Listas duplamente ligadas

Na aula passada aprendemos sobre *Listas ligadas*, cuja ideia era a de que uma célula estava ligada à sua próxima em um *array*. Ela nos facilitou em relação à implementação e velocidade de execução.

Conseguimos melhorar ainda mais a Lista ligada para terminarmos com nossa implementação da aula passada. Nos faltou trabalhar em cima do método que remove um elemento que esteja em qualquer posição.

Nesta aula veremos as ***Listas duplamente ligadas***, cujos elementos não apenas apontam para seu próximo, mas também para seu anterior.

Então, voltando à nossa Classe `Celula`, vamos criar um novo parâmetro com seu *getter* e *setter*:

```
private Celula anterior;

...

public Celula getAnterior() {
    return anterior;
}

public void setAnterior(Celula anterior) {
    this.anterior = anterior;
}
```

E vamos criar um Construtor que irá nos ajudar ao implementarmos o primeiro método:

```
public Celula(Object elemento) {
    this(elemento, null);
}
```

Vamos, a partir de agora, repensar o nosso código implementado na aula anterior para ele se adequar aos novos parâmetros.

Método *adicionaNoComeco*

Na Classe "ListaLigada", o primeiro método que implementamos foi o "adicionaNoComeco". Vamos reescrevê-lo:

```
public void adicionaNoComeco(Object elemento) {
    if(this.totalDeElementos == 0) {
        Celula nova = new Celula(elemento);
        this.primeira = nova;
        this.ultima = nova;
    } else {
        Celula nova = new Celula(this.primeira, elemento);
        this.primeira.setAnterior(nova);
        this.primeira = nova;
    }
    this.totalDeElementos++;
}
```

Vamos entender este código:

- Se a lista está vazia, criamos uma célula e o próximo dela é *null*. Logicamente o anterior também. Isto já havíamos feito anteriormente.
- Criamos uma nova célula cuja próxima é a primeira. E a anterior a esta é a nova. E a primeira é a nova.

Método *adiciona* (no fim)

```
public void adiciona(Object elemento) {
    if(this.totalDeElementos == 0) {
        adicionaNoComeco(elemento);
    } else {
        Celula nova = new Celula(elemento);
        this.ultima.setProxima(nova);
        nova.setAnterior(this.ultima);
        this.ultima = nova;
        this.totalDeElementos++;
}
```

Muito parecido com o método anteriormente implementado. A única diferença é que setamos para a célula anterior.

- Criamos uma nova célula.
- A última foi apontou a próxima para essa nova célula.
- A nova aponta a anterior para a última atual.
- A última atual agora é a nova célula.

Método *adiciona* (numa posição qualquer)

```
public void adiciona(int posicao, Object elemento) {

    if(posicao == 0) {
        adicionaNoComeco(elemento);
    } else if (posicao == this.totalDeElementos) {
        this.adiciona(elemento);
    } else {
        Celula anterior = pegaCelula(posicao - 1);
        Celula proxima = anterior.getProxima();

        Celula nova = new Celula(anterior.getProxima(), elemento);
        nova.setAnterior(anterior);
        anterior.setProxima(nova);
        proxima.setAnterior(nova);
        this.totalDeElementos++;
}
```

Método *remove* (do fim)

Na aula passada não chegamos a implementar esse método, pois ainda não tínhamos o conceito de *lista duplamente ligada*.

Se o *array* possui apenas um elemento, chamamos o método "removeDoComeco":

```
public void removeDoFim() {
    if(this.totalDeElementos == 1) {
        this.removeDoComeco();
    }
```

Para removermos o elemento do fim, precisamos da penúltima célula, que está ligada a ele:

```
public void removeDoFim() {
    if(this.totalDeElementos == 1) {
        this.removeDoComeco();
    } else {
        Celula penultima = this.ultima.getAnterior();
        penultima.setProxima(null);
        this.ultima = penultima;
        this.totalDeElementos--;
    }
}
```

Vamos testar o método. Antes, a lista possuía `mauricio`, `cecilia`, `paulo`. Chamando a função:

```
lista.removeDoFim();
System.out.println(lista);
```

O retorno será:

```
[mauricio, cecilia]
```

Método *remove* (de qualquer posição)

Se o elemento estiver na primeira ou na última posição basta chamar os métodos já implementados:

```
public void remove(int posicao) {
    if(posicao == 0) {
        this.removeDoComeco();
    } else if (posicao == this.totalDeElementos - 1) {
        this.removeDoFim();
    }
}
```

Mas agora precisamos pensar como remover o elemento do meio. Vamos navegar e dar os nomes aos elementos e *setar* seus anteriores e próximos:

```
public void remove(int posicao) {
    if(posicao == 0) {
        this.removeDoComeco();
    } else if (posicao == this.totalDeElementos - 1) {
        this.removeDoFim();
    } else {
        Celula anterior = this.pegaCelula(posicao - 1);
        Celula atual = anterior.getProxima();
        Celula proxima = atual.getProxima();

        anterior.setProxima(proxima);
        proxima.setAnterior(anterior);

        this.totalDeElementos--;
    }
}
```

Vamos testar este método. Primeiramente acrescentemos mais alguns nomes na lista para termos algo assim:

```
[mauricio, cecilia, jose, joao]
```

Agora fazemos, por exemplo,

```
lista.remove(2);
System.out.println(lista);
```

O que nos retorna

```
[mauricio, cecilia, joao]
```

De fato, o elemento na posição 2, José, foi removido da lista.

Método *contem*

Este método não chegamos a implementar na aula passada. Ele será parecido com o do Vetor. Vamos utilizar o `while`, que é uma outra abordagem de laço.

```
public boolean contem(Object elemento) {
    Celula atual = this.primeira;

    while(atual != null) {
        if(atual.getElemento().equals(elemento)) {
            return true;
        }
        atual = atual.getProxima();
    }
    return false;
}
```

O método varrerá todo o *array* até encontrar (*true*), ou não (*false*), o elemento citado.

Vamos testar:

```
System.out.println(lista.contem("mauricio");
System.out.println(lista.contem("danilo");
```

O programa retornará

```
true
false
```

De fato, o Mauricio está na lista e o Danilo não.