    {
        this.valor = valor;
        this.conjunto = conjunto;
    }

    @Override
    public Boolean contemElemento(T elemento)
    {
        return elemento == valor || conjunto.contemElemento(elemento);
    }
}
```

- **Uniao<T>**
  - Seu funcionamento é parecido com a AdicionarElemento<T>, mas nesse caso, a união é feita entre dois conjuntos (que também pode ser algum dos outros tipos que estendem a classe Conjunto<T>).
  - Dessa forma, o objeto Uniao<T> passa a ser a união desses dois conjuntos (conjunto1 U conjunto2)
  - O método contemElemento() retorna true sempre que o elemento buscado está em pelo menos um dos dois subconjuntos

```
class Uniao<T> extends Conjunto<T>
{
    private Conjunto<T> conjunto1 = new ConjuntoVazio<>();
    private Conjunto<T> conjunto2 = new ConjuntoVazio<>();

    public Uniao(Conjunto<T> c1, Conjunto<T> c2){
```

```
        this.conjunto1 = c1;
        this.conjunto2 = c2;
    }

    @Override
    public Boolean contemElemento(T elemento)
    {
      return conjunto1.contemElemento(elemento)
      || conjunto2.contemElemento(elemento);
    }
}
```

- **Intersecao<T>**
  - ➢ Seu funcionamento é parecido com a União<T>, onde há uma união entre dois conjuntos (conjunto1 U conjunto2)
  - ➢ A diferença está no método contemElemento(), onde ele retorna true quando o elemento buscado está obrigatoriamente nos dois subconjuntos.

```
class Intersecao<T> extends Conjunto<T>
{
    private Conjunto<T> conjunto1 = new ConjuntoVazio<>();
    private Conjunto<T> conjunto2 = new ConjuntoVazio<>();

    public Intersecao(Conjunto<T> c1, Conjunto<T> c2)
    {
        this.conjunto1 = c1;
        this.conjunto2 = c2;
    }

    @Override
    public Boolean contemElemento(T elemento)
    {
      return conjunto1.contemElemento(elemento)
      && conjunto2.contemElemento(elemento);
    }
}
```

- **Diferenca<T>**
  - ➢ Seu funcionamento é parecido com a União<T>, onde há uma união entre dois conjuntos (conjunto1 U conjunto2).
  - ➢ Nesse caso, a diferença está no método contemElemento(), que nesse caso retorna true quando um elemento está no conjunto1 e não está no conjunto2 (conjunto1 - conjunto2)

```
class Diferenca<T> extends Conjunto<T>
{
    private Conjunto<T> conjunto1 = new ConjuntoVazio<>();
    private Conjunto<T> conjunto2 = new ConjuntoVazio<>();

    public Diferenca(Conjunto<T> c1, Conjunto<T> c2){
        this.conjunto1 = c1;
        this.conjunto2 = c2;
    }

    @Override
    public Boolean contemElemento(T elemento)
    {
      return conjunto1.contemElemento(elemento)
      && !conjunto2.contemElemento(elemento);
    }
}
```

- **Complemento<T>**
  - ➢ Essa classe, diferentemente das outras, recebe apenas um Conjunto<T>, que é sempre o próprio objeto.
  - ➢ Nesse caso, o método contemElemento() retorna verdadeiro caso o elemento buscado não esteja no conjunto1.

```
class Complemento<T> extends Conjunto<T>
{
    private Conjunto<T> conjunto1 = new ConjuntoVazio<>();

    public Complemento(Conjunto<T> c1){
        this.conjunto1 = c1;
    }
```

```
    @Override
    public Boolean contemElemento(T elemento) {
        return !conjunto1.contemElemento(elemento);
    }
}
```

- **Main**
  - ➢ Ela cria e executa uma série de operações com os conjuntos
  - ➢ No total, são 6 conjuntos criados, todos do tipo integer:
    - c1 = {1,2,3,4,5}
    - c2 = {1,3,5,7,9}
    - c3 = (c1 U c2) = {1,2,3,4,5,7,9}
    - c4 = c1 ∩ c2 = {1, 3, 5}
    - c5 = c1 - c2 = {2, 4}
    - c6 = complemento(c1) = conjuntoUniverso - c1
  - ➢ Além disso, imprime o resultado de algumas buscas por elementos, relatando assim se esse elemento pertence (∈) ou não (∉) ao conjunto

```
public class Main
{
    public static void main(String[] args)
    {
        Conjunto<Integer> c1 = new ConjuntoVazio<>();

        // c1 = {1, 2, 3, 4, 5}
        c1 = c1.adicionarElemento(1);
        c1 = c1.adicionarElemento(2);
        c1 = c1.adicionarElemento(3);
        c1 = c1.adicionarElemento(4);
        c1 = c1.adicionarElemento(5);

        Conjunto<Integer> c2 = new ConjuntoVazio<>();

        // c2 = {1, 3, 5, 7, 9}
        c2 = c2.adicionarElemento(1);
        c2 = c2.adicionarElemento(3);
```

```
        c2 = c2.adicionarElemento(5);
        c2 = c2.adicionarElemento(7);
        c2 = c2.adicionarElemento(9);

        System.out.println(c1.contemElemento(2)? "2 ∈ c1" :"2 ∉ c1");
        System.out.println(c1.contemElemento(7)? "7 ∈ c1" :"7 ∉ c1");

        System.out.println(c2.contemElemento(7)? "7 ∈ c2" :"7 ∉ c2");
        System.out.println(c2.contemElemento(8)? "8 ∈ c2" :"8 ∉ c2");

        // c3 = c1 U c2 = {1, 2, 3, 4, 5, 7, 9}
        Conjunto<Integer> c3 = c1.uniao(c2);

        System.out.println(c3.contemElemento(5)? "5 ∈ c3" :"5 ∉ c3");
        System.out.println(c3.contemElemento(6)? "6 ∈ c3" :"6 ∉ c3");

        // c4 = c1 ∩ c2 = {1, 3, 5}
        Conjunto<Integer> c4 = c1.intersecao(c2);

        System.out.println(c4.contemElemento(3)? "3 ∈ c4" :"3 ∉ c4");
        System.out.println(c4.contemElemento(4)? "4 ∈ c4" :"4 ∉ c4");

        // c5 = c1 - c2 = {2, 4}
        Conjunto<Integer> c5 = c1.diferenca(c2);

        System.out.println(c5.contemElemento(3)? "3 ∈ c5" :"3 ∉ c5");
        System.out.println(c5.contemElemento(4)? "4 ∈ c5" :"4 ∉ c5");

        // c6 = complemento(c1)
        Conjunto<Integer> c6 = c1.complemento();

        System.out.println(c6.contemElemento(2)? "2 ∈ c6" :"2 ∉ c6");
        System.out.println(c6.contemElemento(7)? "7 ∈ c6" :"7 ∉ c6");
    }
}
```

- **Execução**
  - Ao executar a função main da classe Main, obtemos o seguinte resultado:
  - Comparando com os conjuntos criados, concluímos que os resultados são corretos
  - O compilador não alega nenhum erro ou warn durante a execução

Resultado	Certo?	Conjuntos Criados
2 ∈ c1	SIM	c1 = {1,2,3,4,5}
7 ∉ c1	SIM	
7 ∈ c2	SIM	c2 = {1,3,5,7,9}
8 ∉ c2	SIM	
5 ∈ c3	SIM	c3 = {1,2,3,4,5,7,9}
6 ∉ c3	SIM	
3 ∈ c4	SIM	c4 = {1, 3, 5}
4 ∉ c4	SIM	
3 ∉ c5	SIM	c5 = {2, 4}
4 ∈ c5	SIM	
2 ∉ c6	SIM	c6 = conjuntoUniverso - c1
7 ∈ c6	SIM	

- **Conclusão**
  - Um fato interessante sobre a classe Conjunto e das classes que a estendem é o tipo T ser genérico, possibilitando que o eles sejam usados para qualquer tipo básico do java.
  - Durante o desenvolvimento, não tive grandes problemas em relação à linguagem e ao paradigma de orientação aos objetos.
  - Vejo que esse trabalho contribuiu para a compreensão de alguns conceitos como a própria linguagem Java, a interação entre objetos e sobre o próprio paradigma de Orientação a Objetos.