02

Explicação

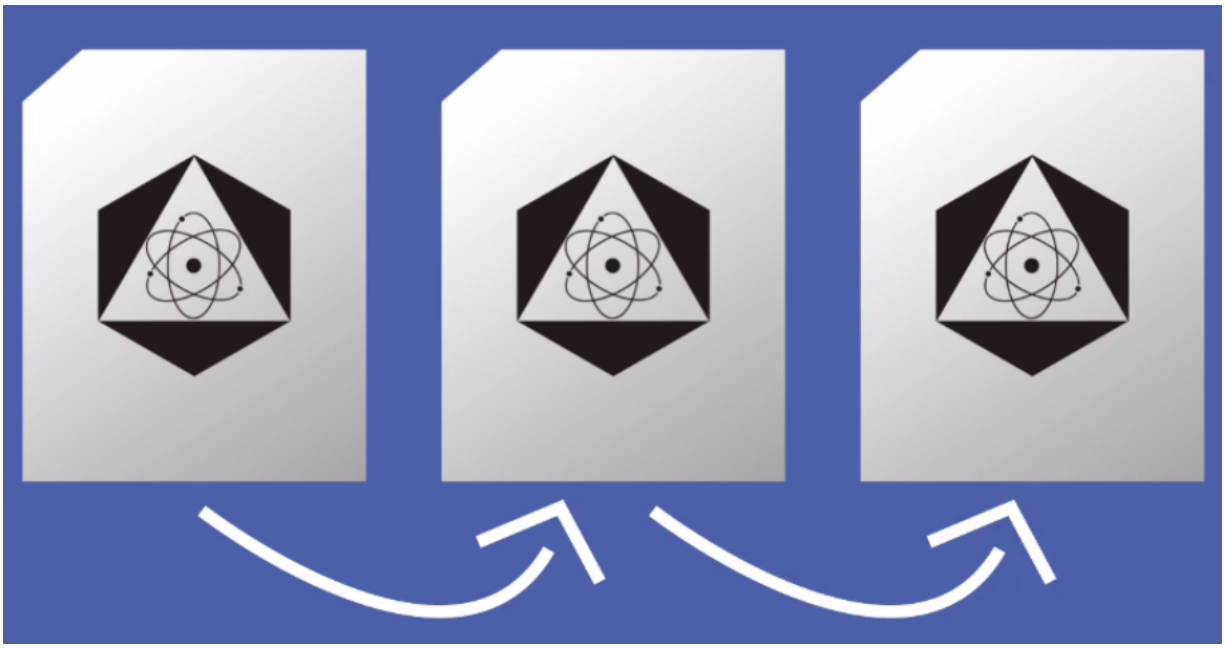
Listas ligadas

Na aula passada aprendemos sobre Vetores, e vimos que eles são boas estruturas de dados para diversos casos:

- adicionar elementos no fim do vetor;
- pegar um elemento aleatório;
- remover elementos, etc.

Porém, outros métodos já não eram tão simples, como inserir um elemento no meio do vetor, esta é uma atividade computacionalmente cara, com processo de execução lento. Dado que vimos o Vetor e observamos seus prós e contras, aprenderemos aqui uma outra lista. Com ela tentaremos melhorar o código para que essa adição de elementos no meio do *array* seja um processo mais rápido.

A essa lista nós damos o nome de ***Lista Ligada***. A diferença dela para o Vetor é que neste os elementos estão um do lado do outro na memória, enquanto que na *Lista Ligada* eles estão em lugares diferentes, porém um aponta para o outro indicando o próximo.



Então é dessa forma que iremos desenhar a estrutura, na qual um elemento também conhecerá o endereço do próximo. Criemos uma Classe "Celula" que posuirá um objeto e seu seguinte (do tipo "Celula"):

```
package ed.listaligada;

public class Celula {

    private Object elemento;
    private Celula proximo;

}
```

Para nos facilitar, vamos criar um Construtor e *getters* (para o elemento) e *setters* (para o elemento e para a Celula):

```
public Celula(Object elemento, Celula proximo) {
    this.elemento = elemento;
    this.proximo = proximo;
}

public Celula getProximo() {
    return proximo;
}

public void setProximo(Celula proximo) {
    this.proximo = proximo;
}

public Object getElemento() {
    return elemento;
}
```

Agora vamos criar a Classe "ListaLigada" e definir suas funções:

```
package ed.listaligada;

public class ListaLigada {

    public void adicionaNoComeco(Object elemento) {}

    public void adiciona(Object elemento) {}

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

    public Object pega(int posicao) { return null; }

    public void remove(int posicao) {}

    public int tamanho() { return 0; }

    public boolean contem(Object o) { return false;}
}
```

Ou seja, as mesmas que fizemos para o Vetor. Vamos começar implementando o método "adicionaNoComeco".

Método *adicionaNoComeco*

Vamos começar imaginando que já temos uma lista com células apontando uma para outra. Para uma nova Célula entrar no começo do *array* ela deve apontar para sua próxima, ou seja, a primeira do *array* atual. Então devemos ter um atributo chamado "primeira". Como a lista começa vazia, essa célula aponta para *null*:

```
public class ListaLigada {

    private Celula primeira = null;
```

```
    public void adicionaNoComeco(Object elemento) {
        Celula nova = new Celula(elemento, primeira);

    }
```

Na lista vazia, ao adicionarmos uma célula na primeira posição do *array*, ela deverá apontar para *null*, ao acrescentarmos uma próxima, também no começo, esta apontará para aquela anterior; e soma-se 1 ao total de elementos

```
public class ListaLigada {

    private Celula primeira = null;
    private int totalDeElementos = 0;

    public void adicionaNoComeco(Object elemento) {
        Celula nova = new Celula(elemento, primeira);
        this.primeira = nova;

        this.totalDeElementos++;
    }
```

Para testarmos, vamos criar a Classe "TestaListaLigada" com o método *main* e implementar para imprimir depois de cada inserção de elemento:

```
package ed.listaligada

public class TestaListaLigada {

    public static void main(String[] args) {
        ListaLigada lista = new ListaLigada();

        System.out.println(lista);
        lista.adicionaNoComeco("mauricio");
        System.out.println(lista);
        lista.adicionaNoComeco("paulo");
        System.out.println(lista);
        lista.adicionaNoComeco("guilherme");
        System.out.println(lista);
    }
}
```

Se deixarmos desse jeito, o retorno não será amigável e não entenderemos nada. Vamos criar um *toString* amigável na Classe "ListaLigada":

```
@Override
public String toString () {

    if(this.totalDeElementos == 0) {
        return "[]";
    }

    Celula atual = primeira;

    StringBuilder builder = new StringBuilder("[");
```

```
        for(int i = 0; i < totalDeElementos; i++) {
            builder.append(atual.getElemento());
            builder.append(",");

            atual = atual.getProximo();
        }

        builder.append("]");

        return builder.toString();
    }
```

Ao rodarmos, retorna

```
[]
[mauricio,]
[paulo,mauricio,]
[guilherme,paulo,mauricio,]
```

Método *adiciona* (no *fim* da lista)

Para Listas Ligadas, este método é um pouco mais complexo. O que nos diz se um elemento é o último do *array* é se ele apontar para um *null*. Para isso é necessário varrer toda a lista. Vamos resolver o problema criando uma seta para o último elemento (da mesma forma que fizemos para o primeiro):

```
    private Celula primeira = null;

    private Celula ultima = null;
```

Com essa mudança teremos que arrumar algumas coisas no método "adicionaNoComeco". Se a lista está vazia, o primeiro elemento também será o último:

```
        public void adicionaNoComeco(Object elemento) {
            Celula nova = new Celula(elemento, primeira);
            this.primeira = nova;

            if(this.totalDeElementos == 0) {
                this.ultima = this.primeira;
            }

            this.totalDeElementos++;
        }
```

Voltemos ao desafio de inserir no final. Criamos uma nova celula cujo próximo elemento é *null*, afinal ela está sendo adicionada no final do *array*. Precisamos fazer com que a última atual aponte para essa nova.

```
    public void adiciona(Object elemento) {

        Celula nova = new Celula(elemento, null);
```

```
        this.ultima.setProximo(nova);
        this.ultima = nova;
        this.totalDeElementos++;
    }
```

Mas precisamos cuidar do caso particular em que a lista está vazia e faremos isso nos utilizando do outro método já implementado:

```
    public void adiciona(Object elemento) {

        if(this.totalDeElementos == 0) {
            adicionaNoComeco(elemento);
        } else {
            Celula nova = new Celula(elemento, null);
            this.ultima.setProximo(nova);
            this.ultima = nova;
            this.totalDeElementos++;
        }
    }
```

Vamos testar:

```
    lista.adiciona("marcelo");
    System.out.println(lista);
```

O que retorna:

```
    [guilherme,paulo,mauricio,marcelo,]
```

Método *adiciona* (no *meio* da lista)

Para implementarmos esse método vamos criar outros dois para ajudar. Um irá indicar quando a posição existir, estiver ocupada:

```
    private boolean posicaoOcupada(int posicao) {
        return posicao >= 0 && posicao < this.totalDeElementos;
    }
```

E o outro irá apontar para a célula na qual queremos inserir o elemento:

```
    private Celula pegaCelula(int posicao) {

        if(!posicaoOcupada(posicao)) {
            throw new IllegalArgumentException("posicao inexistente");
        }

        Celula atual = primeira;

        for(in i = 0; i < posicao; i++) {
```

```
            atual = atual.getProximo();
        }
        return atual;
    }
```

Imaginemos agora, mais uma vez, que já possuímos uma lista onde um elemento aponta para o outro. O elemento da esquerda deve apontar para o novo, e este para o da direita. Então, em código, fazemos:

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

        Celula anterior = this.pegaCelula(posicao - 1);
        Celula nova = new Celula(elemento, anterior.getProximo();
    }
```

Dessa forma pegamos a Célula da esquerda (anterior) e a nova no lugar da próxima (anterior.getProximo). Por fim, basta fazer com que a anterior seja a nova e somar 1 no total de elementos:

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

        Celula anterior = this.pegaCelula(posicao - 1);
        Celula nova = new Celula(elemento, anterior.getProximo();
        anterior.setProximo(nova);
        this.totalDeElementos++;
    }
```

Ainda falta implementar o método para quando a lista estiver vazia ou quando a posição "do meio" seja, na realidade, seja a última:

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

        if(posicao == 0) {
            adicionaNoComeco(elemento);
        } else if (posicao == this.totalDeElementos) {
            adiciona(elemento);
        } else {
            Celula anterior = this.pegaCelula(posicao - 1);
            Celula nova = new Celula(elemento, anterior.getProximo();
            anterior.setProximo(nova);
            this.totalDeElementos++;
    }
```

Vamos testar, fazendo no *main*

```
    lista.adiciona(2, "gabriel");
    System.out.println(lista);
```

O que retorna

```
    [guilherme,paulo,gabriel,mauricio,marcelo,]
```

Métodos *pega* e *tamanho*

Os dois métodos são bem simples de serem implementados.

pega

Para o "pega":

```
public Object pega(int posicao) {
    return this.pegaCelula(posicao).getElemento();
}
```

No *main*:

```
Object x = lista.pega(2);
System.out.println(x);
```

Retorna:

```
gabriel
```

tamanho

Para o "tamanho":

```
public int tamanho() {
    return this.totalDeElementos;
}
```

No *main*:

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

O que retorna

```
5
```

Método *remove*

Antes de implementarmos o método "remove", vamos fazer o "removeDoComeco":

```
public void removeDoComeco() {
    if(this.totalDeElementos == 0) {
        throw new IllegalArgumentException("lista vazia");
    }
```

```
        this.primeira = this.primeira.getProximo();
        this.totalDeElementos--;

        if(this.totalDeElementos == 0) {
            this.ultima = null;
        }
    }
```

Testando:

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

O que retorna:

```
[paulo,gabriel,mauricio,marcelo]
```

De fato, o elemento na primeira posição (guilherme) foi removido.

Para removermos um elemento que está em qualquer posição será muito mais trabalhoso. Veremos tal implementação no próximo capítulo, no qual abordaremos as *Listas **duplamente** ligadas*.