

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 .

```
1  /**
   * Adiciona um dado elemento (sucessor) após um determinado elemento
   * existente na lista.
3  *
   * @param elemento
5  *         Elemento existente na lista e deve ser ter como sucessor
   *         um dado elemento.
   * @param sucessor
7  *         Elemento a ser colocado após um dado elemento existente
   *         na lista.
   */
9  public void adicionarApos(E elemento, E sucessor)
   {
11     No<E> n1 = this.pesquisar(elemento);
       if (n1 != null)
13     {
           n1.setProximo(new No<E>(sucessor, n1.getProximo()));
15     }
   }
```

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.

```
2  /**
   * Remove o elemento que está em uma dada posição na lista.
   * @param posicao Posição do elemento a ser removido.
4  * @return O elemento que estava na posição removida ou <code>null</code>
   *         quando a posição informada não existe.
   */
6  public E remover(int posicao)
   {
8     No<E> no = this.no(posicao);

10     if (no != null)
       {
12         this.remover(no.getElemento());
           return no.getElemento();
14     }
       return null;
16 }
```

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.

```
2  /**
3   * Substitui todas as ocorrências do elemento E1 pelo elemento E2 e
4   * retorna a quantidade de substituições.
5   * @param e1 Elemento a ser substituído.
6   * @param e2 Elemento substituto.
7   * @return A quantidade de substituições.
8   */
9  public int replace(E e1, E e2)
10 {
11     int cont = 0;
12     if (!this.isEmpty())
13     {
14         No<E> anterior = null;
15         No<E> atual = this.primeiro;
16
17         for (int i = 0; i < this.tamanho; i++)
18         {
19             if (atual.equals(new No<E>(e2)))
20             {
21                 if (anterior != null)
22                 {
23                     anterior.setProximo(new No<E>(e2, atual.getProximo()));
24                     cont++;
25                 }
26             }
27             anterior = atual;
28             atual = atual.getProximo();
29         }
30     }
31     return cont;
32 }
```

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.

```
2  /**
   * Concatena duas listas a partir de uma dada posição.
   *
4  * @param l1 Lista a ser concatenada a partir.
   * @param l2 Lista a ser concatenada a partir de uma dada posição da
   *           lista l1.
6  * @param posicao Posição a partir da qual a lista l2 deve ser
   *           concatenada com a lista l2.
   */
8  public void concatenar(ListaEncadeada<E> l1, ListaEncadeada<E> l2, int
   posicao)
   {
10     if (!l1.isEmpty())
       {
12         No<E> no = l1.primeiro();
         for (int i = 0; i < posicao; i++)
14             {
                 no = no.proximo();
16             }

18         l2.primeiro().setProximo(no.getProximo());
         no.setProximo(l2.primeiro());
20     }
   }
```

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.

```
1  /**
   * Verifica se duas listas são iguais.
3  * @param l1 Lista l1 a ser comparada com a lista l2.
   * @param l2 Lista l2 a ser comparada com a lista l1.
5  * @return <code>true</code> se as listas forem iguais ou <code>false
   *           </code> caso contrário.
   */
7  public boolean iguais(ListaEncadeada<E> l1, ListaEncadeada<E> l2)
   {
9     if (l1 == null && l2 == null)
         return true;

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

```

15     if (!l1.isEmpty())
16     {
17         No<E> no1 = l1.primeiro;
18         No<E> no2 = l2.primeiro;
19
20         while (no1 != null)
21         {
22             if (!no1.equals(no2))
23                 return false;
24
25             no1 = no1.getProximo();
26             no2 = no2.getProximo();
27         }
28     }
29     return true;
30 }

```

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 cálculo 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;
2
3 /**
4  * Implementação de Lista Encadeada Simples com alocação dinâmica.
5  *
6  * @author alessandro
7  * @param <E>
8  *         Tipo do valor a ser armazenado na lista.
9  */
10 public class ListaEncadeada<E>
11 {
12     /**
13      * Referência ao primeiro elemento da {@link ListaEncadeada}
14      */
15     private No<E> primeiro;
16
17     /**
18      * Referência ao último elemento da {@link ListaEncadeada}
19      */
20     private No<E> ultimo;
21
22     /**

```

```

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;
77 }

@Override
79 public int hashCode()
81 {
    return this.getElemento() != null ? this.getElemento().hashCode() :
        super.hashCode();
83 }
}

85
/**
87 * Adiciona um dado elemento no final da lista.
88 *
89 * @param elemento
90 *     Elemento a ser adicionado a lista.
91 */
public void adicionar(E elemento)
93 {
    if (this.isEmpty())
95     {
        this.ultimo = this.primeiro = new No<E>(elemento, this.primeiro);
97     } else
    {
        No<E> no = new No<E>(elemento);
        this.ultimo.setProximo(no);
101     this.ultimo = no;
    }
103     this.tamanho++;
}

105
/**
107 * Adiciona um dado elemento (sucesso) após um determinado elemento
    existente na lista.
    *
109 * @param elemento
    *     Elemento existente na lista e deve ser ter como sucessor um
        dado elemento.
111 * @param sucessor
    *     Elemento a ser colocado após um dado elemento existente na
        lista.
113 */
public void adicionarApos(E elemento, E sucessor)
115 {
    No<E> n1 = this.pesquisar(elemento);
117     if (n1 != null)
    {
        n1.setProximo(new No<E>(sucessor, n1.getProximo()));
119     }
121 }

123
/**
    * Concatena a lista l2 a essa lista a partir da posição informada.
125 *

```

```

127 * @param l2
128 *      Lista a ser concatenada com essa lista a partir de uma dada
129 *      posição.
130 * @param posicao
131 *      Posição a partir da qual a lista l2 deve ser concatenada.
132 */
133 public void concatenar(ListaEncadeada<E> l2, int posicao)
134 {
135     this.concatenar(this, l2, posicao);
136 }
137
138 /**
139 * Concatena duas listas a partir de uma dada posição.
140 *
141 * @param l1
142 *      Lista a ser concatenada a partir.
143 * @param l2
144 *      Lista a ser concatenada a partir de uma dada posição da lista
145 *      l1.
146 * @param posicao
147 *      posição a partir da qual a lista l2 deve ser concatenada com a
148 *      lista l2.
149 */
150 public void concatenar(ListaEncadeada<E> l1, ListaEncadeada<E> l2, int
151     posicao)
152 {
153     if (!l1.isEmpty())
154     {
155         No<E> no = l1.primeiro();
156         for (int i = 0; i < posicao; i++)
157         {
158             no = no.proximo();
159         }
160
161         l2.primeiro().setProximo(no.getProximo());
162         no.setProximo(l2.primeiro());
163     }
164 }
165
166 /**
167 * Remove um dado elemento da lista se o elemento pertencer a lista.
168 *
169 * @param elemento
170 *      Elemento que deve ser removido.
171 */
172 public void remover(E elemento)
173 {
174     if (!this.isEmpty())
175     {
176         No<E> no_anterior_ao_que_sera_removido = null;
177         No<E> no_a_remover = this.primeiro();
178         while (no_a_remover != null && !no_a_remover.getElemento().equals(
179             elemento))
180         {
181             no_anterior_ao_que_sera_removido = no_a_remover;
182             no_a_remover = no_a_remover.proximo();
183         }
184         if (no_a_remover != null)
185         {
186             if (no_anterior_ao_que_sera_removido != null)
187                 no_anterior_ao_que_sera_removido.setProximo(no_a_remover.proximo());
188             else
189                 primeiro = no_a_remover.proximo();
190             no_a_remover.setProximo(null);
191         }
192     }
193 }

```

```

175     {
176         no_anterior_ao_que_sera_removido = no_a_remove;
177         no_a_remove = no_a_remove.getProximo();
178     }
179
180     if (no_anterior_ao_que_sera_removido != null)
181     {
182         no_anterior_ao_que_sera_removido.setProximo(no_a_remove.getProximo());
183         ;
184         this.tamanho--;
185     }
186 }
187
188 /**
189  * Remove o elemento que está em uma dada posição na lista.
190  *
191  * @param posicao
192  *         Posição do elemento a ser removido.
193  * @return O elemento que estava na posição removida ou <code>null</code>
194  *         quando a posição
195  *         informada não existe.
196 */
197 public E remover(int posicao)
198 {
199     No<E> no = this.no(posicao);
200
201     if (no != null)
202     {
203         this.remover(no.getElemento());
204         return no.getElemento();
205     }
206     return null;
207 }
208
209 /**
210  * Retorna <code>true</code> se um dado elemento pertence a {@link
211  *     ListaEncadeada} ou
212  *     <code>false</code> caso contrário.
213  *
214  * @param elemento
215  *         Elemento a ser verificado se ele pertence a {@link
216  *         ListaEncadeada}
217  * @return <code>true</code> se o elemento pertence a {@link ListaEncadeada}
218  *         ou
219  *         <code>false</code> caso contrário.
220 */
221 public boolean contem(E elemento)
222 {
223     return this.pesquisar(elemento) != null;
224 }
225
226 /**
227  * Verifica se duas listas são iguais.

```



```

225 *
226 * @param l1
227 *         Lista l1 a ser comparada com a lista l2.
228 * @param l2
229 *         Lista l2 a ser comparada com a lista l1.
230 * @return <code>true</code> se as listas forem iguais ou <code>false</code>
231 *         caso contrário.
232 */
233 public boolean iguais(ListaEncadeada<E> l1, ListaEncadeada<E> l2)
234 {
235     if (l1 == null && l2 == null)
236         return true;
237
238     if (l1.getTamanho() != l2.getTamanho())
239         return false;
240
241     if (!l1.isEmpty())
242     {
243         No<E> no1 = l1.primeiro;
244         No<E> no2 = l2.primeiro;
245
246         while (no1 != null)
247         {
248             if (!no1.equals(no2))
249                 return false;
250
251             no1 = no1.getProximo();
252             no2 = no2.getProximo();
253         }
254     }
255     return true;
256 }
257
258 public boolean iguais(ListaEncadeada<E> l2)
259 {
260     return this.iguais(this, l2);
261 }
262
263 /**
264 * Substitui todas as ocorrências do elemento E1 pelo elemento E2 e retorna
265 * a quantidade de
266 * substituições.
267 *
268 * @param e1
269 *         Elemento a ser substituído.
270 * @param e2
271 *         Elemento substituto.
272 * @return A quantidade de substituições.
273 */
274 public int replace(E e1, E e2)
275 {
276     int cont = 0;
277     if (!this.isEmpty())
278     {

```

```

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

279     for (int i = 0; i < this.tamanho; i++)
280     {
281         if (atual.equals(new No<E>(e2)))
282         {
283             if (anterior != null)
284             {
285                 anterior.setProximo(new No<E>(e2, atual.getProximo()));
286                 cont++;
287             }
288         }
289         anterior = atual;
290         atual = atual.getProximo();
291     }
292 }
293 return cont;
294 }

295 /**
296  * Retorna <code>>true</code> se a lista estiver vazia ou <code>>false</code>
297  * caso contrário.
298  *
299  * @return <code>>true</code> se a lista estiver vazia ou <code>>false</code>
300  * caso contrário.
301  */
302 public boolean isEmpty()
303 {
304     return this.tamanho < 0;
305 }

306 /**
307  * Retorna o tamanho da lista.
308  *
309  * @return O tamanho da lista.
310  */
311 public int getTamanho()
312 {
313     return isEmpty() ? 0 : tamanho + 1;
314 }

315 /**
316  * Retorna o primeiro elemento da lista ou <code>null</code> se a lista
317  * estiver vazia.
318  *
319  * @return O primeiro elemento da lista ou <code>null</code> se a lista
320  * estiver vazia.
321  */
322 public E getPrimeiro()
323 {
324     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
336  /**
337  * Retorna a referência ao {@link No} de um dado elemento, ou <code>null</
    code> se o elemento
338  * 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 {
377     return this.primeiro;
379 }
379 No<E> ultimo()
381 {
383     return this.ultimo;
385 }
383 @Override
385 public boolean equals(Object obj)
387 {
389     if (this == obj)
391         return true;
393
395     if (!(obj instanceof ListaEncadeada<?>))
397         return false;
399
401     return this.iguais(this, (ListaEncadeada<E>) obj);
403 }
405 @Override
407 public int hashCode()
409 {
411     return this.primeiro() != null ? this.primeiro().hashCode() * 17 : super.
        hashCode();
413 }
415 }

```

Listing 1: Implementação de Lista Encadeada Simples em Java