Gabarito da Lista de Exercícios 5

1. Considere uma lista de valores inteiros e implemente um algoritmo que receba como parâmetro dois valores (n_1 e n_2) e uma lista, e insira o valor n_2 após o nó que contém o valor n_1.

```
/**
 * Adiciona um dado elemento (sucesso) após um determinado elemento
    existente na lista.
 *
 * @param elemento
 *            Elemento existente na lista e deve ser ter como sucessor
    um dado elemento.
 * @param sucessor
 *            Elemento a ser colocado após um dado elemento existente
    na lista.
 */
public void adicionarApos(E elemento, E sucessor)
{
  No<E> n1 = this.pesquisar(elemento);
  if (n1 != null)
  {
    n1.setProximo(new No<E>(sucessor, n1.getProximo()));
  }
}
```

2. Construa um algoritmo que receba como parâmetros uma lista e um valor, valor este que representa a posição de um **nó** na lista. O algoritmo deverá retornar as informações contidas neste nó e a lista resultante da exclusão deste nó.

```
/**
 * Remove o elemento que está em uma dada posição na lista.
 * @param posicao Posição do elemento a ser removido.
 * @return O elemento que estava na posição removida ou <code>null</
    code> quando a posição informada não existe.
 */
public E remover(int posicao)
{
  No<E> no = this.no(posicao);

  if (no != null)
  {
    this.remover(no.getElemento());
    return no.getElemento();
  }
  return null;
}
```

3. Construa um algoritmo que receba, como parâmetro, o endereço do primeiro nó de uma lista encadeada e um valor. O algoritmo deve retornar:
    1. O número total de nós da lista;
    2. O número de nós da lista que possuem em seu conteúdo o valor passado como parâmetro e sua respectiva posição na lista;
    3. O número de nós que possuem em seu conteúdo valores maiores do que o valor passado como argumento.
4. Construa um algoritmo que receba como parâmetros a referência para o primeiro nó de uma lista encadeada e dois valores, e retorne:
    1. A lista resultante da troca de todas as ocorrências do primeiro valor pelo segundo, ambos passados como parâmetros;
    2. Número total de troca efetuadas.

```java
/**
 * Substitui todas as ocorrências do elemento E1 pelo elemento E2 e
    retorna a quantidade de substituições.
 * @param e1 Elemento a ser substituído.
 * @param e2 Elemento substituto.
 * @return A quantidade de substituições.
 */
public int replace(E e1, E e2)
{
  int cont = 0;
  if (!this.isEmpty())
  {
    No<E> anterior = null;
    No<E> atual = this.primeiro;

    for (int i = 0; i < this.tamanho; i++)
    {
      if (atual.equals(new No<E>(e2)))
      {
        if (anterior != null)
        {
          anterior.setProximo(new No<E>(e2, atual.getProximo()));
          cont++;
        }
      }
      anterior = atual;
      atual = atual.getProximo();
    }
  }
  return cont;
}
```

5. Construa um algoritmo que receba como parâmetro duas listas e um número inteiro N e retorne a lista resultante da inserção da segunda lista na primeira, sequencialmente, a partir da posição N da primeira lista.

```
/**
 * Concatena duas listas a partir de uma dada posição.
 *
 * @param l1 Lista a ser concatenada a partir.
 * @param l2 Lista a ser concatenada a partir de uma dada posição da
     lista l1.
 * @param posicao Posição a partir da qual a lista l2 deve ser
     concatenada com a lista l2.
 */
public void concatenar(ListaEncadeada<E> l1, ListaEncadeada<E> l2, int
     posicao)
{
  if (!l1.isEmpty())
  {
    No<E> no = l1.primeiro();
    for (int i = 0; i < posicao; i++)
    {
      no = no.proximo;
    }

    l2.primeiro().setProximo(no.getProximo());
    no.setProximo(l2.primeiro());
  }
}
```

6. Construa um algoritmo que receba como parâmetro duas listas encadeadas ordenadas e retorne a lista resultante da combinação das duas sendo que a lista resultante também deve estar ordenada.
7. Construa um algoritmo que receba como parâmetro duas listas encadeadas e retorne um valor lógico que indique se as duas listas são idênticas.

```
/**
 * Verifica se duas listas são iguais.
 * @param l1 Lista l1 a ser comparada com a lista l2.
 * @param l2 Lista l2 a ser comparada com a lista l1.
 * @return <code>true</code> se as listas forem iguais ou <code>false
     </code> caso contrário.
 */
public boolean iguais(ListaEncadeada<E> l1, ListaEncadeada<E> l2)
{
  if (l1 == null && l2 == null)
    return true;

  if (l1.getTamanho() != l2.getTamanho())
    return false;
```

```
15      if (!l1.isEmpty())
        {
17        No<E> no1 = l1.primeiro;
          No<E> no2 = l2.primeiro;
19
          while (no1 != null)
21        {
            if (!no1.equals(no2))
23            return false;

25          no1 = no1.getProximo();
            no2 = no2.getProximo();
27        }
        }
29      return true;
    }
```

8. [*Desafio*] Polinômios podem ser representados por meio de listas encadeadas, cujos nós são objetos com três atributos: coeficiente, expoente e uma referência ao nó seguinte. Construa um algoritmo que receba a variável $X \in R$ como parâmetro, e retorne o resultado do calculo de $p(x)$.

A Listagem 1, apresenta o código-fonte completo de uma implementação de Lista Encadeada simples em Java juntamente com os métodos de todas as questões dessa lista.

```
1  package br.projecao.ed.dinamica;

3  /**
    * Implementação de Lista Encadeada Simples com alocação dinâmica.
5   *
    * @author alessandro
7   * @param <E>
    *            Tipo do valor a ser armazenado na lista.
9   */
   public class ListaEncadeada<E>
11 {
     /**
13    * Referência ao primeiro elemento da {@link ListaEncadeada}
      */
15   private No<E> primeiro;

17   /**
      * Referência ao último elemento da {@link ListaEncadeada}
19    */
     private No<E> ultimo;
21
     /**
```

```
23      * Tamanho da {@link ListaEncadeada}.
24      */
25     private int tamanho = -1;
26
27     private static class No<E>
28     {
29       private E elemento;
30
31       private No<E> proximo;
32
33       public No(E elemento)
34       {
35         this(elemento, null);
36       }
37
38       public No(E elemento, No<E> proximo)
39       {
40         this.elemento = elemento;
41         this.proximo = proximo;
42       }
43
44       public void setProximo(No<E> no)
45       {
46         this.proximo = no;
47       }
48
49       public E getElemento()
50       {
51         return elemento;
52       }
53
54       public No<E> getProximo()
55       {
56         return proximo;
57       }
58
59       @Override
60       public boolean equals(Object obj)
61       {
62         if (obj == this)
63           return true;
64
65         if (!(obj instanceof No))
66           return false;
67
68         No<?> other = (No<?>) obj;
69
70         if (this.getElemento() == null && other.getElemento() == null)
71           return true;
72
73         if (this.getElemento() != null && this.getElemento().equals(other.
74             getElemento()))
75           return true;
```

```
      return false;
    }

    @Override
    public int hashCode()
    {
      return this.getElemento() != null ? this.getElemento().hashCode() :
          super.hashCode();
    }
  }

  /**
   * Adiciona um dado elemento no final da lista.
   *
   * @param elemento
   *            Elemento a ser adicionado a lista.
   */
  public void adicionar(E elemento)
  {
    if (this.isEmpty())
    {
      this.ultimo = this.primeiro = new No<E>(elemento, this.primeiro);
    } else
    {
      No<E> no = new No<E>(elemento);
      this.ultimo.setProximo(no);
      this.ultimo = no;
    }
    this.tamanho++;
  }

  /**
   * Adiciona um dado elemento (sucesso) após um determinado elemento
       existente na lista.
   *
   * @param elemento
   *            Elemento existente na lista e deve ser ter como sucessor um
       dado elemento.
   * @param sucessor
   *            Elemento a ser colocado após um dado elemento existente na
       lista.
   */
  public void adicionarApos(E elemento, E sucessor)
  {
    No<E> n1 = this.pesquisar(elemento);
    if (n1 != null)
    {
      n1.setProximo(new No<E>(sucessor, n1.getProximo()));
    }
  }

  /**
   * Concatena a lista l2 a essa lista a partir da posição informada.
   *
```

```
     * @param l2
     *            Lista a ser concatenada com essa lista a partir de uma dada
        posição.
     * @param posicao
     *            Posição a partir da qual a lista l2 deve ser concatenada.
     */
    public void concatenar(ListaEncadeada<E> l2, int posicao)
    {
      this.concatenar(this, l2, posicao);
    }

    /**
     * Concatena duas listas a partir de uma dada posição.
     *
     * @param l1
     *            Lista a ser concatenada a partir.
     * @param l2
     *            Lista a ser concatenada a partir de uma dada posição da lista
        l1.
     * @param posicao
     *            posição a partir da qual a lista l2 deve ser concatenada com a
         lista l2.
     */
    public void concatenar(ListaEncadeada<E> l1, ListaEncadeada<E> l2, int
        posicao)
    {
      if (!l1.isEmpty())
      {
        No<E> no = l1.primeiro();
        for (int i = 0; i < posicao; i++)
        {
          no = no.proximo;
        }

        l2.primeiro().setProximo(no.getProximo());
        no.setProximo(l2.primeiro());
      }
    }

    /**
     * Remove um dado elemento da lista se o elemento pertencer a lista.
     *
     * @param elemento
     *            Elemento que deve ser removido.
     */
    public void remover(E elemento)
    {

      if (!this.isEmpty())
      {
        No<E> no_anterior_ao_que_sera_removido = null;
        No<E> no_a_remover = this.primeiro;
        while (no_a_remover != null && !no_a_remover.getElemento().equals(
            elemento))
```

```
175         {
              no_anterior_ao_que_sera_removido = no_a_remover;
177           no_a_remover = no_a_remover.getProximo();
            }
179
            if (no_anterior_ao_que_sera_removido != null)
181         {
              no_anterior_ao_que_sera_removido.setProximo(no_a_remover.getProximo())
                  ;
183           this.tamanho--;
            }
185       }
    }
187
    /**
189  * Remove o elemento que está em uma dada posição na lista.
     *
191  * @param posicao
     *            Posição do elemento a ser removido.
193  * @return O elemento que estava na posição removida ou <code>null</code>
         quando a posição
     *         informada não existe.
195  */
    public E remover(int posicao)
197 {
      No<E> no = this.no(posicao);
199
      if (no != null)
201   {
        this.remover(no.getElemento());
203     return no.getElemento();
      }
205   return null;
    }
207
    /**
209  * Retorna <code>true</code> se um dado elemento pertence a {@link
         ListaEncadeada} ou
     * <code>false</code> caso contrário.
211  *
     * @param elemento
213  *            Elemento a ser verificado se ele pertence a {@link
         ListaEncadeada}
     * @return <code>true</code> se o elemento pertence a {@link ListaEncadeada}
          ou
215  *         <code>false</code> caso contrário.
     */
217 public boolean contem(E elemento)
    {
219   return this.pesquisar(elemento) != null;
    }
221
    /**
223  * Verifica se duas listas são iguais.
```

```
     *
225  * @param l1
     *            Lista l1 a ser comparada com a lista l2.
227  * @param l2
     *            Lista l2 a ser comparada com a lista l1.
229  * @return <code>true</code> se as listas forem iguais ou <code>false</code>
         caso contrário.
     */
231  public boolean iguais(ListaEncadeada<E> l1, ListaEncadeada<E> l2)
     {
233    if (l1 == null && l2 == null)
         return true;
235
       if (l1.getTamanho() != l2.getTamanho())
237      return false;

239    if (!l1.isEmpty())
       {
241      No<E> no1 = l1.primeiro;
         No<E> no2 = l2.primeiro;
243
         while (no1 != null)
245      {
           if (!no1.equals(no2))
247          return false;

249        no1 = no1.getProximo();
           no2 = no2.getProximo();
251      }
       }
253    return true;
     }
255
     public boolean iguais(ListaEncadeada<E> l2)
257  {
       return this.iguais(this, l2);
259  }

261  /**
      * Substitui todas as ocorrências do elemento E1 pelo elemento E2 e retorna
          a quantidade de
263   * substituições.
      *
265   * @param e1
      *            Elemento a ser substituído.
267   * @param e2
      *            Elemento substituto.
269   * @return A quantidade de substituições.
      */
271  public int replace(E e1, E e2)
     {
273    int cont = 0;
       if (!this.isEmpty())
275    {
```

```
        No<E> anterior = null;
277     No<E> atual = this.primeiro;

279     for (int i = 0; i < this.tamanho; i++)
        {
281       if (atual.equals(new No<E>(e2)))
          {
283         if (anterior != null)
            {
285           anterior.setProximo(new No<E>(e2, atual.getProximo()));
              cont++;
287         }
          }
289       anterior = atual;
          atual = atual.getProximo();
291     }
      }
293   return cont;
    }
295
    /**
297  * Retorna <code>true</code> se a lista estiver vazia ou <code>false</code>
         caso contrário.
     *
299  * @return <code>true</code> se a lista estiver vazia ou <code>false</code>
         caso contrário.
     */
301 public boolean isEmpty()
    {
303   return this.tamanho < 0;
    }
305
    /**
307  * Retorna o tamanho da lista.
     *
309  * @return O tamanho da lista.
     */
311 public int getTamanho()
    {
313   return isEmpty() ? 0 : tamanho + 1;
    }
315
    /**
317  * Retorna o primeiro elemento da lista ou <code>null</code> se a lista
         estiver vazia.
     *
319  * @return O primeiro elemento da lista ou <code>null</code> se a lista
         estiver vazia.
     */
321 public E getPrimeiro()
    {
323   return !isEmpty() ? this.primeiro.getElemento() : null;
    }
325
```

```
  /**
327  * Retorna o último elemento da lista ou <code>null</code> se a lista
       estiver vazia.
     *
329  * @return O último elemento da lista ou <code>null</code> se a lista
       estiver vazia.
     */
331  public E getUltimo()
     {
333    return !isEmpty() ? ultimo.getElemento() : null;
     }
335
     /**
337  * Retorna a referência ao {@link No} de um dado elemento, ou <code>null</
       code> se o elemento
     * não pertence a lista.
339  *
     * @param elemento
341  *            Elemento a ser retornada a referência do {@link No} que o
       encapsula.
     * @return A referência ao {@link No} de um dado elemento, ou <code>null</
       code> se o elemento
343  *         não pertence a lista.
     */
345  No<E> pesquisar(E elemento)
     {
347    if (!this.isEmpty())
       {
349      for (No<E> no = this.primeiro; no != null; no = no.getProximo())
         {
351        if (no.getElemento().equals(elemento))
           {
353          return no;
           }
355      }
       }
357    return null;
     }
359
     No<E> no(int posicao)
361  {
       if (!this.isEmpty())
363      return null;

365    No<E> no = this.primeiro;

367    for (int i = 0; i < posicao; i++)
       {
369      no = no.getProximo();
       }
371    return no;
     }
373
     No<E> primeiro()
```

```
375     {
          return this.primeiro;
377     }

379     No<E> ultimo()
        {
381       return this.ultimo;
        }
383
        @Override
385     public boolean equals(Object obj)
        {
387       if (this == obj)
            return true;
389
          if (!(obj instanceof ListaEncadeada<?>))
391         return false;

393       return this.iguais(this, (ListaEncadeada<E>) obj);
        }
395
        @Override
397     public int hashCode()
        {
399       return this.primeiro() != null ? this.primeiro().hashCode() * 17 : super.
              hashCode();
        }
401 }
```

Listing 1: Implementação de Lista Encadeada Simples em Java