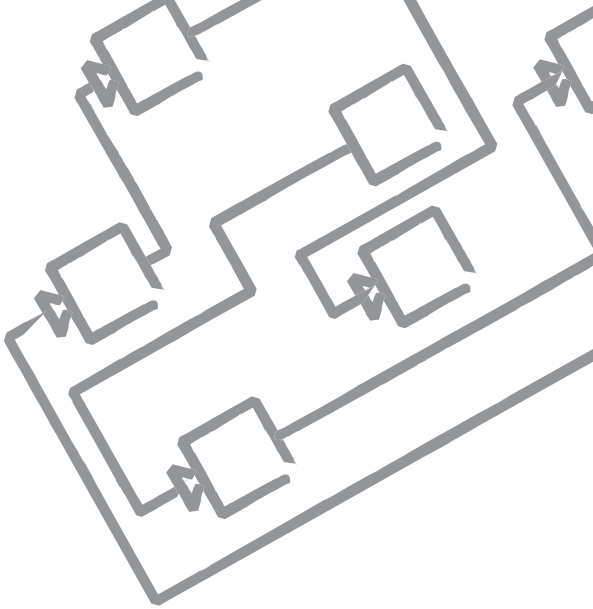
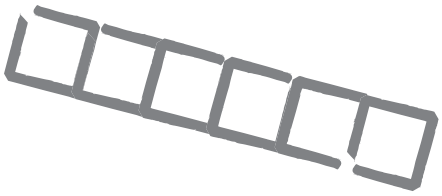
Caelum

Ensino e Soluções em Java





CS-14

Algoritmos e Estruturas de Dados em Java

www.caelum.com.br

Índice

Versão: 9.7.19

CAPÍTULO 1

Prefácio

Este material foi escrito por Paulo Eduardo Azevedo Silveira e Rafael Antonio Cosentino para ser utilizado no curso de verão **Introdução a Estrutura de dados e Algoritmos em Java**, do Instituto de Matemática e Estatística da Universidade de São Paulo.

Paulo Silveira é bacharel e mestre em ciência da computação pelo Instituto de Matemática e Estatística da Universidade de São Paulo (IME-USP). É um dos fundadores do Grupo de Usuários Java (http://www.guj.com.br) e trabalha atualmente na Caelum (http://www.caelum.com.br).

Rafael Cosentino é bacharel em ciência da computação pelo Instituto de Matemática e Estatística da Universidade de São Paulo (IME-USP), é mestrando em Ciência da Computação também pelo IME-USP e trabalha atualmente na Caelum.

As experiências como alunos e professores mostrou aos autores desta apostila os principais pontos de dificuldades no aprendizado. Isso motivou uma abordagem um pouco diferente para esta apostila em relação ao que é comum na maioria dos livros sobre o assunto

A cada capítulo, apresentaremos diversos problemas que servirão de motivação para os principais conceitos de estrutura de dados. Os exercícios complementarão o apredizado pois ajudam a fixar os conceitos vistos na apostila e estimulam o aluno para conceitos avançados que fogem do escopo deste material.

Além disso, no fim de cada capítulo, apresentaremos as bibliotecas do Java que modelam as estruturas de dados apresentadas nesta apostila.

CAPÍTULO 2

Introdução

“As três coisas mais difíceis no mundo: guardar segredo, perdoar uma injúria e aproveitar o tempo”
– Benjamin Franklin

2.1 - Introdução

Considere o problema de descobrir a altura da pessoa mais alta de um grupo de pessoas. Suponha que estas pessoas estão em seqüência, como em uma fila de banco, e que esta fila não está vazia.

Vamos elaborar uma estratégia para resolver este problema. Uma solução bem simples seria fazer o seguinte:

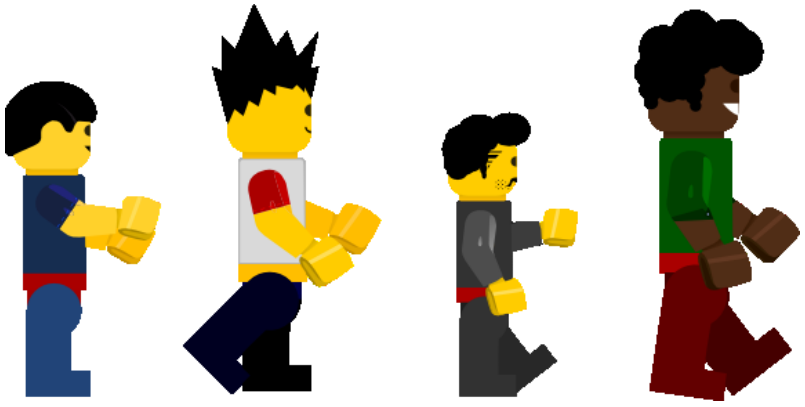


Fig.: Fila de pessoas

1) Pegue a altura da primeira pessoa. A única **informação** que você tem é que esta altura é a máxima até o momento. Então, “guarde” em algum lugar esta informação.
2) Percorra cada uma das próximas pessoas e faça o seguinte:
    1) Pegue a altura da pessoa, esta é a altura atual.
    2) Compare a altura atual com a máxima até o momento. Esta comparação pode resultar em três possibilidades: a altura atual é menor, é igual ou é maior.
    3) Se a altura atual for maior, então faça o valor da altura máxima ser igual a atual.

Será que este procedimento é bom? Nesse caso temos de percorrer todas as pessoas da fila até ter certeza que encontramos a maior pessoa, pois a única invariante que temos, a cada passo do procedimento, é que até aquele momento todas as pessoas percorridas tem tamanho igual ou menor a um determinado número.

Existe solução melhor? E se tivessemos as pessoas organizadas de alguma maneira especial, chegariamos a solução em um menor tempo? E se houveram empates, quais outras pessoas são tão altas quanto essa nossa resposta?

2.2 - Algoritmo e Implementação

Um **algoritmo** é uma seqüência de passos que resolve algum problema ou alcança algum objetivo, como a seqüência de passos para resolver o problema de descobrir a máxima altura. É importante salientar que um algoritmo simplesmente diz o que **deve** ser feito.

Para resolver de fato um problema, devemos definir **como** executar os passos do algoritmo. Por exemplo, para o problema anterior de achar a máxima altura, deveríamos definir como "pegar" as informações sobre as alturas da pessoas (perguntar para a própria pessoa, medir a altura usando uma fita métrica ou obter a altura de algum cadastro que a pessoa tenha feito) e como manter as informações sobre as alturas (anotar em um papel ou guardar em uma varíavel no computador).

A definicão de como os passos de um algoritmo serão executados é uma implementação do algoritmo. Resumindo, algoritmo é o que deve ser feito e implementação é o como deve ser feito.

Estamos interessados em desenvolver algoritmos computacionais. Para isso, utilizaremos um modelo de programação. Um modelo de programação fornece idéias e conceitos para nos ajudar a criar algoritmos. Neste curso, será utilizado o paradigma de programação orientado a obejetos (**OO**).

Os nossos algoritmos serão executados por um computador. Então, devemos implementá-lo através de programas de computador. Um programa é a definição de como os passos de um algoritmo serão executados no computador.

Os programas são escritos em alguma linguagem de programação. Uma linguagem de programação é a maneira de "conversarmos" com um computador. A linguagem que utilizaremos aqui é a **Java**. Esta linguagem é voltador para o paradigma de programação orientado a objetos.

2.3 - Estrutura de Dados

Hoje em dia, a grande maioria das pessoas utilizam a agenda do celular para armazenar seus contatos. As tarefas de uma agenda de contatos são basicamente duas:

1) Definir como as informações dos contatos serão armazenadas. Uma informação armazenada em algum lugar (pedaço de papel, livro, computador, etc) é um **dado**.

2) Disponibilizar operações para criar, recuperar, ordernar, atualizar e remover contatos. Além de operações para informar o estado da agenda (a quantidade de contatos existentes na agenda ou a quantidade de espaço disponível para novos contatos).

A primeira tarefa é crucial para o desempenho. Se a agenda armazena as informações de uma forma desorganizada então será muito mais complicado manipular os dados de forma eficiente. A escolha de como guardar as informações deve levar em cosideração as operações que serão disponibilizadas pela agenda. Por exemplo, seria interessante manter os contatos em ordem alfabética para facilitar a busca.

Mas, apesar da importância de como os contatos são armazenados, a organização interna da agenda não precisa e não deve ser exposta ao usuário. Afinal de contas, o que o usuário deseja é usar a agenda através das operações e que tudo seja feito o mais rápido possível.

A única coisa que precisa ser mostrada para o usuário são as operações que ele pode fazer na agenda (inserir, recuperar, atualizar, remover contato, saber quantos contatos estão na agenda, etc). Este conjunto de operações é a **interface** que o usuário tem com a agenda.

Cada celular pode implementar a sua agenda de contatos de uma forma totalmente diferente um do outro, na tentativa de obter mais performance, ser mais confiável ou gastar menos memória. Porém o conjunto básico de operações oferecidas pelas agendas é sempre o mesmo. Isso facilita a vida do usuário pois se ele tiver que trocar de celular não vai ter que aprender novamente como usar uma agenda de contatos.

Essa é a grande vantagem em se pensar em interface. Mantida a interface, podemos trocar uma agenda que não é tão eficiente ou que já não atende mais as nossas necessidades por outra mais eficiente ou adequada, sem problemas em ter de aprender a usar a nova agenda: troca-se a implementação, mas não mudamos a interface.

Uma agenda de celular pode ser vista como uma **estrutura de dados**. Uma estrutura de dados mantém os dados organizados seguindo alguma lógica e disponibiliza operações para o usuário manipular os dados.

É importante, quando programar, não misturar dado e estrutura de dados em uma coisa só. Um dado é uma informação armazenada e estrutura de dados é quem administra os dados. O ideal é que a estrutura de dados seja o mais independente possível dos dados que ela vai armazenar. Desta forma pode-se aproveitar a mesma estrutura de dados para diversos tipos de dados. Por exemplo, é melhor ter uma agenda genérica do que uma agenda de telefones. Uma agenda genérica pod ser utilizada para guardar telefones, ou emails, ou até mesmo guardar uma outra estrutura dentro dela: contatos, que seriam compostos por nome, telefone e email.

Algumas estruturas de dados são apenas agrupamentos de dados sem um objetivo de aplicar algum algoritmo ou tirar proveito de sua estrutura. Um exemplo seria a estrutura **Contato**. Algumas outras estruturas são mais espertas e trabalhosas, como a **Agenda**, assim como Listas Ligadas, Vetores, Tabelas de Espalhamento e outras que veremos no decorrer do texto. Estas estruturas, por sua característica mais complexa e de poder ser reutilizada em outros contextos, devem ser criadas da forma mais independente possível dos dados que estarão dentro dela. Em outras palavras, não devemos misturar a **Agenda** e o **Contato** de maneira rígida, para que com isso possamos criar outras Agendas, como por exemplo uma Agenda de **Fornecedor**.

2.4 - Sobre este texto

Este texto vai mostrar a você diversas estruturas de dados, suas vantagens e desavantagens, assim como suas implementações básicas e classes já existentes na bibilioteca padrão do Java.

Vamos usar recursos do Java como interfaces, generics, exceptions, pacotes e outros. É bom já ter um conhecimento razoável da linguagem e um pouco de orientação a objetos.

CAPÍTULO 3

Armazenamento Sequencial

"A morte do homem começa no instante em que ele desiste de aprender"
– Albino Teixeira

3.1 - Motivação

Todo mundo já experimentou sopa de letrinhas quando criança. Aliás, quando as crianças tomam sopa de letrinhas, elas ficam muito mais interessadas em formar as palavras do que em tomar a sopa.

O que chama mais a atenção é que nesta brincadeira de criança aparece um conceito bem interessante de estrutura de dados. Quando a sopa é servida, as letras estão todas espalhadas sem ordem alguma e sem nenhum significado. Quando você escolhe um grupo de letras e as coloca em seqüência formando uma palavra, este grupo de letras ganha um valor semântico, ou seja, ganha um significado.

O grupo de letrinhas que você escolhe para formar uma palavra pode conter letras repetidas sem problema nenhum. A única regra é que as letras postas em seqüência devem formar uma palavra existente em alguma língua.

Fig.: Sopa de Letrinhas

As músicas também são exemplos em que a definição de uma seqüência dos elementos agrega valor semântico. Uma música é composta de notas musicais. Quando estas notas estão "espalhadas", elas não siginificam muita coisa. Já quando colocadas em uma seqüência adequada, formam uma música que as pessoas podem apreciar. Além disso, uma mesma nota musical pode aparecer mais de uma vez em uma única música.

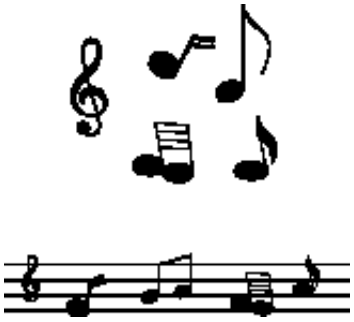


Fig.: Música

Outro exemplo em que a seqüência dos elementos é importante são as rotas de avião. Imagine o nome de várias cidades sem nehuma ordem. Desta maneira, estes nomes não significam muita coisa. Mas, se eles forem colocados em alguma seqüência então poderiam formar a rota de uma aeronave.

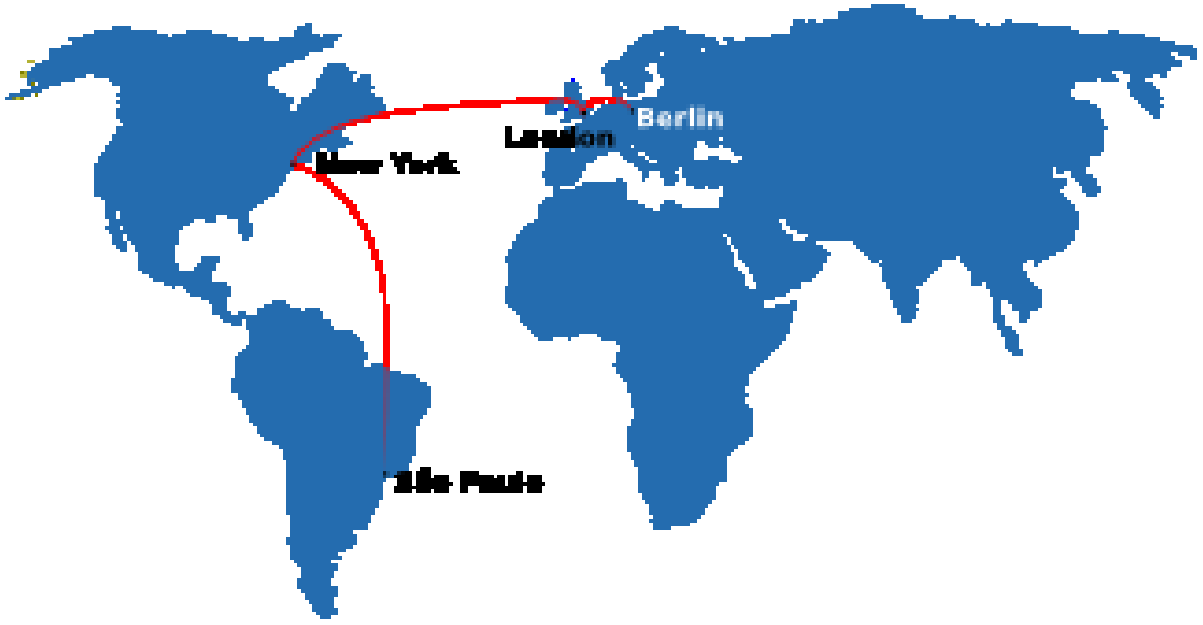


Fig.: Rota de avião

3.2 - O problema da listagem de alunos

Uma certa instituição de ensino está incentivando os seus alunos a participarem de eventos acadêmicos em troca de créditos para obter desconto nas mensalidades.

Para participar, o aluno deve comparecer em algum dos eventos cadastrados na instituição e depois escrever um relatório sobre o conteúdo apresentado no evento. Este relatório será avaliado por um professor e receberá uma pontuação de `0` a `100`.

A instituição quer manter uma listagem dos alunos que entregaram relatórios. Cada relatório entregue por um aluno corresponde a uma entrada na lista. Então, os alunos que entregarem mais de um relatório irão aparecer mais de uma vez na listagem.

Por exemplo, suponha que entregaram relatórios os alunos Rafael, Paulo e Ana. Rafael entregou apenas um relatório; Paulo entregou dois relatórios; e Ana entregou três relatórios. Uma possível listagem neste caso seria assim:

1) Rafael
2) Paulo
3) Ana
4) Paulo
5) Ana
6) Ana

A listagem também deve manter a ordem de pontuação obtidas pelos relatórios dos alunos.

Por exemplo, suponha que o Rafael teve pontuação máxima(100) no seu relatório; O Paulo teve pontuação 70 em um relatório e 50 no outro; Ana teve pontuação 60, 40 e 40 nos seus relatórios. Então, a listagem ficaria assim:

1) Rafael (100)
2) Paulo (70)
3) Ana (60)
4) Paulo (50)
5) Ana (40)
6) Ana (40)

Conforme os alunos forem entregando os relatórios, novas entradas serão adicionadas na listagem. Uma nova entrada pode ser inserida em qualquer posição. A posição é definida de acordo com a pontuação do relatório do aluno.

Por exemplo, suponha que o Rafael entregou mais um relatório com pontuação 65. A listagem deveria ser atualizada para:

1) Rafael (100)
2) Paulo (70)
3) Rafael (65)
4) Ana (60)
5) Paulo (50)
6) Ana (40)
7) Ana (40)

Para gastar os créditos obtidos com os relatórios, o aluno deve pedir para retirar uma das suas entradas na listagem. Os créditos são proporcionais a colocação da entrada do aluno na listagem.

Por exemplo, se o Paulo quiser gastar uma das suas duas entradas na lista ele deve escolher entre a de 70 pontos a 50 pontos. A de 70 é a segunda da lista e a de 50 é a quinta. Suponha que ele escolha a de 50 pontos então a nova listagem ficaria assim:

1) Rafael (100)
2) Paulo (70)
3) Rafael (65)
4) Ana (60)
5) Ana (40)
6) Ana (40)

Quando o aluno quiser usar seus créditos ele deve verificar antes se ele tem entradas na listagem. Para isso, ele deve ir na secretária da instituição.

A instituição pode querer saber qual é o aluno que está na primeira posição da listagem ou o que está na última. Na verdade, seria interessante para a instituição poder facilmente saber qual é o aluno que está em qualquer posição que ela queira.

3.3 - Listas

Nesta seção, vamos definir uma estrutura de dados para resolver o problema da listagem de alunos. Chamaremos esta estrutura de **Lista**.

Vimos que uma estrutura de dados deve definir duas coisas:

1) A maneira como a informação será armazenada.
2) A interface de uso com o usuário.

Nos dois próximos capítulos, veremos as principais implementações de Listas. Cada implementação tem uma maneira particular de armazenar os dados, que trará vantagens e desvantagens em determinados casos, em termos de uso de memória e tempo consumido para cada operação. Cabe a você conhecer esses casos e saber fazer a melhor escolha conforme o problema enfrentado.

Neste capítulo, definiremos apenas a interface de uso que a Lista deve fornecer ao usuário.

Com certeza, vamos querer adicionar elementos então precisamos de algumas operações de adição. Talvez duas operações já sejam suficientes, uma que permita adicionar um elemento no fim da Lista e outra que deixe o usuário escolher a posição onde ele quer adicionar um elemento.

Também, precisaremos recuperar elementos então vamos definir uma operação que dado uma posição da Lista ela devolve o elemento que está lá.

Outro tipo de operação necessária e o de remover elementos. Esta operação deve receber a posição do elemento que deve ser removido.

Seria interessante que a Lista tivesse também uma operação Para verificar se um dado elemento está contido na Lista.

Por fim, uma operação que será bastante útil é uma que informe a quantidade de elementos da Lista.

Uma vez definidas todas as operações temos então a interface de uso da nossa estrura de dados.

Interface da Lista:

1) Adiciona um dado elemento no fim da Lista.
2) Adiciona um dado elemento em um dada posição.
3) Pega o elemento de uma dada posição.
4) Remove o elemento de uma dada posição.
5) Verifica se um dado elemento está contido na Lista.
6) Informa a quantidade de elementos da Lista.

Um fato bem interessante que ocorre quando programamos pensando primeiro na interface, como estamos fazendo aqui, é que após a definicão da interface já sabemos como usar a estrutura que ainda nem desenvolvemos.

Se sabemos como usar a estrutura já sabemos como testá-la. Estes testes poderão ser executados durante o desenvolvimento e não somente no fim. Isso é interessante pois possibilita a eliminar erros mais rápido, logo que eles apareçem, e pode evitar erros em cascata (erros que são causados por outros erros).

3.4 - Modelagem

Queremos desenvolver um sistema para resolver o problema da listagem de alunos. Este sistema deve ser orientado a objetos e deve de alguma forma representar os alunos. Em um sistema orientado a OBJETOS, um aluno será representado por um **objeto**.

Para criar objetos, precisamos definir como ele deve ser e o que ele deve fazer, ou seja, devemos criar uma "receita de construir objetos". Em termos técnicos esta "receita" é uma **Classe**.

```
public class Aluno {

    private String nome;

    public String getNome() {
        return nome;
    }

    public void setNome(String nome) {
        this.nome = nome;
    }

    public String toString() {
        return this.nome;
    }
```

```
    public boolean equals(Object o) {
        Aluno outro = (Aluno)o;
        return this.nome.equals(outro.nome);
    }
}
```

Com a classe **Aluno**, o sistema é capaz de criar objetos para representar os alunos da instituição. Teremos apenas alguns poucos atributos nessa classe, e alguns pares de getters e setters. Vale lembrar que não é boa prática ter classes que apenas carregam dados: seria mais interessante que **Aluno** também tivesse métodos de negócio.

Perceba que reescrevemos os métodos `toString()` e `equals(Object)`. O primeiro será útil para imprimir os alunos na tela. O segundo servirá para comparar dois objetos do tipo `Aluno`, o critério de comparação será os nomes dos alunos.

Devemos tomar cuidado no método `equals(Object)` pois estamos correndo o risco de dois tipos de erro. O primeiro acontece quando a referência recebida no parâmetro não está apontando para um objeto do tipo `Aluno`. O segundo ocorre quando ou a referência do parâmetro é **null** ou o atributo `nome` está **null**.

3.5 - Exercícios: Armazenamento

1) Crie um projeto no eclipse chamado **ed**. Não esqueça de selecionar a opção que separa o código fonte do binário.

2) Crie um pacote no projeto **ed** com o nome **br.com.caelum.ed**.

3) Faça a classe `Aluno` no pacote **br.com.caelum.ed** para poder criar objetos que representarão os alunos.

CAPÍTULO 4

Vetores

"A melhor maneira de fugir do seu problema é resolvê-lo"
– Robert Anthony

Vamos implementar uma Lista para resolver o problema da listagem de alunos. Lembrando que a interface da Lista já foi definida no capítulo de armazenamento sequencial, seguem as operações que devemos implementar:

1) Adiciona um dado aluno no fim da Lista.

2) Adiciona um dado aluno em uma dada posição.

3) Pega o aluno de dada posicão.

4) Remove o aluno de dada posicão.

5) Verifica se um dado aluno está armazenado.

6) Informa o número de alunos armazenados.

Ainda falta definir como os alunos serão armazenados. Como queremos manter muitos alunos vamos alocar um grande espaço de memória sequencial com capacidade para guardar uma certa quantidade de alunos, talvez 100 alunos por enquanto seja razoável.

Para facilitar o acesso aos alunos, dividiremos o espaço de memória alocado em pequenos pedaços idênticos. Cada pedaço armazenará um aluno. Além disso, vamos indexar (numerar) os pequenos pedaços para ser fácil recuperar um aluno.

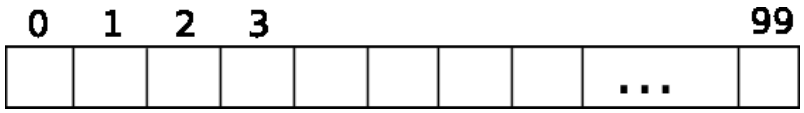


Fig.: Array

Praticamente todas as linguagens de programação têm um recurso similar ao que descrevemos acima. Em Java, este recurso é chamado de **Array**.

Um array é uma porção de memória fixa e sequencial dividida em pedaços idênticos indexados a partir do `0`. Em cada posição do array, podemos guardar um aluno. Na verdade, cada posição pode guardar uma **referência** para um objeto do tipo `Aluno`.

A capacidade de um array é fixa e deve ser informada no momento da criação do array. Não é possível redimensionar um array em Java, teremos de contornar isso mais adiante.

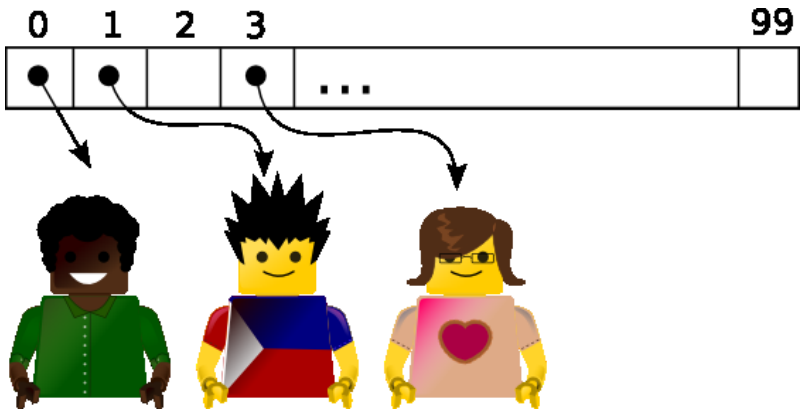


Fig.: Array de Referências

Uma Lista implementada com array muitas vezes é denomidada **Vetor**. Então criaremos uma classe chamada `Vetor` que armazena os alunos em array e tem todas as operações de uma Lista, encapsulando o acesso a esta Array.

```
public class Vetor {

    // Declarando e Inicializando um array de Aluno com capacidade 100.
    private Aluno[] alunos = new Aluno[100];

    public void adiciona(Aluno aluno) {
        // implementação
    }

    public void adiciona(int posicao, Aluno aluno) {
        // implementação
    }

    public Aluno pega(int posicao) {
        // implementação
    }

    public void remove(int posicao) {
        // implementação
    }

    public boolean contem(Aluno aluno) {
        // implementação
    }

    public int tamanho() {
        // implementação
    }
}
```

O código do `Vetor` acima não compila porque alguns de seus métodos não são **void**, obrigando a você retornar alguma coisa de um certo tipo. Se você quiser fazer com que o código acima compile, adicione alguns **return**s apropriados, como **0**, **null**s e **false**s.

4.1 - Os testes primeiro

Como temos a interface do Vetor já bem definida, podemos criar alguns testes antes mesmo de começar a implementar o código. Criaremos testes para cada uma das operações.

Adiciona no fim da lista

Teste:

```
public class TesteAdicionaNoFim {
    public static void main(String[] args) {
        Aluno a1 = new Aluno();
        Aluno a2 = new Aluno();

        a1.setNome("Rafael");
        a2.setNome("Paulo");

        Vetor lista = new Vetor();

        lista.adiciona(a1);
        lista.adiciona(a2);

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

Saída:

```
[Rafael, Paulo]
```

Adiciona em uma dada posição

Teste:

```
public class TesteAdicionaPorPosicao {
    public static void main(String[] args) {
        Aluno a1 = new Aluno();
        Aluno a2 = new Aluno();
        Aluno a3 = new Aluno();

        a1.setNome("Rafael");
        a2.setNome("Paulo");
        a3.setNome("Ana");

        Vetor lista = new Vetor();

        lista.adiciona(a1);
        lista.adiciona(0, a2);
        lista.adiciona(1, a3);

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

```
    }
}
```

Saída:

```
[Paulo, Ana, Rafael]
```

Pegar um aluno por posição

Teste:

```
public class TestePegaPorPosicao {
    public static void main(String[] args) {
        Aluno a1 = new Aluno();
        Aluno a2 = new Aluno();

        a1.setNome("Rafael");
        a2.setNome("Paulo");

        Vetor lista = new Vetor();

        lista.adiciona(a1);
        lista.adiciona(a2);

        Aluno aluno1 = lista.pega(0);
        Aluno aluno2 = lista.pega(1);

        System.out.println(aluno1);
        System.out.println(aluno2);
    }
}
```

Saída:

```
Rafael

Paulo
```

Remover um aluno por posição

Teste:

```
public class TesteRemovePorPosicao {
    public static void main(String[] args) {
        Aluno a1 = new Aluno();
        Aluno a2 = new Aluno();

        a1.setNome("Rafael");
        a2.setNome("Paulo");

        Vetor lista = new Vetor();
```

```
        lista.adiciona(a1);
        lista.adiciona(a2);

        lista.remove(0);

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

Saída:

```
[Paulo]
```

Verificar se a lista contem um dado aluno

Teste:

```
public class TesteContemAluno {
    public static void main(String[] args) {
        Aluno a1 = new Aluno();
        Aluno a2 = new Aluno();

        a1.setNome("Rafael");
        a2.setNome("Paulo");

        Vetor lista = new Vetor();

        lista.adiciona(a1);
        lista.adiciona(a2);

        System.out.println(lista.contem(a1));

        System.out.println(lista.contem(a2));

        Aluno aluno = new Aluno();
        aluno.setNome("Ana");

        System.out.println(lista.contem(aluno));
    }
}
```

Saída:

```
true

true

false
```

Informar o tamanho da lista

Teste:

```
public class TesteTamanhoLista {
    public static void main(String[] args) {
        Aluno a1 = new Aluno();
        Aluno a2 = new Aluno();
        Aluno a3 = new Aluno();

        a1.setNome("Rafael");
        a2.setNome("Paulo");

        Vetor lista = new Vetor();

        lista.adiciona(a1);
        lista.adiciona(a2);

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

        lista.adiciona(a3);

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

Saída:

```
2

3
```

Estes testes podem ser rodados a medida que preenchemos nosso `Vetor` com sua respectiva implementação. Em uma aplicação profissional Java, criaríamos **testes unitários**, utilizando bibliotecas auxiliares, como JUnit ou TestNG, para faciliar a escrita destes mesmos testes.

O desenvolvimento dirigido a testes (Test Driven Devlopment, TDD) é uma prática que ganha cada vez mais força, onde escreveríamos primeiro os testes das nossas classes, antes mesmo de começar a escrever a sua classe. O intuito disso é que você acaba apenas criando as classes e os métodos que realmente necessita, e eles já estão testados! O design da classe também costuma sair mais simples, pois uma classe com muitas dependências e acoplamento é difícil ser testada.

4.2 - Operações em vetores

Na seqüência, implementaremos cada uma das operações de uma Lista.

4.3 - Adicionar no fim da Lista

Esta operação receberá um aluno e irá guardá-lo no fim da Lista. Então, precisamos saber onde é o fim da Lista. A dificuldade em descobrir a última posição depende de como os alunos serão armazenados no array.

Há duas possibilidades: ou guardamos os alunos compactados a esquerda do array ou não. No primeiro caso, será bem mais fácil achar a última posição da Lista. Além disso, o índice do array será o mesmo índice da Lista.

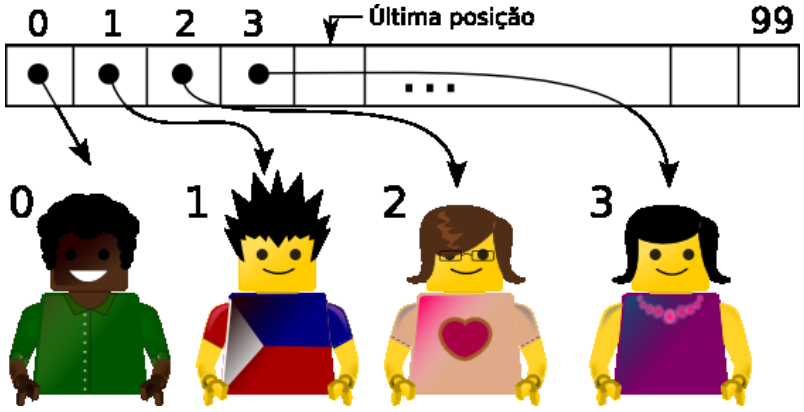


Fig.: Array compactado a esquerda

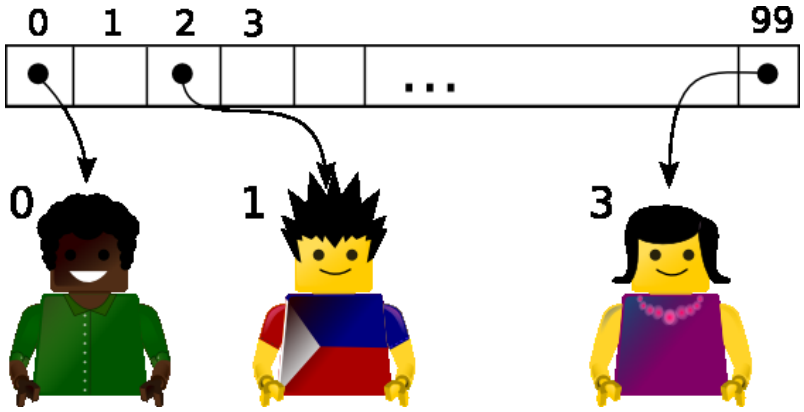


Fig.: Array não compactado a esquerda

Então, vamos definir que sempre os alunos serão armazenados compactados a esquerda no array, sem "buracos", que além de tudo economiza espaço.

Para achar a última posição da Lista ou a primeira posição vazia basta percorrer o array da esquerda para a direita até achar um valor **null** (lembrando que os arrays do Java guardam referências, e o valor padrão para referências é **null**). Achado a primeira posição vazia é só guardar o aluno nela.

Para percorrer o array usaremos um laço. Perceba que usamos o controlador de laço **break** para parar o **for** quando o aluno já foi adicionado.

```
public class Vetor {

    private Aluno[] alunos = new Aluno[100];

    public void adiciona(Aluno aluno) {
        for (int i = 0; i < this.alunos.length; i++) {
            if (this.alunos[i] == null) {
                this.alunos[i] = aluno;
                break;
            }
        }
```

```
    }
}
```

Neste ponto já seria interessante testar com o `TesteAdicionaNoFim`.

Fazendo uma análise simples deste método, é fácil ver que quanto mais alunos forem inseridos pior será o desempenho deste método. Por exemplo, se a Lista tem 50 alunos, o laço irá rodar 51 vezes para achar a primeira posição vazia.

Já que o consumo de tempo deste método piora proporcionalmente na medida que o número de elementos que existem na Lista aumenta, dizemos que o consumo é **linear**.

Será que tem alguma maneira de melhorar este consumo de tempo?

Uma pequena modificação é capaz de melhorar muito o desempenho do `adiciona(Aluno)`. Perceba que percorremos o array somente para procurar a primeira posição vazia. Mas isso é realmente necessário?

Vamos lembrar que o nosso array está compactado a esquerda então o índice da primeira posição vazia é igual a quantidade de elementos. Logo, se guardarmos a quantidade de elementos em algum lugar então no momento de adicionar um aluno já saberiamos qual é o índice da primeira posição vazia.

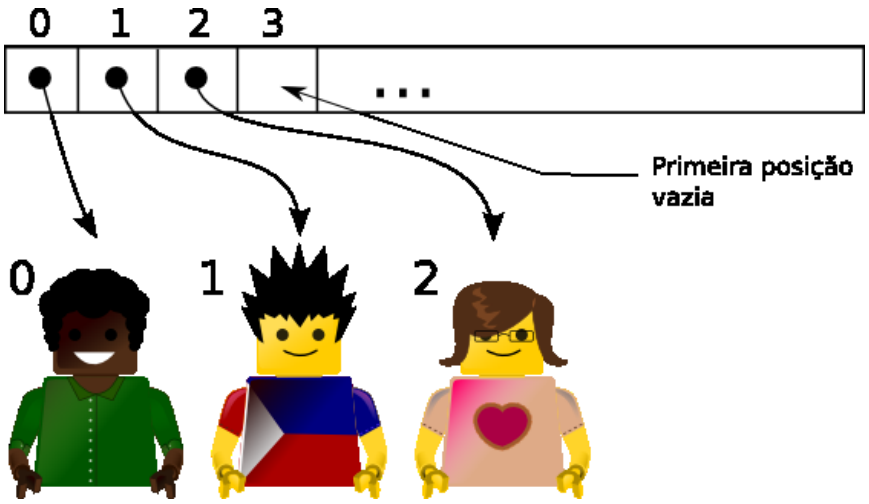


Fig.: Quantidade de elementos = Índice da primeira posição vazia

Para guardar a quantidade de alunos existentes na Lista definiremos um atributo na classe `Vetor` do tipo `int`.

```
public class Vetor {

    private Aluno[] alunos = new Aluno[100];

    private int totalDeAlunos = 0;

    public void adiciona(Aluno aluno) {
        this.alunos[this.totalDeAlunos] = aluno;
        this.totalDeAlunos++;
    }
```

```
}
```

Agora, o consumo de tempo do método é sempre o mesmo não importa quantos alunos estejam armazenados. Neste caso, dizemos que o consumo é **constante**.

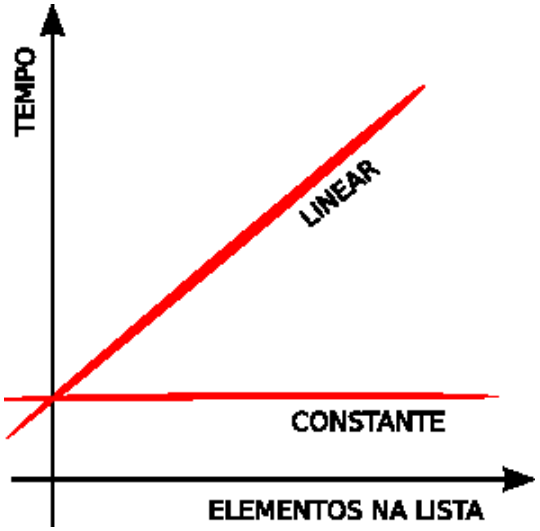


Fig.: Consumo Linear VS Consumo Constante

Um ponto interessante de enxergar aqui é que modificamos nossa **implementação** sem alterar nossa **interface**, e conseguimos deixar o método `adiciona` mais rápido. Com isso, se alguém já estivesse usando nossa classe `Vetor` antiga, poderia substituir pela nova sem alterar nenhuma outra linha de código. Se o acesso a nossa array fosse público, teríamos problemas nos códigos das pessoas que estão usando a nossa classe. Graças ao encapsulamento e a boa definição de nossa interface, evitamos ter de reescrever uma quantidade grande de código.

Para verificar se o método continua funcionando devemos executar novamente o `TesteAdicionaNoFim`.

4.4 - O método toString() para o Vetor

Vamos reescrever o método `toString()` para visualizar facilmente o conteúdo da Lista. Utilizamos a classe `StringBuilder` para construir a String que mostrará os elementos da Lista.

```
public String toString() {
    if (this.totalDeAlunos == 0) {
        return "[]";
    }

    StringBuilder builder = new StringBuilder();
    builder.append("[");

    for (int i = 0; i < this.totalDeAlunos - 1; i++) {
        builder.append(this.alunos[i]);
        builder.append(", ");
    }
```

```
        builder.append(this.alunos[this.totalDeAlunos - 1]);
        builder.append("]");

        return super.toString();
    }
```

4.5 - Informar o tamanho da Lista

Esta operação ficou muito simples de ser implementada porque a classe `Vetor` tem um atributo que guarda a quantidade de alunos armazenados. Então, basta devolver o valor do `totalDeAlunos`. Perceba que o consumo de tempo será constante: não há laços.

```
public class Vetor {

    ...
    private int totalDeAlunos = 0;

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

Se não tivessemos criado o atributo `totalDeAlunos` o método `tamanho()` teria que fazer um laço para percorrer o array inteiro e contar quantas posições estão ocupadas. Ou seja, o desempenho seria linear que é muito pior que constante.

Não podemos esquecer de rodar o teste para o tamanho da Lista.

4.6 - Verificar se um aluno está presente no vetor

Nesta operação, precisamos comparar o aluno dado com todos os alunos existentes na Lista. Para implementar esta funcionalidade faremos um laço.

```
public class Vetor {

    private Aluno[] alunos = new Aluno[100];

    private int totalDeAlunos = 0;

    ...
    public boolean contem(Aluno aluno) {
        for (int i = 0; i < this.alunos.length; i++) {
            if (aluno == this.alunos[i]) {
                return true;
            }
        }
        return false;
    }
}
```

Neste método, se o aluno procurado for encontrado então o valor **true** é devolvido. Se a array acabar e o aluno não for encontrado, significa que ele não está armazenado logo o método deve devolver falso. A capacidade do array é obtida pelo atributo `length`.

O nosso método é ineficiente quando a Lista tem poucos elementos. Perceba que ele sempre percorre o array todo. Não é necessário percorrer o array inteiro basta percorrer as posições ocupadas, ou seja, o laço tem que ir até a última posição ocupada. Nós podemos obter a última posição ocupada através do atributo `totalDeAlunos`.

```
public class Vetor {

    private Aluno[] alunos = new Aluno[100];

    private int totalDeAlunos = 0;

    ...
    public boolean contem(Aluno aluno) {
        for (int i = 0; i < this.totalDeAlunos; i++) {
            if (aluno == this.alunos[i]) {
                return true;
            }
        }
        return false;
    }
}
```

Nós estamos comparando os alunos com o operador `==`. Este operador compara o conteúdo das variáveis. No Java, as variáveis de tipos não primitivos, como o tipo `Aluno`, guardam referências para objetos. Então, na verdade, estamos comparando a referências e não os objetos propriamente.

Para comparar objetos devemos usar o método `equals(Object)`. Lembrando que reescrevemos este método para considerar que dois objetos do tipo `Aluno` são iguais quando os seus atributos `nome` são iguais.

```
public class Vetor {

    private Aluno[] alunos = new Aluno[100];

    private int totalDeAlunos = 0;

    ...
    public boolean contem(Aluno aluno) {
        for (int i = 0; i < this.totalDeAlunos; i++) {
            if (aluno.equals(this.alunos[i])) {
                return true;
            }
        }
        return false;
    }
}
```

Aqui deveríamos executar o teste do `contem`.

4.7 - Pegar o aluno de uma dada posição do array

Esta operação parece bem simples, ela deve simplesmente acessar e devolver o aluno da posição desejada.

```
public class Vetor {

    private Aluno[] alunos = new Aluno[100];
    ...

    ...
    public Aluno pega(int posicao) {
        return this.alunos[posicao];
    }
}
```

Mas não é tão simples assim: esquecemos de considerar a possibilidade do usuário pedir por uma posição inválida. Se descosiderarmos esta possibilidade vamos correr o risco de acessar uma posição vazia ou inexistente. Então, antes de acessar a posição, vamos verificar se ela está ocupada.

Será criado o método `posicaoOcupada(int)` que devolve verdadeiro se a posição estiver ocupada, e falso caso contrário. Uma posição é válida se ela pertence ao intervalo fechado [0, this.totalDeAlunos - 1].

```
public class Vetor {
    private Aluno[] alunos = new Aluno[100];

    private int totalDeAlunos = 0;

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

A maneira que o método `posicaoOcupada(int)` foi implementado é bem interessante pois ela não usa `if`s.

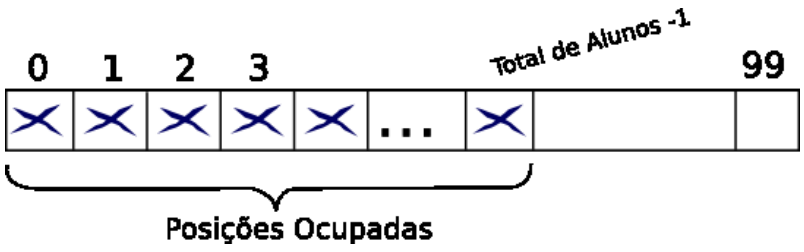


Fig.: Posições ocupadas

É importante observar que o método `posicaOcupada(int)` deve ser privado pois não deve ser acessado por quem está usando a classe `Vetor`. Desta forma, o sistema fica mais encapsulado.

Agora, o método `pega(int)` pode invocar o `posicaoOcupada(int)` para saber se a posição está ocupada ou não. Caso a posição seja válida, o `pega(int)` devolve o aluno correspondete e caso contrário, ele lança uma exceção (optamos por `IllegalArgumentException`).

As exceções são uma maneira de informar para quem chamou um método que algo aconteceu de maneira diferente da comum.

```
public class Vetor {
    private Aluno[] alunos = new Aluno[100];

    private int totalDeAlunos = 0;

    ...
    public Aluno pega(int posicao) {
        if (!this.posicaoOcupada(posicao)) {
            throw new IllegalArgumentException("Posicao inválida");
        }
        return this.alunos[posicao];
    }

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

Feito o método devemos testá-lo com o teste definido anteriormente.

4.8 - Adicionar um aluno em uma determinada posição do array

A operação de adicionar um aluno em uma determinada posição é mais delicada. Primeiro, precisamos verificar se a posição faz sentido ou não. Só podemos adicionar um aluno em alguma posição que já estava ocupada ou na primeira posição vazia da Lista.

Para verificar se podemos adicionar um aluno em uma dada posição devemos testar se a posição está no intervalo `[0, this.totalDeAlunos]`. Vamos criar um método para isolar esta verificação.

```
public class Vetor {
    private Aluno[] alunos = new Aluno[100];

    private int totalDeAlunos = 0;

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

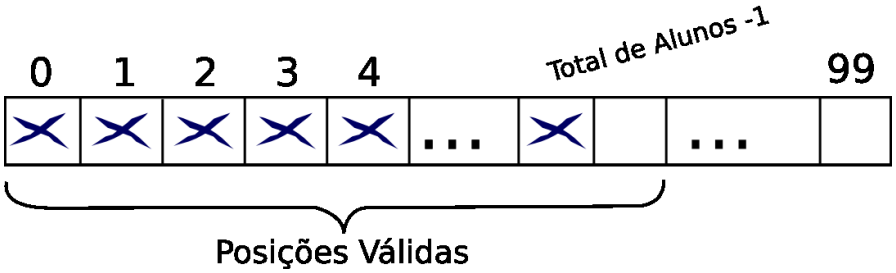


Fig.: Posições válidas

Após verificar se podemos adicionar o aluno na posição dada, devemos tomar cuidado para não colocar um aluno sobre outro. É preciso deslocar todos os alunos a "direita" da posição onde vamos inserir uma vez para a "frente". Isso abrirá um espaço para guardar a referência para o aluno novo.

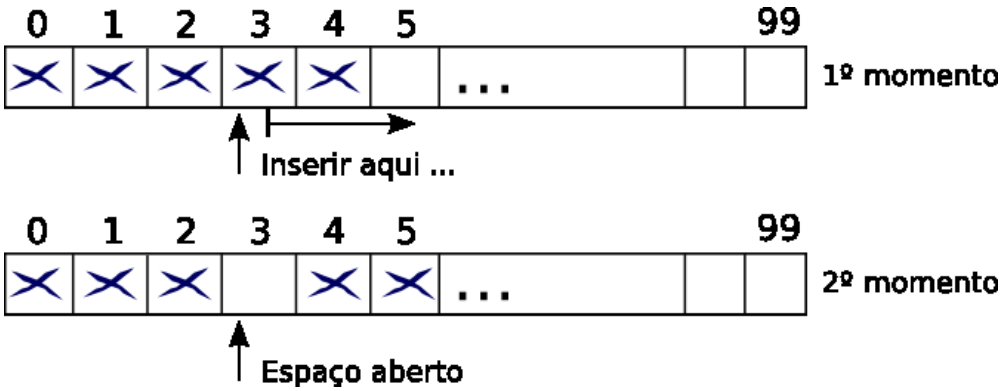


Fig.: Deslocamento para a direita

```
public class Vetor {
    private Aluno[] alunos = new Aluno[100];

    private int totalDeAlunos = 0;

    ...
    public void adiciona(int posicao, Aluno aluno) {
        if (!this.posicaoValida(posicao)) {
            throw new IllegalArgumentException("Posicao inválida");
        }

        for (int i = this.totalDeAlunos - 1; i >= posicao; i-) {
            this.alunos[i + 1] = this.alunos[i];
        }

        this.alunos[posicao] = aluno;
        this.totalDeAlunos++;
    }

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

Quanto este método consome de tempo? Depende! Se for a última posição, consumirá tempo contante. No caso de ser a primeira posição, ele terá de deslocar **todos** os elementos para a direita, consumindo tempo linear em relação ao número de elementos distantes. É comum calcularmos o tempo consumido de um método pensando sempre no pior caso, então diremos que o método que adiciona em qualquer posição de nosso Vetor consome tempo linear.

Agora é um ótimo momento para testar. Podemos rodar o teste de adicionar por posição.

4.9 - Remover um aluno de uma dada posição

Inicialmente, precisamos verificar se a posição está ocupada ou não. Afinal, não faz sentido remover algo que não existe. Para saber se a posição está ocupada ou não podemo usar o método `posicaoOcupada(int)`.

Se a posição estiver ocupada então podemos remover o aluno. Além disso, precisamos deslocar os alunos que estavam a direita daquele que removemos uma vez para esquerda para fechar o "buraco" aberto pela remoção.

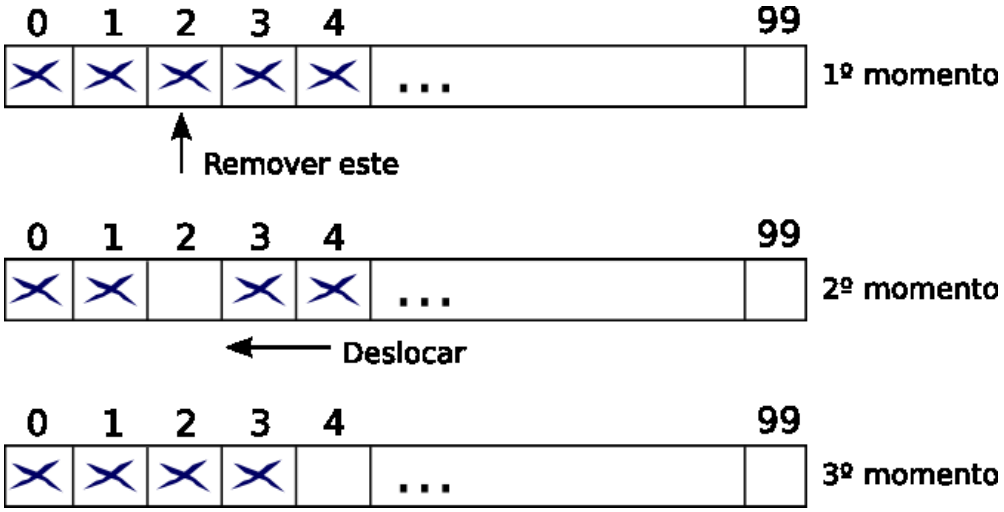


Fig.: Deslocamento para a esquerda

```
public class Vetor {
    private Aluno[] alunos = new Aluno[100];

    private int totalDeAlunos = 0;

    ...
    public void remove(int posicao) {
        if (!this.posicaoOcupada(posicao)) {
            throw new IllegalArgumentException("Posicao inválida");
        }
        for (int i = posicao; i < this.totalDeAlunos - 1; i++) {
            this.alunos[i] = this.alunos[i + 1];
        }
        this.totalDeAlunos-;
    }

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

Você sabe dizer quanto tempo este método consome?

E agora falta executar o teste para esta operação.

4.10 - Alocação Dinâmica

Há um grande problema na implementação apresentada até o momento. Imagine que o vetor já contenha 100 alunos. Ao adicionar mais um aluno ocorreria um erro pois o array foi criado com 100 posições e o método `adiciona()` tentaria inserir um aluno em uma posição que não existe: o java lançará um exceção.

Para resolver isso, podemos tentar inicializar o array com um tamanho maior. Isso não resolveria problema. Por exemplo, se o tamanho da array fosse 200 em vez de 100, no momento que fosse inserido o aluno número 201, ocorreria novamente um erro.

Uma abordagem mais eficiente seria cada vez que o array ficar cheio alguma providência seja tomada, como, por exemplo, dobrar o tamanho dele.

Vamos criar um método que tem como tarefa verificar se o array está cheio. Caso estiver cheio, ele criará um novo array com o dobro do tamanho do antigo e moverá os alunos do array antigo para o novo.

```
public class Vetor {
    private Aluno[] alunos = new Aluno[100];

    private int totalDeAlunos = 0;

    ...
    private void garantaEspaco() {
        if (this.totalDeAlunos == this.alunos.length) {
            Aluno[] novaArray = new Aluno[this.alunos.length * 2];
            for (int i = 0; i < this.alunos.length; i++) {
                novaArray[i] = this.alunos[i];
            }
            this.alunos = novaArray;
        }
    }
}
```

O risco de tentar adicionar um aluno sem ter posição disponível só ocorre, evidentemente, nos métodos de adicionar aluno. Então, para evitar este problema, vamos verificar se existe espaço disponível antes de adicionar um aluno.

```
public class Vetor {
    private Aluno[] alunos = new Aluno[100];

    private int totalDeAlunos = 0;

    ...
    public void adiciona(Aluno aluno) {
        this.garantaEspaco();
        this.alunos[this.totalDeAlunos] = aluno;
        this.totalDeAlunos++;
    }

    public void adiciona(int posicao, Aluno aluno) {
        this.garantaEspaco();
        if (!this.posicaoValida(posicao)) {
            throw new IllegalArgumentException("Posicao inválida");
        }
```

```
        for (int i = this.totalDeAlunos - 1; i >= posicao; i-) {
            this.alunos[i + 1] = this.alunos[i];
        }

        this.alunos[posicao] = aluno;
        this.totalDeAlunos++;
    }
}
```

O método `garantaEspaco()` só é útil dentro da classe `Vetor`, ou seja, não deve ser disponibilizado para o usuário então ele deve ser um método **private**.

4.11 - Generalização

A implementação de vetor feita até agora funciona muito bem para armazenar alunos. Porém, não serve para armazenar nenhum outro tipo de objeto. Nossa estrutura de dados está muito atrelada ao tipo de dado que ela armazena (Alunos). Se amanhã ou depois precisarmos de uma Lista de carro ou uma Lista de computador teríamos que implementar novamente o Vetor.

Em vez de colocarmos um array de `Aluno` na classe `Vetor` vamos colocar um array de `Object`. Assim, estamos generalizando a nossa estrutura de dados. Desta forma, poderemos armazenar qualquer tipo de objeto.

```
public class Vetor {

    private Object[] objetos = new Object[100];

    private int totalDeObjetos = 0;

    public void adiciona(Object objeto) {
        this.garantaEspaco();
        this.objetos[this.totalDeObjetos] = objeto;
        this.totalDeObjetos++;
    }

    public void adiciona(int posicao, Aluno aluno) {
        this.garantaEspaco();
        if (!this.posicaoValida(posicao)) {
            throw new IllegalArgumentException("Posicao inválida");
        }

        for (int i = this.totalDeObjetos - 1; i >= posicao; i-) {
            this.objetos[i + 1] = this.objetos[i];
        }

        this.objetos[posicao] = aluno;
        this.totalDeObjetos++;
    }

    public Object pega(int posicao) {
        if (!this.posicaoOcupada(posicao)) {
            throw new IllegalArgumentException("Posicao inválida");
```

```
        }
        return this.objetos[posicao];
    }

    public void remove(int posicao) {
        if (!this.posicaoOcupada(posicao)) {
            throw new IllegalArgumentException("Posicao inválida");
        }
        for (int i = posicao; i < this.totalDeObjetos - 1; i++) {
            this.objetos[i] = this.objetos[i + 1];
        }
        this.totalDeObjetos-;
    }

    public boolean contem(Aluno aluno) {
        for (int i = 0; i < this.totalDeObjetos; i++) {
            if (aluno.equals(this.objetos[i])) {
                return true;
            }
        }
        return false;
    }

    public int tamanho() {
        return this.totalDeObjetos;
    }

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

    private boolean posicaoValida(int posicao) {
        return posicao >= 0 && posicao <= this.totalDeObjetos;
    }

    private void garantaEspaco() {
        if (this.totalDeObjetos == this.objetos.length) {
            Object[] novaArray = new Object[this.objetos.length * 2];
            for (int i = 0; i < this.objetos.length; i++) {
                novaArray[i] = this.objetos[i];
            }
            this.objetos = novaArray;
        }
    }
}
```

No Java todas as classes herdam, diretamente ou indiretamente, da classe `Object`. Então, um objeto de qualquer tipo pode ser referenciado com uma variável do tipo `Object`. Este conceito de referenciar um mesmo objeto de várias maneira (`Aluno` ou `Object`) é chamado de **polimorfismo**.

O que ganhamos com esta generalização foi um forte reaproveitamento da classe `Vetor`, porém na hora do uso perdemos a segurança da tipagem do java. Isso acarretará na necessidade do uso de **casting**.

```
Vetor lista = new Vetor();
```

```
// Inserindo uma String
lista.adiciona("Rafael");

// Fazendo um casting de String para Aluno. Erro de EXECUÇÃO.
Aluno aluno = (Aluno)lista.pega(0);
```

Existe uma outra maneira de fazer a mesma classe sem essa desvantagem de usar castings, que é criar uma classe parametrizada, um recurso existente no Java a partir da versão 5.

4.12 - API do Java

Na biblioteca do Java, há uma classe que implementa a estrutura de dados que foi vista neste capítulo, esta classe chama-se **ArrayList** e será testada pelo código abaixo.

```
public class Teste {

    public static void main(String[] args) {

        ArrayList vetor = new ArrayList();

        Aluno aluno1 = new Aluno();
        Aluno aluno2 = new Aluno();
        Aluno aluno3 = new Aluno();

        vetor.add(aluno1);
        vetor.add(aluno2);
        vetor.add(0, aluno3);

        int tamanho = vetor.size();

        if (tamanho != 3) {
            System.out.println("Erro: O tamanho da lista está errado.");
        }

        if (!vetor.contains(aluno1)) {
            System.out
                    .println("Erro: Não achou um aluno que deveria estar na lista");
        }

        vetor.remove(0);
        tamanho = vetor.size();

        if (tamanho != 2) {
            System.out.println("Erro: O tamanho da lista está errado.");
        }

        if (vetor.contains(aluno3)) {
            System.out
                    .println("Erro: Achou um aluno que não deveria estar na lista");
        }
    }
}
```

A classe `Vector` é muito similar a `ArrayList`, a grande diferença é que `ArrayList` não é segura para ser compartilhada entre várias threads simultâneamente sem o devido cuidado. Dizemos que `Vector` é thread safe, mas isso tem um custo, e é por isso que evitamos usar `Vector` e preferimos usar `ArrayList` sempre que possível.

Para evitar fazer casting de objetos, podemos utilizar o recurso de generics do Java 5. A utilização de generics é bem simples, a gente deve informar que o nossa Lista irá guardar alunos. Isso é feito como mostra o código a seguir:

```
public class Teste {
    public static void main(String[] args) {

        ArrayList vetorSemGenerics = new ArrayList();
        ArrayList<Aluno> vetorComGenerics = new ArrayList<Aluno>();

        Aluno aluno = new Aluno();

        vetorSemGenerics.add(aluno);
        vetorComGenerics.add(aluno);

        Aluno a1 = (Aluno) vetorSemGenerics.get(0);
        Aluno a2 = vetorComGenerics.get(0);
    }
}
```

Com o generics temos uma segurança em tempo de compilação em relação a tipagem dos objetos. Se tentarmos adicionar um objeto que não é do tipo `Aluno` um erro de **compilação** acontecerá.

```
ArrayList<Aluno> vetorComGenerics = new ArrayList<Aluno>();
vetorComGenerics.add("Rafael"); // erro de compilação
```

Qual a vantagem de um erro de compilação sobre um erro de execução? O de execução acontece quando o usuário está do lado do computador. O de compilação acontece quando o programador está no computador.

4.13 - Exercícios: Vetores

1) Crie a classe `Vetor` no pacote **br.com.caelum.ed.vetores** com as assinaturas dos métodos vistos neste capítulo e com um atributo do tipo array de Aluno inicializado com `100000` posições.

```
package br.com.caelum.ed.vetores;

import br.com.caelum.ed.Aluno;

public class Vetor {

    // Declarando e Inicializando um array de Aluno com capacidade 100.
    private Aluno[] alunos = new Aluno[100000];

    public void adiciona(Aluno aluno) {
        // implementação
    }

    public void adiciona(int posicao, Aluno aluno) {
```

```
        // implementação
    }

    public Aluno pega(int posicao) {
        // implementação
        return null;
    }

    public void remove(int posicao) {
        // implementação
    }

    public boolean contem(Aluno aluno) {
        // implementação
        return false;
    }

    public int tamanho() {
        // implementação
        return 0;
    }
}
```

2) Escreva os testes unitários vistos neste capítulo. Coloque os testes no pacote **br.com.caelum.ed.vetores.testes**.

Teste:

```
public class TesteAdicionaNoFim {
    public static void main(String[] args) {
        Aluno a1 = new Aluno();
        Aluno a2 = new Aluno();

        a1.setNome("Rafael");
        a2.setNome("Paulo");

        Vetor lista = new Vetor();

        lista.adiciona(a1);
        lista.adiciona(a2);

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

Saída:

```
[Rafael, Paulo]
```

Teste:

```
public class TesteAdicionaPorPosicao {
    public static void main(String[] args) {
        Aluno a1 = new Aluno();
        Aluno a2 = new Aluno();
        Aluno a3 = new Aluno();
```

```
        a1.setNome("Rafael");
        a2.setNome("Paulo");
        a3.setNome("Ana");

        Vetor lista = new Vetor();

        lista.adiciona(a1);
        lista.adiciona(0, a2);
        lista.adiciona(1, a3);

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

Saída:

```
[Paulo, Ana, Rafael]
```

Teste:

```
public class TestePegaPorPosicao {
    public static void main(String[] args) {
        Aluno a1 = new Aluno();
        Aluno a2 = new Aluno();

        a1.setNome("Rafael");
        a2.setNome("Paulo");

        Vetor lista = new Vetor();

        lista.adiciona(a1);
        lista.adiciona(a2);

        Aluno aluno1 = lista.pega(0);
        Aluno aluno2 = lista.pega(1);

        System.out.println(aluno1);
        System.out.println(aluno2);
    }
}
```

Saída:

```
Rafael
Paulo
```

Teste:

```
public class TesteRemovePorPosicao {
    public static void main(String[] args) {
        Aluno a1 = new Aluno();
        Aluno a2 = new Aluno();

        a1.setNome("Rafael");
        a2.setNome("Paulo");
```

```
        Vetor lista = new Vetor();

        lista.adiciona(a1);
        lista.adiciona(a2);

        lista.remove(0);

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

Saída:

```
[Paulo]
```

Teste:

```
public class TesteContemAluno {
    public static void main(String[] args) {
        Aluno a1 = new Aluno();
        Aluno a2 = new Aluno();

        a1.setNome("Rafael");
        a2.setNome("Paulo");

        Vetor lista = new Vetor();

        lista.adiciona(a1);
        lista.adiciona(a2);

        System.out.println(lista.contem(a1));

        System.out.println(lista.contem(a2));

        Aluno aluno = new Aluno();
        aluno.setNome("Ana");

        System.out.println(lista.contem(aluno));
    }
}
```

Saída:

```
true
true
false
```

Teste:

```
public class TesteTamanhoLista {
    public static void main(String[] args) {
        Aluno a1 = new Aluno();
        Aluno a2 = new Aluno();
        Aluno a3 = new Aluno();
```

```
        a1.setNome("Rafael");
        a2.setNome("Paulo");

        Vetor lista = new Vetor();

        lista.adiciona(a1);
        lista.adiciona(a2);

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

        lista.adiciona(a3);

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

Saída:

```
2
3
```

3) Uma vez definida a interface do Vetor poderíamos programar os testes. Para os testes vamos reescrever o `toString()`.

```
public String toString() {
    if (this.totalDeAlunos == 0) {
        return "[]";
    }

    StringBuilder builder = new StringBuilder();
    builder.append("[");

    for (int i = 0; i < this.totalDeAlunos - 1; i++) {
        builder.append(this.alunos[i]);
        builder.append(", ");
    }

    builder.append(this.alunos[this.totalDeAlunos - 1]);
    builder.append("]");

    return super.toString();
}
```

4) Implemente o método `adiciona(Aluno)` da primeira maneira vista neste capítulo.

```
public void adiciona(Aluno aluno) {
    for (int i = 0; i < this.alunos.length; i++) {
        if (this.alunos[i] == null) {
            this.alunos[i] = aluno;
            break;
        }
    }
}
```

Faça um teste para calcular o tempo gasto. Este teste deve adicionar `100000` alunos. Execute e marque o tempo.

```
package br.com.caelum.ed.vetores;

import br.com.caelum.ed.Aluno;

public class TesteLinearVSConstante {
    public static void main(String[] args) {
        Vetor vetor = new Vetor();
        long inicio = System.currentTimeMillis();
        for (int i = 1; i < 100000; i++) {
            Aluno aluno = new Aluno();
            vetor.adiciona(aluno);
        }
        long fim = System.currentTimeMillis();
        double tempo = (fim - inicio) / 1000.0;
        System.out.println("Tempo em segundos = " + tempo);
    }
}
```

Implemente o método `adiciona(Aluno)` da segunda maneira vista neste capítulo. Não esqueça de acrescentar o atributo `totalDeAlunos`.

```
public class Vetor {

    private Aluno[] alunos = new Aluno[100000];

    private int totalDeAlunos = 0;

    public void adiciona(Aluno aluno) {
        this.alunos[this.totalDeAlunos] = aluno;
        this.totalDeAlunos++;
    }
}
```

Execute a classe `TesteLinearVSConstante` e veja o tempo agora. A diferença de tempo é bem considerável.

5) Implemete o método `tamanho()` na classe `Vetor` como visto neste capítulo.

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

Execute o teste apropriado feito anteriormente.

6) Implemente o método `contem()` na classe `Vetor` da primeira maneira mostrada neste capítulo.

```
public boolean contem(Aluno aluno) {
    for (int i = 0; i < this.alunos.length; i++) {
        if (aluno == this.alunos[i]) {
            return true;
        }
    }
    return false;
```

```
}
```

Verifique este método com o teste do contem aluno. Faça outro teste para calcular o tempo gasto.

```
package br.com.caelum.ed.vetores;

import br.com.caelum.ed.Aluno;

public class TesteDeTempoDoContem {
    public static void main(String[] args) {
        Vetor vetor = new Vetor();
        long inicio = System.currentTimeMillis();

        // Adicionado 100000 alunos e
        // Verificando se eles foram realmente adicionados.
        for (int i = 1; i < 30000; i++) {
            Aluno aluno = new Aluno();
            vetor.adiciona(aluno);
            if(!vetor.contem(aluno)){
                System.out.println("Erro: aluno "  + aluno + " não foi adicionado.");
            }
        }

        // Verificando se o Vetor não encontra alunos não adicionados.
        for (int i = 1; i < 30000; i++) {
            Aluno aluno = new Aluno();
            if(vetor.contem(aluno)){
                System.out.println("Erro: aluno "  + aluno + " foi adicionado.");
            }
        }

        long fim = System.currentTimeMillis();
        double tempo = (fim - inicio) / 1000.0;
        System.out.println("Tempo = " + tempo);
    }
}
```

Implemente o método `contem()` na classe `Vetor` da segunda maneira mostrada neste capítulo.

```
public boolean contem(Aluno aluno) {
    for (int i = 0; i < this.totalDeAlunos; i++) {
        if (aluno == this.alunos[i]) {
            return true;
        }
    }
    return false;
}
```

Execute o teste novamente e veja a diferença de tempo.

7) Implemente o método `pega(int)` na classe `Vetor` da primeira maneira vista neste capítulo.

```
public Aluno pega(int posicao) {
    return this.alunos[posicao];
}
```

Teste este método! Depois faça um teste pegando uma posição ocupada, uma vazia e uma que não existe.

```
public class Teste {
    public static void main(String[] args) {
        Aluno a1 = new Aluno();

        a1.setNome("Rafael");

        Vetor1 lista = new Vetor1();

        lista.adiciona(a1);

        lista.pega(0);

        lista.pega(1);

        lista.pega(100000000);
    }
}
```

Este teste deve gerar um erro somente na última linha.

Implemente o método `pega(int)` na classe `Vetor` da segunda maneira vista neste capítulo. Não esqueça de implementar o método `posicaoOcupada(int)` também.

```
public Aluno pega(int posicao) {
    if (!this.posicaoOcupada(posicao)) {
        throw new IllegalArgumentException("Posicao inválida");
    }
    return this.alunos[posicao];
}

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

Rode o teste novamente e agora veja que o erro já ocorre no segundo “pega”. Este é o resultado esperado pois ao pegar uma posição não ocupada deve ocorrer erro.

8) Implemente o método `adiciona(int, Aluno)` na classe `Vetor`. Não esqueça do método `posicaoValida(int)`.

```
public void adiciona(int posicao, Aluno aluno) {
    if (!this.posicaoValida(posicao)) {
        throw new IllegalArgumentException("Posicao inválida");
    }

    for (int i = this.totalDeAlunos - 1; i >= posicao; i-) {
        this.alunos[i + 1] = this.alunos[i];
    }

    this.alunos[posicao] = aluno;
    this.totalDeAlunos++;
}
```

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

Rode o teste adequado!

9) Implemente o método `remove(int)` na classe `Vetor`.

```
public void remove(int posicao) {
    if (!this.posicaoOcupada(posicao)) {
        throw new IllegalArgumentException("Posicao inválida");
    }
    for (int i = posicao; i < this.totalDeAlunos - 1; i++) {
        this.alunos[i] = this.alunos[i + 1];
    }
    this.totalDeAlunos-;
}
```

Faça o teste do remove para verificar se tudo deu certo.

10) Exetute o seguinte teste:

```
package br.com.caelum.ed.vetores;

import br.com.caelum.ed.Aluno;

public class TesteEstoura {
    public static void main(String[] args) {
        Vetor vetor = new Vetor();
        for (int i = 0; i < 100001; i++) {
            Aluno aluno = new Aluno();
            vetor.adiciona(aluno);
        }
    }
}
```

Um erro ocorre pois a capacidade do Vetor estoura.

Implemente o método `garantaEspaco()` na classe `Vetor` para evitar o problema de estourar a capacidade.

```
private void garantaEspaco() {
    if (this.totalDeAlunos == this.alunos.length) {
        Aluno[] novaArray = new Aluno[this.alunos.length * 2];
        for (int i = 0; i < this.alunos.length; i++) {
            novaArray[i] = this.alunos[i];
        }
        this.alunos = novaArray;
    }
}
```

Não esqueça de invocar o método `garantaEspaco()` dentro dos dois métodos de adicionar alunos.

Execute a classe `TesteEstoura` mais uma vez.

11) Implemente a classe `Vetor` para objetos (Genérico) com os métodos e atributos vistos neste capítulo na seção de generalização.

12) Utilize a classe `Vetor` que foi implementada nos exercícios anteriores e as classes da API do Java `Vector` ou `ArrayList`. Para saber os métodos das classes da API, utilize a documentação do Java (http://java.sun.com/j2se/1.5.0/docs/api/index.html).

    1) Crie um vetor usando a classe `Vetor`; guarde nele 1000 alunos;
    2) Imprima o tamanho do vetor antes e depois de adicionar os 1000 alunos.
    3) Crie um vetor usando a classe `Vector`; passe os alunos do vetor anterior para este novo vetor; imprima o tamanho do novo vetor antes e depois de adicionar os alunos.
    4) Crie um vetor usando a classe \classe{ArrayList}; passe os alunos do vetor criado no item anterior para este novo vetor; imprima o tamanho do novo vetor antes e depois de adiciomar os alunos.

4.14 - Exercícios opcionais

1) Use o recurso de generics do Java 5 e crie um Vetor usando a classe `ArrayList` para guardar objetos do tipo `String`. Teste adicionar neste vetor algumas `Strings`; e também tente adicionar `Alunos` (O que acontecerá quando você tentar adicionar alunos?). Retire elementos deste vetor (você presisa fazer casting para `String` nestes elementos?).

2) Acrescente uma operação na Lista, para isso, implemente um novo método. A nova operação deve remover da Lista todas as ocorrências de um elemento que é passado com parametro. Não esqueça de rearranjar os elementos do vetor após a remoção.

```
public void remove(Object objeto) {
    // implementação
}
```

3) Acrescente uma operação na Lista, para isso, implemente um novo método. A nova operação deve limpar a lista, ou seja, remover todos os elementos.

```
public void clear() {
    // implementação
}
```

4) Acrescente uma operação na Lista, para isso, implemente um novo método. A nova operação deve procurar o índice da primeira ocorrência de um elemento passado como parâmetro.

```
public void indexOf(Object objeto) {
    // implementação
}
```

5) Acrescente uma operação na Lista, para isso, implemente um novo método. A nova operação deve procurar o índice da última ocorrência de um elemento passado como parâmetro.

```
public void lastIndexOf(Object objeto) {
    // implementação
}
```

6) (Desafio) Pesquise sobre análise amortizada para saber porque é mais eficiente dobrar a capacidade da array quando o vetor fica cheio. A outra opção seria incrementar. Por exemplo, quando acaba o espaço na

array, criaríamos uma com o mesmo tamanho + 10. É incrível a diferença de uma estratégia e de outra, ao você tentar adicionar uma enorme quantidade de elementos.

CAPÍTULO 5

Listas Ligadas

"É impossível para um homem aprender aquilo que ele acha que já sabe"
– Epíteto

Vetores (listas implementadas com array) são uma excelente alternativa ao uso direto de arrays em Java, porém para algumas tarefas eles podem não ser eficientes, justamente por esse fato.

Adicionar um elemento na primeira posição de um Vetor, por exemplo, consome muito tempo, pois temos de deslocar todos os outros elementos uma posição para a frente. A performance dessa operação degrada a medida que a quantidade de elementos do nosso vetor cresce: ela consome tempo **linear** em relação ao número de elementos.

Analogamente, remover um elemento da primeira posição implica em deslocar todos os outros elementos que estão na sua frente para tras.

Em alguns casos, queremos uma implementação de Lista na qual a operação de adicionar ou a de remover um aluno na primeira posição seja computacionalmente eficiente. Conseguiremos isso através de uma **Lista Ligada**[1].

Esta estrutura que vamos criar terá a inserção e remoção nas "pontas" da lista computacionalmente eficientes.

Lembrando que as operações da Lista para o nosso sistema de gerenciamento de alunos são:

1) Adiciona um dado aluno no fim da Lista.

2) Adiciona um dado aluno em uma dada posição.

3) Pega o aluno de dada posicão.

4) Remove o aluno de dada posicão.

5) Verifica se um dado aluno está armazenado.

6) Informar o número de alunos armazenados.

[1] alguns textos se referem a essa estrutura como lista encadeada

5.1 - Solução clássica de Lista Ligada

Para manipular uma estrutura sem ter de alterar todos seus vizinhos, precisamos de algo que não seja fixo. Uma array é uma estrutura fixa na memória, precisamos sempre empurrar todos seus vizinhos para um lado para conseguir espaço, ou para remover algum dos elementos, para não deixar espaços vazios.

Então a idéia aqui é ter uma forma de, dado um aluno, saber quem é o próximo, sem usar uma estrutura fixa. Podemos mudar a própria classe `Aluno` modificada! Repare no código abaixo:

```
public class AlunoLista {

    private String nome;

    private int idade;

    private AlunoLista proximo;
}
```

A classe `AlunoLista` possui uma referência para um objeto de seu próprio tipo! É comum aqui o estudante ficar confuso em relação a isso, como se esse código fosse gerar um laço infinito, ou uma recursão infinita. Mas não é o caso, já que a declaração do atributo `proximo` é apenas uma referência, quee não cria uma instância de `AlunoLista` na memória!

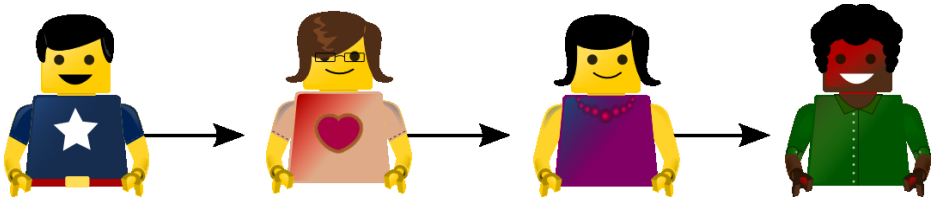


Fig.: Lista Ligada

Repare que, apesar do efeito de um aluno estar “ao lado” do outro, na memória é mais provável que eles não se encontrem um ao lado do outro, e sim em regiões bem diferentes da memória, só que cada um deles sabe dizer em que local se encontra o próximo aluno (pois temos a referência ao `proximo`).

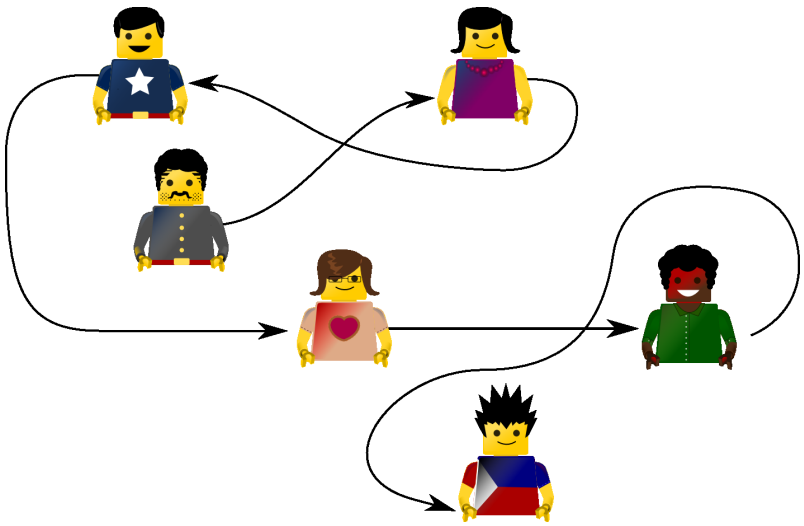


Fig.: Lista Ligada na memória

O ruim de um código assim é que estamos misturando responsabilidades em uma classe muito estranha! A `AlunoLista` além de guardar informações sobre um aluno, guarda também uma Lista de alunos. Manipulá-la pode ficar estranho e trabalhoso. Precisaríamos sempre manter uma referência para o primeiro aluno na memória em algum outro lugar, o que também pode ser confuso!

Além disso, quando precisarmos ter uma Lista Ligada de peças de carro, precisaríamos criar uma classe PecaLista parecidíssima com a AlunoLista, e não estaríamos reaproveitando código (além de ficar sucetível a copiar e colar modificações, uma péssima prática de programação, que pode gerar muitos erros).

Em vez de usar essa estrutura engessada, vamos utilizar uma classe separada como célula [label algumas pessoas gostam de chamar essa estrutura de nó], evitando a mistura da nossa classe de dados (a `Aluno`) da nossa estrutura.

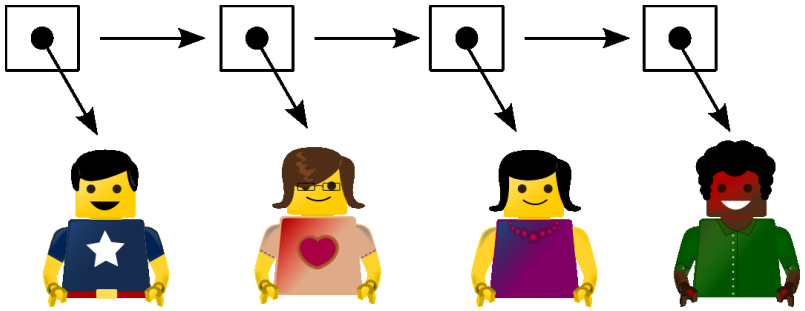


Fig.: Lista Ligada com Célula

5.2 - Célula e Lista Ligada

Vamos então isolar a manipulação de nossa estrutura de dados, para isso teremos uma classe para representar uma célula. Ela deve possuir uma referência para o elemento a qual ela se refere, e uma referência para a próxima célula [label pode ser interessante uma célula guardar referência também para a sua anterior, formando uma lista duplamente ligada], que pode ser **null** caso essa seja a útima célula da Lista.

A classe `Celula` a seguir possui apenas esses dois atributos e getters e setters, além de construtores para facilitar nosso trabalho.

```
public class Celula {

    private Celula proxima;

    private Object elemento;

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

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

```
    public void setProxima(Celula proxima) {
        this.proxima = proxima;
    }

    public Celula getProxima() {
        return proxima;
    }

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

Essa classe `Celula` nunca será acessada diretamente por quem usar a classe `ListaLigada`, dessa forma estamos escondendo como a nossa classe funciona e garantindo que ninguém mexa na estrutura interna da nossa Lista.

A classe `ListaLigada` possui inicialmente apenas uma referência para a primeira [2] célula e outra para a última. Através da primeira conseguimos iterar até chegar em qualquer posição necessária. Já a referência à úlitma irá facilitar a inserção no fim da Lista.

```
public class ListaLigada {

    private Celula primeira;

    private Celula ultima;

}
```

5.3 - Definindo a interface

Com as operações definidas podemos esboçar a classe `ListaLigada`.

```
public class ListaLigada {

    private Celula primeira;

    private Celula ultima;

    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;}
}
```

Estamos discutindo sobre Lista Ligada especialmente porque queremos um tipo de Lista que as operações de adicionar e remover da primeira e da última posição sejam computacionalmente eficientes.

Então, vamos acrescentar mais três métodos na classe `ListaLigada`.

[2] alguns gostam de denominar essa célula de cabeça da lista

```
public void adicionaNoComeco(Object elemento) {}
public void removeDoComeco() {}
public void removeDoFim() {}
```

O método `adiciona()` insere no fim da Lista então não adicionaremos mais um método para realizar essa mesma tarefa.

5.4 - Testes

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

        lista.adiciona("Rafael");
        lista.adiciona("Paulo");

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

Saída esperada:

```
[Rafael, Paulo]
```

```
public class TesteAdicionaPorPosicao {
    public static void main(String[] args) {
        ListaLigada lista = new ListaLigada();
        lista.adiciona("Rafael");
        lista.adiciona(0, "Paulo");
        lista.adiciona(1, "Camila");

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

Saída esperada:

```
[Paulo, Camila, Rafael]
```

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

        lista.adiciona("Rafael");
        lista.adiciona("Paulo");

        System.out.println(lista.pega(0));
        System.out.println(lista.pega(1));
    }
}
```

Saída esperada:

```
Rafael

Paulo
```

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

        lista.adiciona("Rafael");
        lista.adiciona("Paulo");
        lista.adiciona("Camila");

        lista.remove(1);

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

Saída esperada:

```
[Rafael, Camila]
```

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

        lista.adiciona("Rafael");
        lista.adiciona("Paulo");

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

        lista.adiciona("Camila");

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

Saída esperada:

```
2

3
```

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

        lista.adiciona("Rafael");
        lista.adiciona("Paulo");

        System.out.println(lista.contem("Rafael"));
        System.out.println(lista.contem("Camila"));
    }
}
```

Saída esperada:

```
true

false
```

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

        lista.adicionaNoComeco("Rafael");
        lista.adicionaNoComeco("Paulo");

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

Saída esperada:

```
[Paulo, Rafael]
```

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

        lista.adiciona("Rafael");
        lista.adiciona("Paulo");

        lista.removeDoComeco();

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

Saída esperada:

```
[Paulo]
```

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

        lista.adiciona("Rafael");
        lista.adiciona("Paulo");

        lista.removeDoFim();

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

Saída esperada:

```
[Rafael]
```

5.5 - Operações sobre uma Lista

Aqui vamos começar a aumentar o código da nossa `ListaLigada` baseado nas operações que precisamos realizar com ela.

Para facilitar a contagem de elementos e algumas operações, vamos definir um atributo **int** `totalDeElementos`.

5.6 - Adicionando no começo da Lista

Inserir no começo da Lista é bastante trivial, basta criarmos uma nova célula, e esta nova célula terá a referência `proxima` apontando para a atual `primeira` da lista. Depois atualizamos o atributo `primeira` para se referenciar a esta nova célula recém criada.

Ainda falta tratar o caso especial da Lista estar vazia. Neste caso, devemos atualizar a referência que aponta para a última célula também. Através do atributo `totalDeAlunos` podemos identificar este caso.

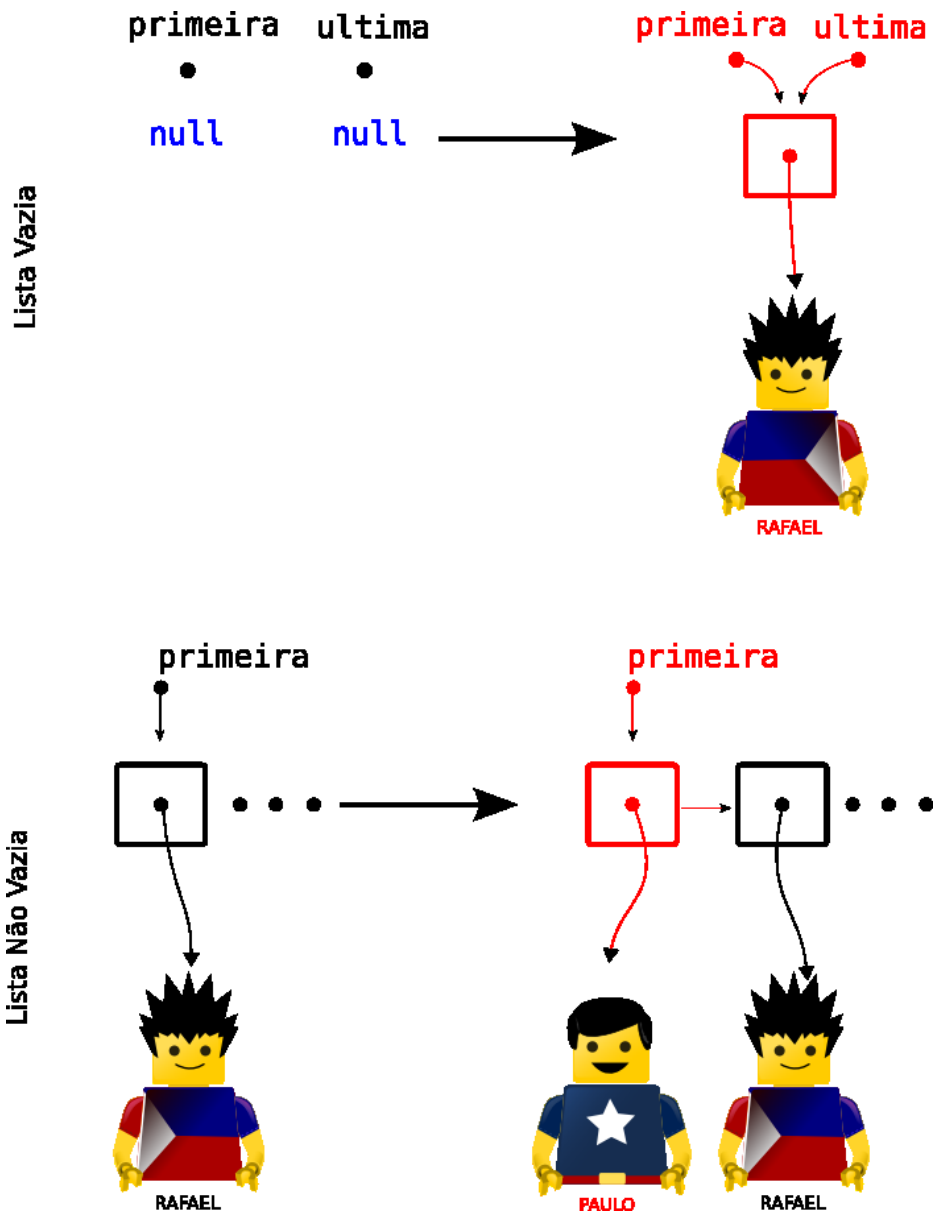


Fig.: Os dois casos para adicionar no começo

```
public class ListaLigada {

    private Celula primeira;

    private Celula ultima;

    private int totalDeElementos;

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

        if(this.totalDeElementos == 0){
            // caso especial da lista vazia
```

```
            this.ultima = this.primeira;
        }
        this.totalDeElementos++;
    }
}
```

5.7 - Adicionando no fim da Lista

Se não tivessemos guardado a referência para a última célula precisaríamos percorrer célula a célula até o fim da Lista para alterar a referência `proxima` da última célula! Com um grande número de elementos isso ficaria lento, pois leva tempo linear.

No caso especial da Lista estar vazia, adicionar no começo ou no fim dessa lista dá o mesmo efeito. Então, se a Lista estiver vazia, chamaremos o método `adicionaNoComeco(Object)`.

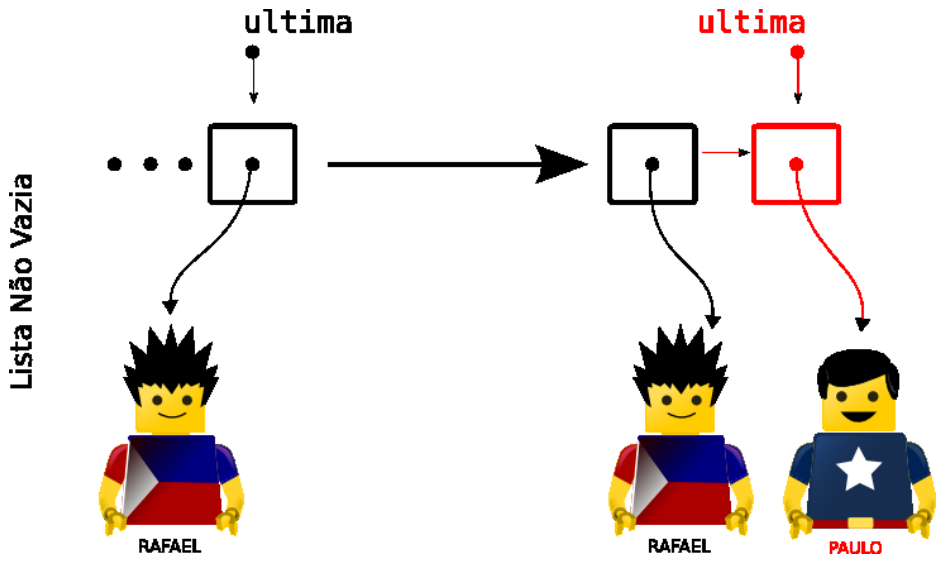


Fig.: Adicionando no fim

```
public class ListaLigada {

    private Celula primeira;

    private Celula ultima;

    private int totalDeElementos;

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

```
    }
}
```

Repare que aqui estamos fazendo que haja `n` células para `n` elementos, isto é, uma célula para cada elemento. Uma outra implementação clássica de lista ligada é usar uma célula **sentinela** a mais para indicar o começo da Lista, e outra para o fim, assim poderíamos simplificar um pouco alguns desses métodos, como ocaso particular de inserir um elemento quando não há ainda elemento algum. Vale sempre lembrar que aqui estamos estudando uma implementação de uma estrutura de dados, e que há sempre outras formas de codificalas, que podem ser mais ou menos elegantes.

5.8 - Percorrendo nossa Lista

Para poder rodar alguns testes, precisamos imprimir o conteúdo da nossa Lista. Como ainda não temos como acessar um elemento pela sua posição, vamos reescrever o método `toString()` da classe `ListaLigada` para que ele concatene todos os elementos de dentro da Lista Ligada em uma única `String`.

O nosso método utiliza-se de uma referência temporátia para uma célula chamada atual, que vai apontando para a próxima a cada iteração, concatenando o elemento dessa célula:

```
public String toString() {

    // Verificando se a Lista está vazia
    if(this.totalDeElementos == 0){
        return "[]";
    }

    StringBuilder builder = new StringBuilder("[");
    Celula atual = primeira;

    // Percorrendo até o penúltimo elemento.
    for (int i = 0; i < this.totalDeAlunos - 1; i++) {
        builder.append(atual.getElemento());
        builder.append(", ");
        atual = atual.getProxima();
    }

    // último elemento
    builder.append(atual.getElemento());
    builder.append("]");

    return builder.toString();
}
```

Aqui estamos utilizando a classe `StringBuilder` que é muito útil para criar Strings potencialmente grandes, em vez de concatenar Strings pelo operador +[label o fato da classe String do java ser imutável acarreta num grande gasto de memória temporária no caso de você concatenas muitas Strings, vale a pena consultar a documentação da mesma][/label].

Também poderíamos ter utilizado um `while(atual != null)` em vez do `for` dentro do método `toString`, que as vezes pode ser mais legível.

5.9 - Adicionando em qualquer posição da Lista

A inserção no começo e no fim da Lista já foram tratadas nas seções anteriores. Aqui vamos nos preocupar em inserir em uma posição do interior da Lista (qualquer posição que não é a primeira e nem a última).

Para inserir um elemento em qualquer posição precisamos pegar a célula anterior a da posição desejada, porque precisamos mexer na sua referência `proxima`. Para isso vamos criar um método auxiliar que pega determinada célula. Este método deve tomar cuidado com o caso da posição não existir. A verificação se a posição existe ou não é feita pelo método `posicaoOcupada(int)`.

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

private Celula pegaCelula(int posicao) {
    if(!this.posicaoOcupada(posicao)){
        throw new IllegalArgumentException("Posição não existe");
    }

    Celula atual = primeira;
    for (int i = 0; i < posicao; i++) {
        atual = atual.getProxima();
    }
    return atual;
}
```

Os métodos são privados pois não queremos que ninguém de fora tenha acesso ao funcionamento interno da nossa estrutura de dados. É importante notar que o método `pegaCelula(int)` consome tempo linear!

Agora o método `adiciona(int, Object)` fica simples de ser feito. Basta pegar a célula anterior a posição onde a inserção será feita e atualizar as referências. A anterior deve apontar para uma nova célula e a nova célula deve apontar para a antiga próxima da anterior.

Devemos tomar cuidado com os casos particulares nos quais a posição para inserir é o começo ou o fim da Lista.

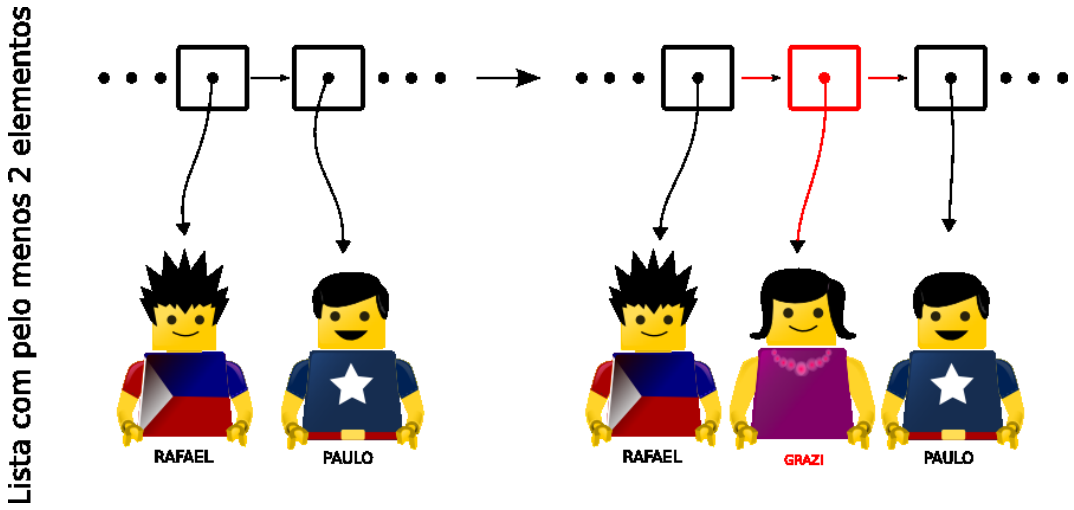


Fig.: Adicionando no interior da Lista

```
public void adiciona(int posicao, Object elemento) {
    if(posicao == 0){ // No começo.
        this.adicionaNoComeco(elemento);
    } else if(posicao == this.totalDeElementos){ // No fim.
        this.adiciona(elemento);
    } else {
        Celula anterior = this.pegaCelula(posicao - 1);
        Celula nova = new Celula(anterior.getProxima(), elemento);
        anterior.setProxima(nova);
        this.totalDeElementos++;
    }
}
```

5.10 - Pegando um elemento da Lista

Para pegar um elemento é muito fácil: basta pegarmos a célula em que aquele elemento se encontra e acessar o elemento de dentro dela. Podemos utilizar o `pegaCelula(int)` previamente criado:

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

Perceba que este método consome tempo linear. Esta é uma **grande** desvantagem da Lista Ligada em relação aos Vetores. Vetores possuem o chamado **acesso aleatório** aos elementos: qualquer posição pode ser acessada em tempo constante. Apesar dessa grande desvantagem, diversas vezes utilizamos uma Lista e não é necessário ficar acessando posições aleatórias: comumente percorremos a lista por completa, que veremos como fazer mais adiante.

5.11 - Removendo do começo da Lista

Antes de tentar remover devemos verificar se a posição está ocupada. Não faz sentido remover algo que não existe. Depois, basta "avançar" a referência que aponta para a primeira célula.

Por fim, é importante perceber que a Lista pode ficar vazia. Neste caso, devemos colocar **null** na referência que aponta para a última célula.

Se não fizermos isso ficaríamos em um estado inconsistente, em que o atributo `primeira` é `null` e o `última` não, ou seja, tem uma última mas não tem uma primeira. Isso não faria sentido.

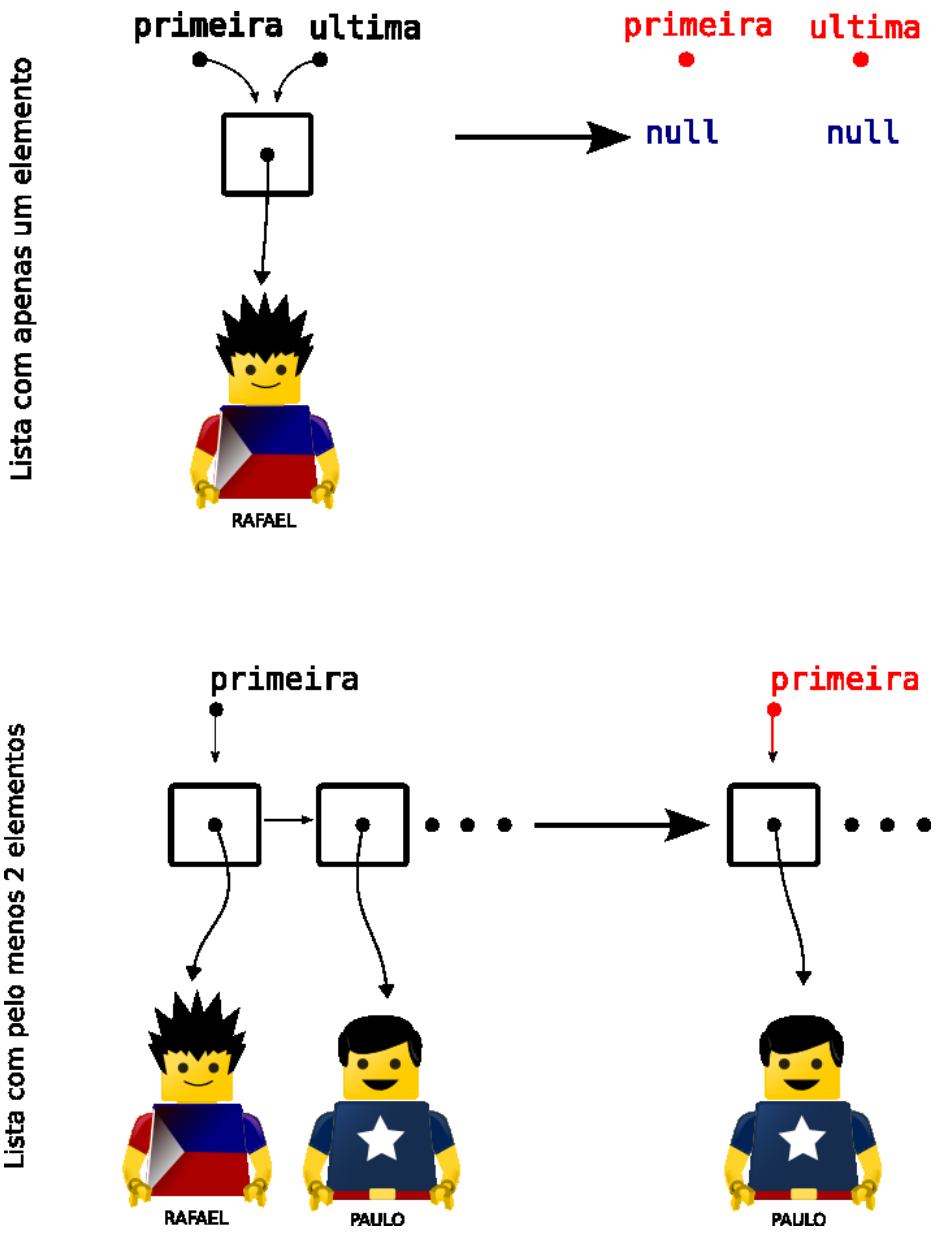


Fig.: Os dois casos de remover do começo

```
public void removeDoComeco() {
    if (!this.posicaoOcupada(0)) {
        throw new IllegalArgumentException("Posição não existe");
    }

    this.primeira = this.primeira.getProxima();
    this.totalDeElementos-;

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

5.12 - Removendo do fim da Lista

A primeira verificação a ser feita é se a última posição existe. Podemos fazer isso através do método já criado `posicaoOcupada(int)`.

Se a Lista estiver com apenas um elemento então remover do fim é a mesma coisa que remover do começo. Logo, podemos reutilizar o método `removeDoComeco()` para este caso.

Agora, se a Lista tem mais que um elemento então devemos pegar a penúltima célula; fazer a próxima da penúltima ser **null**; e fazer o atributo `ultima` apontar para a penúltima.

O problema aqui é como pegar a penúltima célula. Podemos fazer isso usando o `pegaCelula(int)` mas isso consumiria tempo linear. Como queremos consumo constate teremos que achar outra solução.

Então, em vez de fazer uma Lista Ligada simples vamos fazer uma Lista Duplamente Ligada. Ou seja, cada célula aponta para a sua anterior além de apontar para a próxima:

```
public class Celula {

    ...

    private Celula anterior;

    ...

    public Celula getAnterior() {
        return anterior;
    }

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

Com cada célula sabendo também quem é a sua anterior, fica fácil escrever o método `removeDoFim()`

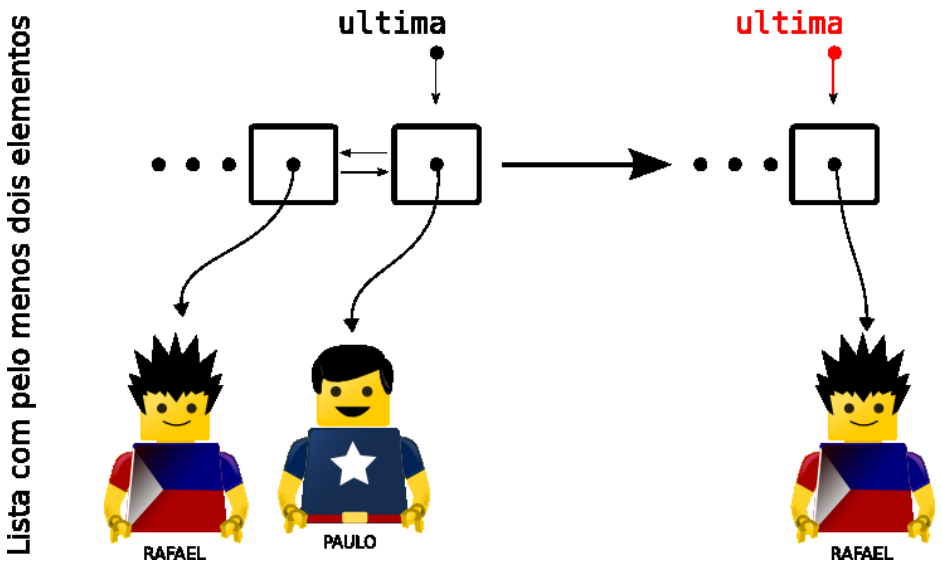


Fig.: Removendo do fim da Lista

```
public void removeDoFim() {
    if (!this.posicaoOcupada(this.totalDeElementos - 1)) {
        throw new IllegalArgumentException("Posição não existe");
    }
    if (this.totalDeElementos == 1) {
        this.removeDoComeco();
    } else {
        Celula penultima = this.ultima.getAnterior();
        penultima.setProxima(null);
        this.ultima = penultima;
        this.totalDeElementos-;
    }
}
```

A modificação para Lista Duplamente Ligada implicará em pequenas modificações nos outros métodos que já tinhamos implementado. As modificações serão apresentadas na seção de Lista Duplamente Ligada.

5.13 - Removendo de qualquer posição

Inicialmente, devemos verificar se a posição está ou não ocupada. Se não estiver devemos lançar uma exceção. Caso contrário, devemos verificar se a remoção é do começo ou do fim da Lista se for um destes casos simplesmente chamamos os métodos que já fizemos.

Por fim, se a remoção é no interior da Lista devemos atualizar as referências das células relacionadas a célula que vamos remover (anterior e próxima). A próxima da anterior deve ser a proxima e a anterior da proxima deve ser a anterior.

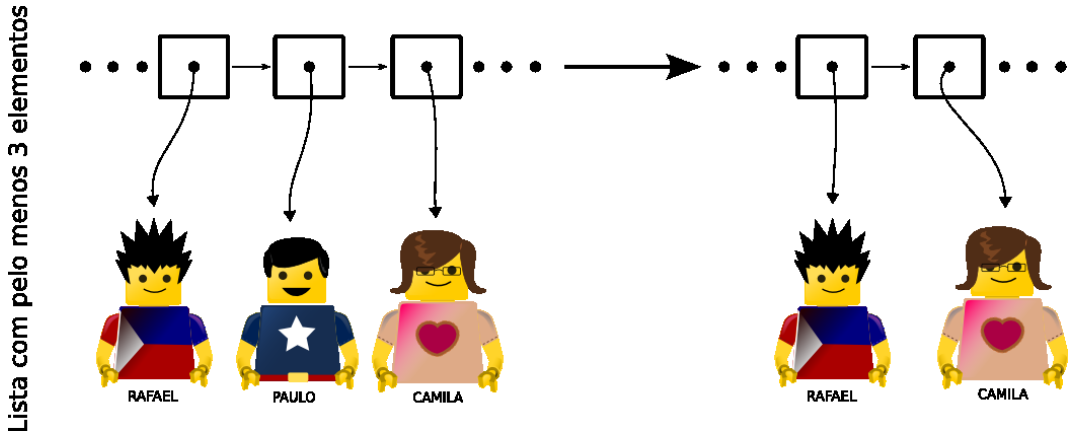


Fig.: Removendo do interior da Lista

```
public void remove(int posicao) {
    if (!this.posicaoOcupada(posicao)) {
        throw new IllegalArgumentException("Posição não existe");
    }

    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-;
    }
}
```

5.14 - Verificando se um elemento está na Lista

Está operação deve percorrer a Lista e comparar com o método `equals(Object)` o elemento procurado contra todos os elementos da Lista.

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

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

5.15 - O tamanho da Lista

Está operação não tem segredo, pois já temos um atributo que possui esta informação.

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

5.16 - Lista Duplamente Ligada

Para o método que remove o elemento do fim da Lista ter consumo de tempo constante, a última célula deve se referir a penúltima. Isso é possível se utilizarmos o esquema de Lista Duplamente Ligada, no qual as células possuem referências tanto para a próxima e quanto para a célula anterior.

Está modificação implica alterações na implementação dos outros métodos, como já falamos anteriormente. Seguem:

5.17 - Adicionando no começo da Lista

Temos que considerar dois casos: Lista Vazia e Lista não Vazia.

Se a Lista está vazia então a nova celula será a primeira e a última. Além disso, ela não terá próxima nem anterior pois ela será a única célula.

Se a Lista não está vazia então devemos ajustar os ponteiros para a nova segunda (antiga referência `primeira`) apontar para a nova `primeira` e vice versa.

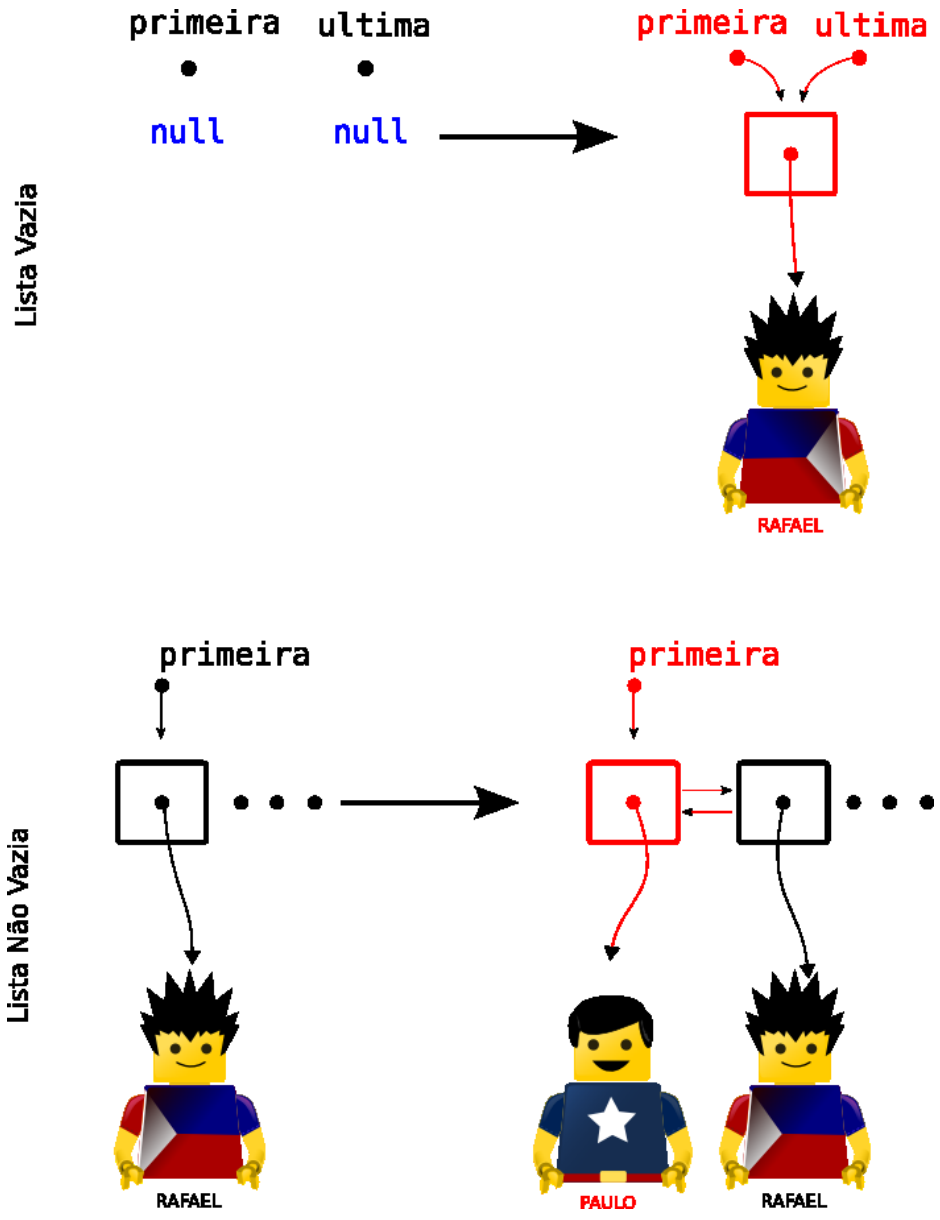


Fig.: Adicionado no começo da Lista Duplamente Ligada

```
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++;
}
```

5.18 - Adicionando no fim da Lista

No caso em que a Lista está vazia, adicionar no fim é a mesma coisa que adicionar no começo.

Agora, caso a Lista não esteja vazia então devemos ajustar as referências de tal forma que a nova última célula aponte para a nova penúltima (antiga última) e vice versa.

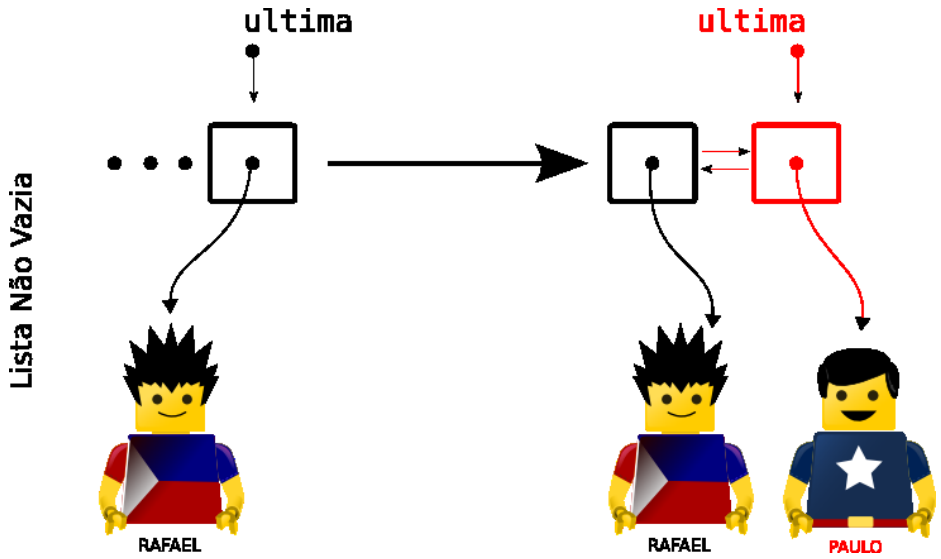


Fig.: Adicionado no fim da Lista Duplamente Ligada

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

5.19 - Adicionando em qualquer posição da Lista

Separamos os casos em que a inserção é no começo ou no fim porque podemos reaproveitar os métodos já implementados.

Sobra o caso em que a inserção é no meio da Lista, ou seja, entre duas células existentes. Neste caso, devemos ajustar as referências para a nova célula ser apontada corretamente pela duas células relacionadas a ela (a anterior e a proxima). E também fazer a nova célula apontar para a anterior e a próxima.

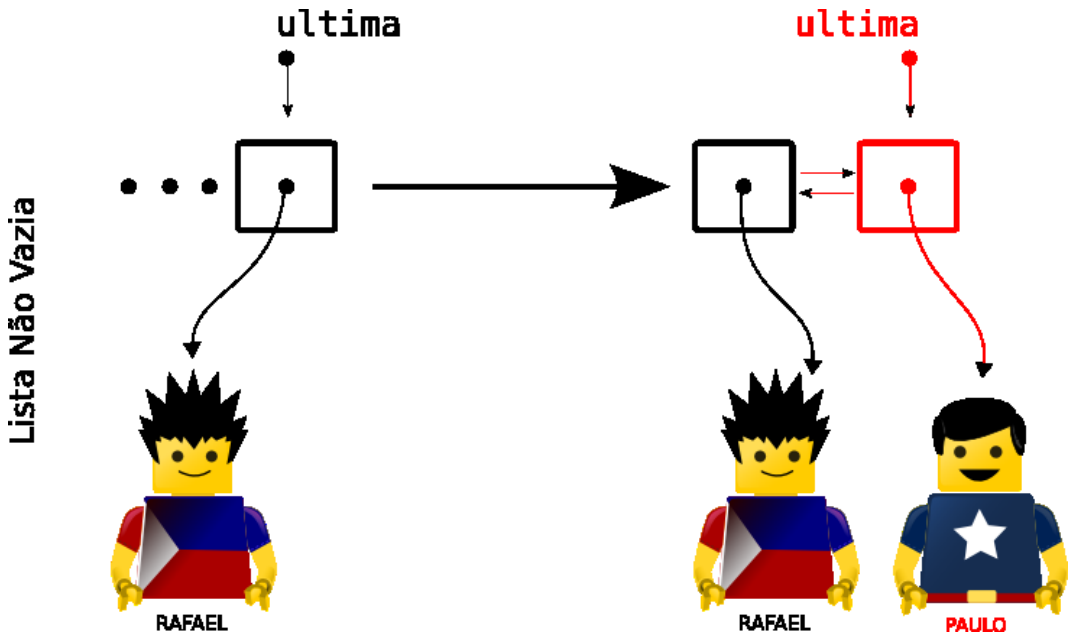


Fig.: Adicionado no fim da Lista Duplamente Ligada

```
public void adiciona(int posicao, Object elemento) {
    if(posicao == 0){ // No começo.
        this.adicionaNoComeco(elemento);
    } else if(posicao == this.totalDeElementos){ // No fim.
        this.adiciona(elemento);
    } else {
        Celula anterior = this.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++;
    }
}
```

5.20 - Removendo do começo da Lista

Esta operação é idêntica em ambos os tipos de Lista Ligada (simple ou dupla). Ela apenas deve avançar a referência primeira para a segunda célula e tomar cuidado com a caso da Lista ficar vazia pois, neste caso, a

referência ultima deve ser atualizada também.

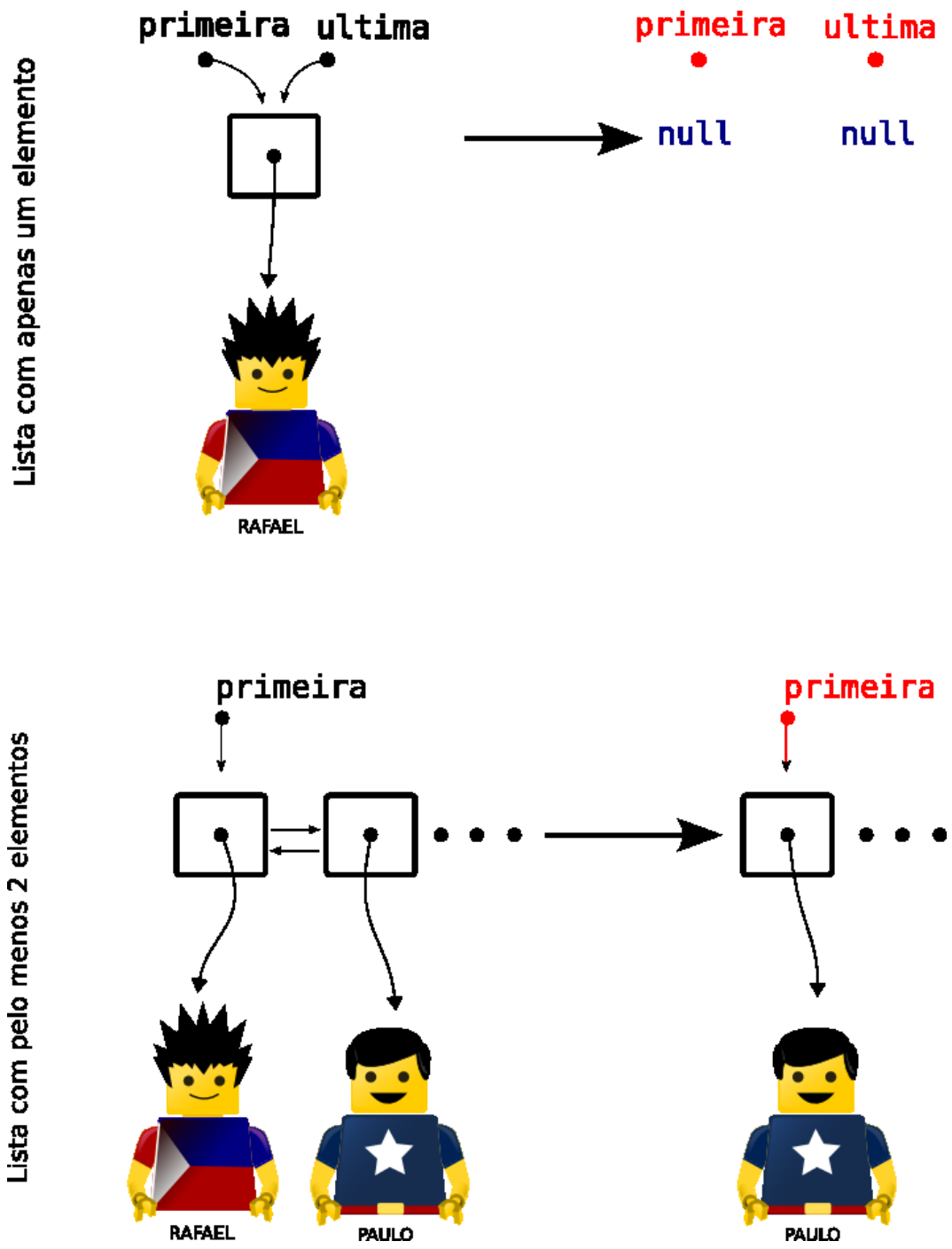


Fig.: Removendo do começo da Lista Duplamente Ligada

5.21 - Removendo do fim da Lista ou de qualquer posição

Estas duas operações já foram tratadas e implementadas anteriormente usando o esquema de Lista Duplamente Ligada.

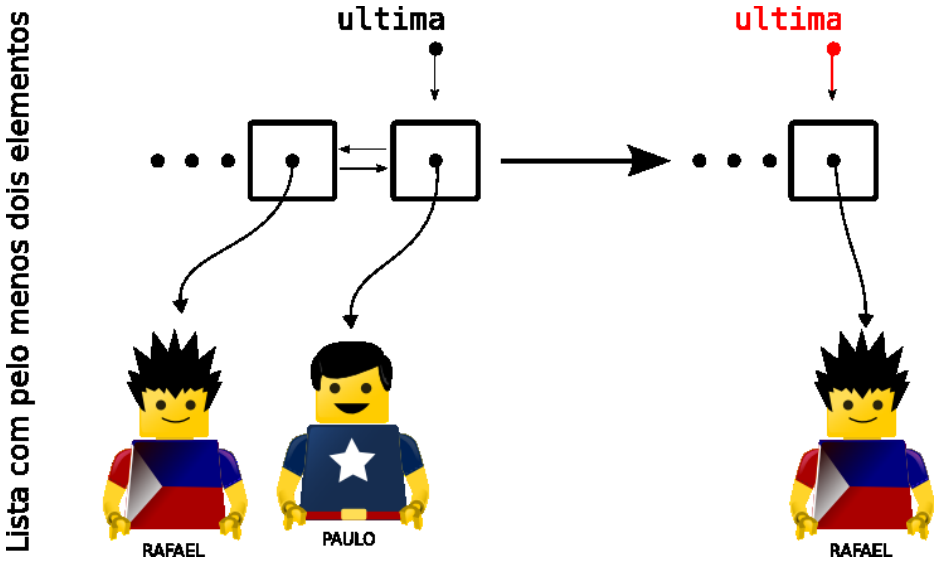


Fig.: Removendo do fim da Lista Duplamente Ligada

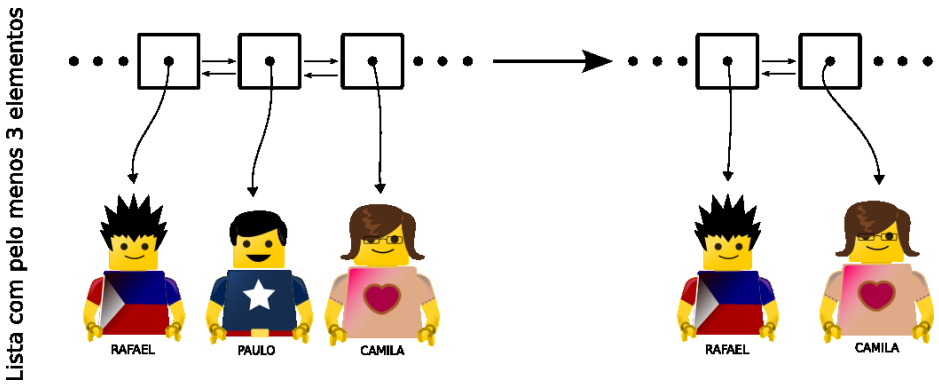


Fig.: Removendo no interior da Lista Duplamente Ligada

5.22 - API

A classe `LinkedList` faz o papel da nossa Lista Ligada dentro da bibilioteca do Java. Ela possui os mesmos métodos que a `ArrayList`, e adiciona alguns outros, como o `addFirst(Object)`, `removeFirst()`, `addLast(Object)` e `removeLast()`, que operam no começo e no fim da Lista em tempo constante.

```
public class Teste {

    public static void main(String[] args) {
        Aluno paulo = new Aluno();
        paulo.setNome("paulo silveira");

        Aluno rafael = new Aluno();
        rafael.setNome("rafael cosentino");

        Aluno fiodor = new Aluno();
        fiodor.setNome("fiodor dostoievski");
```

```
        LinkedList listaLigada = new LinkedList();

        listaLigada.add(paulo);
        listaLigada.add(rafael);
        listaLigada.add(1, fiodor);

        for (int i = 0; i < listaLigada.size(); i++) {
            System.out.println(listaLigada.get(i));
        }
    }
}
```

5.23 - Exercícios: Lista Ligada

1) Implemente a estrutura Célula através de uma classe no pacote **br.com.caelum.ed.listasligadas**.

```
package br.com.caelum.ed.listasligadas;

public class Celula {
    private Celula proxima;

    private Celula anterior;

    private Object elemento;

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

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

    public void setProxima(Celula proxima) {
        this.proxima = proxima;
    }

    public Celula getProxima() {
        return proxima;
    }

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

    public Celula getAnterior() {
        return anterior;
    }

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

```
    }
}
```

2) Implemente a classe `ListaLigada` com o "esqueleto" das operações. Utilize o pacote **br.com.caelum.ed.listasligadas**.

```
package br.com.caelum.ed.listasligadas;

public class ListaLigada {

    private Celula primeira;

    private Celula ultima;

    private int totalDeElementos;

    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;
    }

    public void adicionaNoComeco(Object elemento) {
    }

    public void removeDoComeco() {
    }

    public void removeDoFim() {
    }
}
```

3) Implemente **todos** os testes feito na seção Teste. Coloque as classes no pacote **br.com.caelum.ed.listasligadas**.

4) Para testar as operações precisamos reescrever o método `toString()` para devolver os elementos da Lista em uma String.

```
public String toString() {
```

```
        // Verificando se a Lista está vazia
        if(this.totalDeElementos == 0){
            return "[]";
        }

        StringBuilder builder = new StringBuilder("[");
        Celula atual = primeira;

        // Percorrendo até o penúltimo elemento.
        for (int i = 0; i < this.totalDeElementos - 1; i++) {
            builder.append(atual.getElemento());
            builder.append(", ");
            atual = atual.getProxima();
        }

        // último elemento
        builder.append(atual.getElemento());
        builder.append("]");

        return builder.toString();
    }
```

5) Implemente todos métodos para a Lista Ligada. A cada método implementado não esqueça de executar o teste apropriado.

- Adiciona no começo.

```
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++;
}
```

- Adiciona no fim.

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

- Adiciona em qualquer posição. Esta operação depende do método `pegaCelula(int)` que por sua vez depende do método `posicaoOcupada(int)`.

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

private Celula pegaCelula(int posicao) {
    if(!this.posicaoOcupada(posicao)){
        throw new IllegalArgumentException("Posição não existe");
    }

    Celula atual = primeira;
    for (int i = 0; i < posicao; i++) {
        atual = atual.getProxima();
    }
    return atual;
}

public void adiciona(int posicao, Object elemento) {
    if(posicao == 0){ // No começo.
        this.adicionaNoComeco(elemento);
    } else if(posicao == this.totalDeElementos){ // No fim.
        this.adiciona(elemento);
    } else {
        Celula anterior = this.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++;
    }
}
```

- Remove do começo.

```
public void removeDoComeco() {
    if (!this.posicaoOcupada(0)) {
        throw new IllegalArgumentException("Posição não existe");
    }

    this.primeira = this.primeira.getProxima();
    this.totalDeElementos-;

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

- Remove do fim.

```
public void removeDoFim() {
    if (!this.posicaoOcupada(this.totalDeElementos - 1)) {
        throw new IllegalArgumentException("Posição não existe");
```

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

- Remove de qualquer posição.

```
    public void remove(int posicao) {
        if (!this.posicaoOcupada(posicao)) {
            throw new IllegalArgumentException("Posição não existe");
        }

        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-;
        }
    }
```

- Informar o tamanho.

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

- Pega elemento por posição.

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

- Verifica se um elemento pertence a Lista.

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

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

```
            atual = atual.getProxima();
        }
        return false;
    }
```

6) Vamos comparar o consumo de tempo entre o Vetor e a Lista Ligada para saber em quais situações um é melhor que o outro.

```
public class TestePerformance {
    public static void main(String[] args) {

        ArrayList<String> vetor = new ArrayList<String>();
        LinkedList<String> lista = new LinkedList<String>();
        int numeroDeElementos = 40000;

        // ADICIONADO NO COMEÇO

        long inicio = System.currentTimeMillis();

        for (int i = 0; i < numeroDeElementos; i++) {
            vetor.add(0, "" + i);
        }

        long fim = System.currentTimeMillis();
        System.out.println("Vetor adiciona no começo: " + (fim - inicio)
                / 1000.0);

        inicio = System.currentTimeMillis();

        for (int i = 0; i < numeroDeElementos; i++) {
            lista.add(0, "" + i);
        }

        fim = System.currentTimeMillis();
        System.out.println("Lista Ligada adiciona no começo: " + (fim - inicio)
                / 1000.0);


        // PERCORRENDO
        inicio = System.currentTimeMillis();

        for (int i = 0; i < numeroDeElementos; i++) {
            vetor.get(i);
        }

        fim = System.currentTimeMillis();
        System.out
                .println("Vetor percorrendo: " + (fim - inicio) / 1000.0);

        inicio = System.currentTimeMillis();

        for (int i = 0; i < numeroDeElementos; i++) {
            lista.get(i);
        }
```

```
        fim = System.currentTimeMillis();
        System.out.println("Lista Ligada percorrendo: " + (fim - inicio)
                / 1000.0);


        // REMOVENDO DO COMEÇO
        inicio = System.currentTimeMillis();

        for (int i = 0; i < numeroDeElementos; i++) {
            vetor.remove(0);
        }

        fim = System.currentTimeMillis();
        System.out
                .println("Vetor remove do começo: " + (fim - inicio) / 1000.0);

        inicio = System.currentTimeMillis();

        for (int i = 0; i < numeroDeElementos; i++) {
            lista.remove(0);
        }

        fim = System.currentTimeMillis();
        System.out.println("Lista Ligada remove do começo: " + (fim - inicio)
                / 1000.0);

    }
}
```

CAPÍTULO 6

Pilhas

"O pessimista queixa-se do vento, o otimista espera que ele mude e o realista ajusta as velas. "
– Willian George Ward

6.1 - Introdução

Um determinado produto é composto por diversas peças (digamos `p1, p2, ...,pn`). O processo de montagem deste produto é automático (executado por uma máquina) e exige que as peças sejam colocadas em uma ordem específica (primeiro a `p1`, depois a `p2`, depois a `p3` e assim por diante). As peças são empilhadas na ordem adequada e a máquina de montagem vai retirando peça por peça do topo desta pilha para poder montar o produto final.

A mesma máquina que faz a montagem é capaz de trocar uma peça quebrada de um produto já montado. O que a máquina faz é desmontar o produto até chegar na peça defeituosa, trocá-la e então depois recolocar as peças que foram retiradas. Isso também é feito com o uso da pilha de peças. Veja a seguir o algoritmo que a máquina montadora implementa para fazer a manutenção de um produto com defeito.

1) Retirar e empilhar peça por peça do produto até chegar na peça defeituosa.

2) Retirar a peça defeituosa

3) Colocar uma peça nova sem defeitos

4) Desempilhar e montar peça por peça do topo da pilha até a pilha ficar vazia.

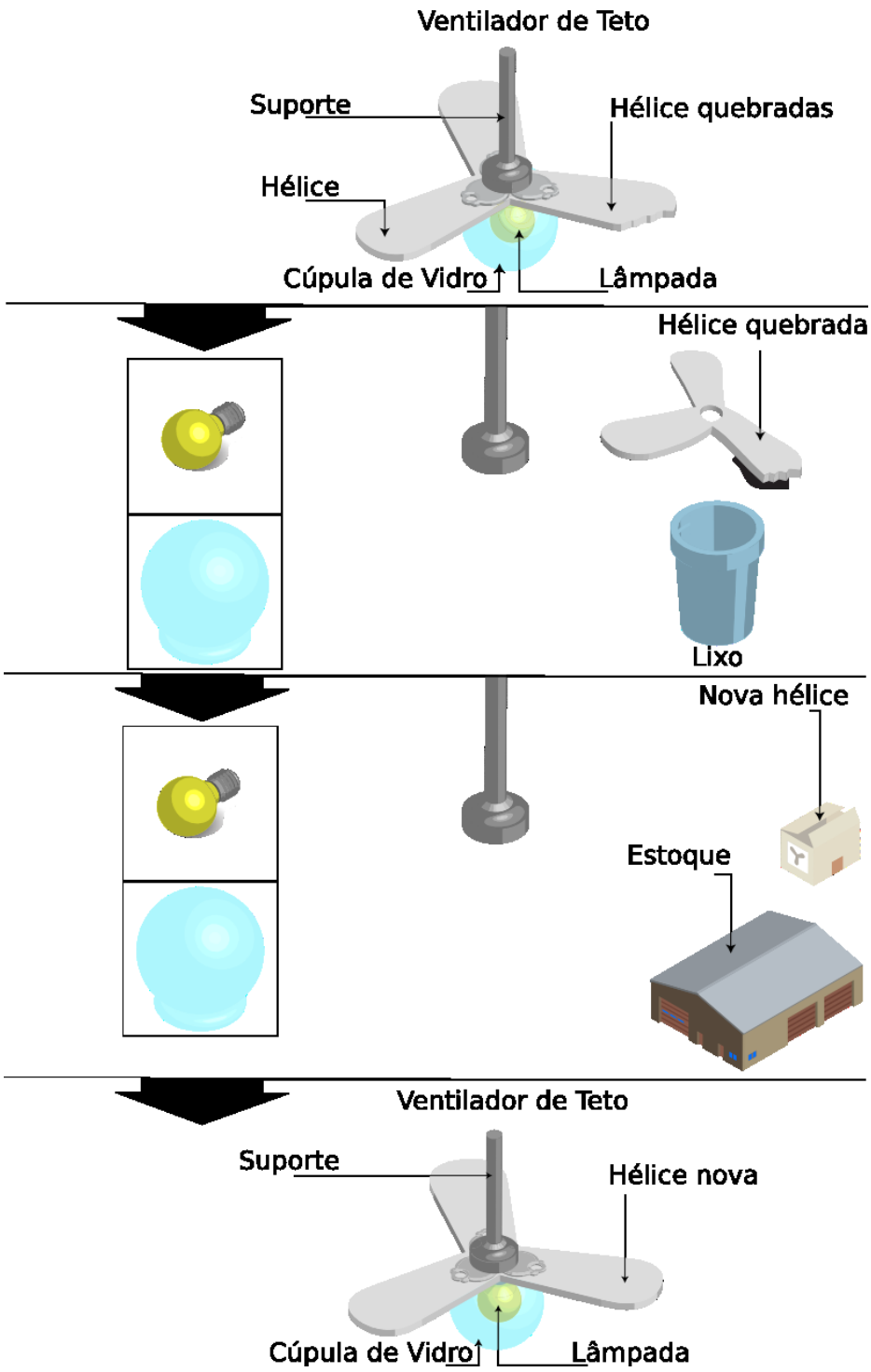


Fig.: Consertando o ventilador

De alguma forma uma peça precisa ser representada em nosso programa. Como estamos usando orientação a objetos as peças serão representadas por objetos. Uma classe Java será criada somente para modelar as peças, algo similar ao código a seguir:

```
public class Peca {
```

```
    private String nome;

    public String getNome() {
        return nome;
    }

    public void setNome(String nome) {
        this.nome = nome;
    }
}
```

Com a classe `Peca`, já é possível criar objetos para representar as peças que a máquina montadora utiliza. Porém, o sistema deve definir como guardar estes objetos, ou seja, ele precisa escolher uma estrutura de dados. Esta estrutura de dados deve manter os dados seguindo alguma lógica e deve fornecer algumas operações para a manipulação destes e outras operações para informar sobre seu próprio estado.

6.2 - Solução do problemas das Peças

Para implementar o algoritmo de manutenção do carro, é necessário criar uma estrutura de dados que se comporte como a pilha de peças. Vamos chamar esta estrutura de dados de **Pilha**.

Primeiro, definimos a interface da Pilha (conjunto de operações queremos utilizar em uma Pilha).

1) Insere uma peça (coloca uma peça no topo da Pilha).

2) Remove uma peça (retira a peça que está no topo da Pilha).

3) Informa se a Pilha está vazia.

Podemos criar uma classe `Pilha` para implementar a esta estrutura de dados. Os métodos públicos desta classe serão a implementação das operações.

```
public class Pilha {

    public void insere(Peca peca) {
        // implementação
    }

    public Peca remove() {
        // implementação
    }

    public boolean vazia() {
        // implementação
    }
}
```

O primeiro fato importante que devemos observar é que uma vez que a interface da Pilha foi definida, já saberiamos usar a classe `Pilha`. Vamos criar uma classe de teste bem simples para exemplificar o uso de uma Pilha.

```
public class Teste {
```

```
    public static void main(String[] args) {
        Pilha pilha = new Pilha();

        Peca pecaInsere = new Peca();
        pilha.insere(pecaInsere);

        Peca pecaRemove = pilha.remove();

        if (pilha.vazia()) {
            System.out.println("A pilha está vazia");
        }
    }
}
```

O segundo fato importante é que a estrutura que queremos aqui é muito similar as Listas que vimos anteriormente. A semelhança fundamental entre as Listas e as Pilhas é que ambas devem armazenar os elementos de maneira sequencial. Este fato é o ponto chave deste capítulo.

Qual é a diferença entre uma Lista e uma Pilha? A diferença está nas operações destas duas estruturas de dados. As operações de uma Pilha são mais **restritas** do que as de uma Lista. Por exemplo, você pode adicionar ou remover um elemento em qualquer posição de uma Lista mas em uma Pilha você só pode adicionar ou remover do topo.

Então, uma Lista é uma estrutura mais poderosa emais genérica do que uma Pilha. A Pilha possui apenas um subconjunto de operações da Lista. Então o interessante é que para implementar uma Pilha podemos usar uma Lista. Isso mesmo! Vamos criar restrições sobre as operações da Lista e obteremos uma Pilha.

Nós implementamos dois tipos de Listas: Vetores e Listas Ligadas. Vimos, também que, na biblioteca do Java, há implementações prontas para estes dois tipos de Listas. Neste capítulo, vamos utilizar a classe `LinkedList` para armazenar as peças que serão guardadas Pilha.

```
public class Pilha {

    private List<Peca> pecas = new LinkedList<Peca>();

}
```

Dentro de nossa `Pilha` teremos uma `LinkedList` encapsulada, que vai simplificar bastante o nosso trabalho: delegaremos uma série de operações para essa Lista Ligada, porém sempre pensando nas diferenças essenciais entre uma Pilha e uma Lista.

Devemos ter um `getPecas()` que devolve uma referência para essa nossa `LinkedList`? Nesse caso a resposta é não, pois estaríamos expondo detalhes de nossa implementação, e o usuário dessa classe poderia mexer no funcionamento interno da nossa pilha, desrespeitando as regras de sua interface. É sempre uma boa prática expor o mínimo possível do funcionamento interno de uma classe, gera um menor acoplamento entre as classes.

6.3 - Operações em pilhas

Agora, vamos implementar as operações da Pilha.

6.4 - Inserir uma peça

As peças são sempre inseridas no topo da Pilha. Ainda não definimos onde fica o topo da Pilha. Como estamos utilizando uma Lista para guardar os elementos então o topo da Pilha poderia ser tanto o começo ou o fim da Lista. Aqui escolheremos o fim da Lista.

Então, inserir na Pilha é simplesmente adicionar no fim da Lista.

```
public class Pilha {

    private List<Peca> pecas = new LinkedList<Peca>();

    public void insere(Peca peca) {
        this.pecas.add(peca);
    }
}
```

Recordando que o método `add(Object)` adiciona no fim da Lista.

6.5 - Remover uma peça

A remoção também é bem simples, basta retirar o último elemento da Lista.

```
public class Pilha {

    private List<Peca> pecas = new LinkedList<Peca>();

    ...

    public Peca remove() {
        return this.pecas.remove(this.pecas.size() - 1);
    }
}
```

É bom observar que se o método `remove()` for usado com a Pilha vazia então uma exceção será lançada pois o método `remove(int)` da `List` lança `IndexOutOfBoundsException` quando não existir elemento para remover.

6.6 - Informar se a pilha está vazia

Para implementar esta operação basta verificar se o tamanho da Lista é zero.

```
public class Pilha {

    private List<Peca> pecas = new LinkedList<Peca>();

    ...
    public boolean vazia() {
        return this.pecas.size() == 0;
    }
}
```

Na classe `LinkedList` existe também o método `isEmpty()` que poderia ter sido usado aqui mais convenientemente.

6.7 - Generalização

Nossa Pilha só funciona para guardar objetos da classe `Peca`. Vamos generalizar a Pilha para poder armazenar qualquer tipo de objeto. Isso será feito utilizando a classe `Object` da qual todas as classe derivam direta ou indiretamente. Criaremos uma `LinkedList` de `Object` em vez de uma `LinkedList` de `Peca`.

```
public class Pilha {

    private List<Object> objetos = new LinkedList<Object>();

    public void insere(Object objeto) {
        this.objetos.add(objeto);
    }

    public Object remove() {
        return this.objetos.remove(this.objetos.size() - 1);
    }

    public boolean vazia() {
        return this.objetos.size() == 0;
    }
}
```

Agora, podemos guardar qualquer tipo de objeto na Pilha. Isso é uma grande vantagem pois a classe `Pilha` poderá ser reaproveitada em diversas ocasiões. Mas, há uma desvantagem, quando removemos um elemento da Pilha não podemos garantir qual é o tipo de objeto que virá.

No Java 5, poderíamos usar Generics para solucionar este problema. A nossa classe `Pilha` poderia ser uma classe parametrizada. Na criação de um objeto de uma classe parametrizada é possível dizer com qual tipo de objeto que queremos trabalhar.

Algo deste tipo:

```
Pilha de Peças pilha = new Pilha de Peças();
```

Traduzindo este código para Java 5 de verdade, ficaria assim:

```
Pilha<Peca> pilha = new Pilha<Peca>();
```

Só que para utilizar o recurso do Generics devemos parametrizar a class `Pilha`.

```
public class Pilha<T> {

    private LinkedList<T> objetos = new LinkedList<T>();

    public void insere(T t) {
        this.objetos.add(t);
    }

    public T remove() {
```

```
        return this.objetos.remove(this.objetos.size() - 1);
    }

    public boolean vazia() {
        return this.objetos.size() == 0;
    }
}
```

Poderíamos também adicionar outros método provavelmente úteis a manipulação de uma pilha, como saber o tamanho da pilha, espiar um elemento em determinada posição, entre outros.

Vamos testar a classe `Pilha` que usa Generics.

```
public class Teste {

    public static void main(String[] args) {
        Pilha<Peca> pilha = new Pilha<Peca>();

        Peca peca = new Peca();
        pilha.insere(peca);

        Peca pecaRemove = pilha.remove();

        if (pilha.vazia()) {
            System.out.println("A pilha está vazia");
        }

        Pilha<String> pilha2 = new Pilha<String>();
        pilha2.insere("Rafael Cosentino");
        pilha2.insere("Paulo Silvera");

        String paulo = pilha2.remove();
        String rafael = pilha2.remove();

        System.out.println(paulo);
        System.out.println(rafael);
    }
}
```

Neste exemplo, criamos duas Pilhas. A primeira vai armazenar só objetos do tipo `Peca` e a segunda só `String`. Se você tentar adicionar um tipo de objeto que não corresponde ao que as Pilhas estão guardando então um erro de compilação será gerado para evitar que o programador cometa um erro lógico.

6.8 - API do Java

Na biblioteca do Java existe uma classe que implementa a estrutura de dados que foi vista neste capítulo, esta classe chama-se `Stack` e será testada pelo código abaixo.

```
public class Teste {
    public static void main(String[] args) {

        Stack pilha = new Stack();
```

```
        Peca pecaInsere = new Peca();
        pilha.push(pecaInsere);

        Peca pecaRemove = (Peca)pilha.pop();

        if(pilha.isEmpty()){
            System.out.println("A pilha está vazia");
        }
    }
}
```

Para evitar fazer casting de objetos, podemos utilizar o recurso de Generics aqui também.

```
public class Teste {
    public static void main(String[] args) {

        Stack<Peca> pilha = new Stack<Peca>();

        Peca pecaInsere = new Peca();
        pilha.push(pecaInsere);

        Peca pecaRemove = pilha.pop();

        if (pilha.isEmpty()) {
            System.out.println("A pilha está vazia");
        }
    }
}
```

6.9 - Escapando do Labirinto

Ao percorrer um labirinto para fugir do Minotauro, existe um algoritmo bastante simples: você sempre escolher o caminho mais a direita em toda oportunidade que aparecer a separação do caminho atual em dois ou mais.

Caso caia em um beco sem saída, deve voltar até o último ponto em que optou pelo caminho mais a direita e mudar sua decisão: agora você deve tentar o caminho mais a direita porém excluindo o que foi anteriormente selecionado (e que causou a chegada a um beco sem saída). Se você não tiver mais nenhuma opção, deve voltar mais ainda pelo caminho já percorrido até encontrar outro ponto em que você teve a opção de escolha. Esse processo de guardar o caminho e escolhas já feitos para tomar outra decisão é conhecido como **backtracking**.

Para implementar esse algoritmo você deve se lembrar do caminho percorrido e, além disso, em cada posição `(x, y)` em que houve opção de escolha deve também lembrar qual foi a última escolha feita, para poder muda-la quando você voltar o caminho (backtrack).

Uma solução é utilizar uma Pilha que armazene cada `(x,y)` onde uma decisão foi tomada, e também guardar um inteiro `opcao` que represente qual das opções foi escolhida no atual caminho: se escolhemos o mais da direita, guardamos `1`, se já precisamos voltar uma vez e escolhemos o segundo mais da direita, guardamos `2`, etc... Assim quando voltarmos a este ponto e precisarmos mudar de decisão, basta incrementar esse número. Caso não haja mais caminhos a percorrer por esse lado (isso é, se `opcao == totalDeOpcoes` daquele ponto), devemos regredir mais ainda o nosso caminho.

A maneira que percorremos esse labirinto é conhecida como **busca em profundidade**: vamos nos aprofundando no caminho atual até não poder mais, e só quando não der mais voltamos a profundidade do caminho para tentar outra alternativa.

Busca em Profundidade está muito relacionada ao uso de uma Pilha, e também pode ser realizada com o uso de **recursão**.

O algoritmo sempre funciona, desde que haja saída e que você seja mais rápido que o fatal Minotauro!

6.10 - Exercícios: Pilha

1) Implemente a classe `Peca` no pacote **br.com.caelum.ed** para poder criar objetos.

```
package br.com.caelum.ed;

public class Peca {

    private String nome;

    public String getNome() {
        return nome;
    }

    public void setNome(String nome) {
        this.nome = nome;
    }
}
```

2) Implemente a classe `Pilha` para peças vista neste capítulo. Coloque a classe no pacote **br.com.caelum.ed.pilhas**

```
package br.com.caelum.ed.pilhas;

import java.util.LinkedList;
import java.util.List;

import br.com.caelum.ed.Peca;

public class Pilha {

    private List<Peca> pecas = new LinkedList<Peca>();

    public void insere(Peca peca) {
        this.pecas.add(peca);
    }

    public Peca remove() {
        return this.pecas.remove(this.pecas.size() - 1);
    }

    public boolean vazia() {
        return this.pecas.size() == 0;
    }
```

```
}
```

Faça alguns testes.

```
package br.com.caelum.ed.pilhas;

import br.com.caelum.ed.Peca;

public class Teste {

    public static void main(String[] args) {
        Pilha pilha = new Pilha();

        Peca peca = new Peca();
        pilha.insere(peca);

        Peca pecaRemovida = pilha.remove();

        if(peca != pecaRemovida){
            System.out.println("Erro: a peça removida não é igual a que foi inserida");
        }

        if (!pilha.vazia()) {
            System.out.println("Erro: A pilha não está vazia");
        }
    }
}
```

Se não for impresso nenhuma mensagem de erro significa que a Pilha está funcionando.

3) Implemente a classe `PilhaGenerica` para objetos (genérica) vista neste capítulo. Coloque a classe no pacote **br.com.caelum.ed.pilhas**

```
package br.com.caelum.ed.pilhas;

import java.util.LinkedList;
import java.util.List;

public class PilhaGenerica {

    private List<Object> objetos = new LinkedList<Object>();

    public void insere(Object objeto) {
        this.objetos.add(objeto);
    }

    public Object remove() {
        return this.objetos.remove(this.objetos.size() - 1);
    }

    public boolean vazia() {
        return this.objetos.size() == 0;
    }
}
```

Faça alguns testes.

```
package br.com.caelum.ed.pilhas;

import br.com.caelum.ed.Peca;

public class TestePilhaGenerica {

    public static void main(String[] args) {
        PilhaGenerica pilhaDePecas = new PilhaGenerica();

        Peca peca = new Peca();
        pilhaDePecas.insere(peca);

        Peca pecaRemovida = pilhaDePecas.remove();

        if(peca != pecaRemovida){
            System.out.println("Erro: a peça removida não é igual a que foi inserida");
        }

        if (!pilhaDePecas.vazia()) {
            System.out.println("Erro: A pilha não está vazia");
        }
    }
}
```

Perceba que a classe `TestePilhaGenerica` contém um erro de compilção. Quando você remove um elemento da `PilhaGenerica` você recebe uma referência do tipo `Object` e não do tipo `Peca`.

Altere a linha:

```
Peca pecaRemovida = pilhaDePecas.remove();
```

Por:

```
Object pecaRemovida = pilhaDePecas.remove();
```

Isso faz o código compilar mas agora você não tem mais a garantia de tipo. Não sabe se a referência que você recebeu realmente está apontado para um objeto do tipo `Peca`.

4) Implemente a classe `PilhaParametrizada` utilizando o recurso do **Generics**. Coloque a classe no pacote **br.com.caelum.ed.pilhas**

```
package br.com.caelum.ed.pilhas;

import java.util.LinkedList;
import java.util.List;

public class PilhaParametrizada<T> {

    private List<T> objetos = new LinkedList<T>();

    public void insere(T t) {
        this.objetos.add(t);
    }
```

```
    public T remove() {
        return this.objetos.remove(this.objetos.size() - 1);
    }

    public boolean vazia() {
        return this.objetos.size() == 0;
    }
}
```

Faça alguns testes:

```
package br.com.caelum.ed.pilhas;

import br.com.caelum.ed.Peca;

public class TestePilhaGenerica {

    public static void main(String[] args) {
        PilhaParametrizada<Peca> pilhaDePecas = new PilhaParametrizada<Peca>();

        Peca peca = new Peca();
        pilhaDePecas.insere(peca);

        Peca pecaRemovida = pilhaDePecas.remove();

        if(peca != pecaRemovida){
            System.out.println("Erro: a peça removida não é igual a que foi inserida");
        }

        if (!pilhaDePecas.vazia()) {
            System.out.println("Erro: A pilha não está vazia");
        }

        PilhaParametrizada<String> pilhaDeString = new PilhaParametrizada<String>();

        pilhaDeString.insere("Rafael Cosentino");
        pilhaDeString.insere("Paulo Silveira");

        System.out.println(pilhaDeString.remove());
        System.out.println(pilhaDeString.remove());
    }
}
```

5) (**opcional**) É possível implementar a nossa Pilha utilizando internamente uma `ArrayList` em vez de `LinkedList`? Teremos algum ganho ou perda no consumo de tempo de alguma das operações?

6) (**opcional**) Uma mensagem é criptografada invertando cada palavra do texto. O texto "Uma mensagem confidencial" criptografado fica "amU megasnem laicnedifnoc". Implemente a criptografia e a decriptografia de mensagem. Faça isso utilizando Pilha. Para pegar caracteres específicos de uma String, utilize seu método `chatAt(int)`. Quando for colocar o caractere na Pilha, você vai perceber que não pode usar tipos primitivos como tipo parametrizado, então em vez de declarar uma pilha de `char` crie uma pilha do tipo **wrapper** `Character`.

CAPÍTULO 7

Filas

"O êxito não se consegue só com qualidades especiais. É sobretudo um trabalho de constância, de método e de organização"
– J. P. Sergent

7.1 - Introdução

No dia a dia, estamos acostumados com as filas em diversos lugares: nos bancos, nos mercados, nos hospitais, nos cinemas entre outros. As filas são importantes pois elas determinam a ordem de atendimento das pessoas.

As pessoas são atendidas conforme a posição delas na fila. O próximo a ser atendido é o primeiro da fila. Quando o primeiro da fila é chamdo para ser atendido a fila "anda", ou seja, o segundo passa a ser o primeiro, o terceiro passa a ser o segundo e assim por diante até a última pessoa.

Normalmente, para entar em uma fila, uma pessoa deve se colocar na última posição, ou seja, no fim da fila. Desta forma, quem chega primeiro tem prioridade.

Neste capítulo, estamos interessados em densenvolver estrutura de dados com o comportamentos das filas. Assim como Listas e Pilhas, as Filas são estruturas de dados que armazenam os elementos de maneira sequencial.

Assim como as Pilhas, as Filas têm operações mais **restritas** do que as operações das Listas. Nas Filas, os elementos são adicionados na última posição e removidos da primeira posição. Nas Listas, os elementos são adicionados e removidos de qualquer posição.

Então, podemos implementar uma Fila simplesmente colocando as retrições adequadas nas operações de adicionar e remover elementos de uma Lista. Isso é bem parecido ao que fizemos com as Pilhas.

Vamos implementar na seqüência uma Fila de Aluno. Iremos aproveitar a classe Aluno feita no capítulo de armazenamento sequencial.

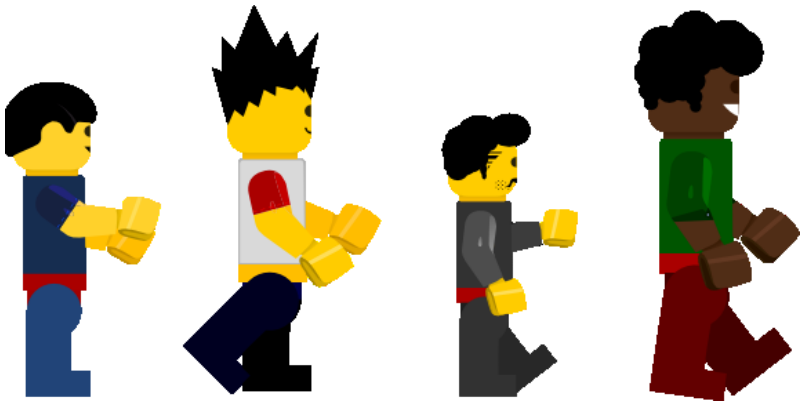


Fig.: Fila de pessoas

7.2 - Interface de uso

As operações que formam a interface de uso da Fila de alunos são:

1) Insere um Aluno (coloca um aluno no fim da Fila).
2) Remove um Aluno (retira o aluno que está no começo da Fila).
3) Informa se a Fila está vazia.

O esboço da classe `Fila` seria mais ou menos assim:

```
public class Fila {

    public void insere(Aluno aluno) {
        // implementação
    }

    public Aluno remove() {
        // implementação
    }

    public boolean vazia() {
        // implementação
    }
}
```

Agora que já temos a interface de uso da Fila definida vamos escrever algum teste sobre como ela deveria se comportar.

```
public class Teste {

    public static void main(String[] args) {
        Fila fila = new Fila();

        Aluno aluno = new Aluno();
        fila.insere(aluno);

        Aluno alunoRemovido = fila.remove();

        if (fila.vazia()) {
            System.out.println("A fila está vazia");
        }
    }
}
```

Como já foi dito aqui iremos implementar Fila utilizando algum tipo Lista. Neste capítulo, vamos utilizar a classe `LinkedList` para armazenar os alunos da Fila.

```
public class Fila {

    private List<Aluno> alunos = new LinkedList<Aluno>();

}
```

7.3 - Operações em Fila

Em seguida, implemetaremos as operações da Fila de aluno.

7.4 - Inserir uma aluno

Os alunos que entram na Fila devem sempre se colocar no fim da mesma. Vamos definir que o fim da Fila é o fim da Lista que estamos utilizando para implementar.

Então, entrar na Fila e adicionar no fim da Lista.

```
public class Fila {

    private List<Aluno> alunos = new LinkedList<Aluno>();

    public void insere(Aluno aluno) {
        this.alunos.add(aluno);
    }
}
```

7.5 - Remover um aluno

O próximo aluno a ser atendido é sempre o que está no início da Fila. No nosso caso, quem está no início da Fila é o aluno que está no início da Lista.

Então, basta remover o primeiro aluno.

```
public class Fila {

    private List<Aluno> alunos = new LinkedList<Aluno>();

    ...
    public Aluno remove() {
        return this.alunos.remove(0);
    }
}
```

É bom observar que se o método `remove()` for usado com a Fila vazia então uma exceção será lançada pois o método `removeFirst()` lança `IndexOutOfBoundsException` quando não existir elemento para remover.

7.6 - Informar se a Fila está vazia

Para implementar esta operação basta verificar se o tamanho da Lista é zero.

```
public class Fila {

    private List<Fila> alunos = new LinkedList<Fila>();

    ...
    public boolean vazia() {
        return this.alunos.size() == 0;
    }
}
```

7.7 - Generalização

Nossa Fila só funciona para guardar objetos da classe `Aluno`. Vamos generalizá-la para poder armazenar qualquer tipo de objeto. Isso será feito utilizando a classe `Object` da qual todas as classe derivam direta ou indiretamente. Criaremos uma `LinkedList` de `Object` em vez de uma `LinkedList` de `Aluno`.

```
public class Fila {

    private List<Object> objetos = new LinkedList<Object>();

    public void insere(Object objeto) {
        this.objetos.add(objeto);
    }

    public Object remove() {
        return this.objetos.remove(0);
    }

    public boolean vazia() {
        return this.objetos.size() == 0;
    }
}
```

Agora, podemos guardar qualquer tipo de objeto na Fila. Isso é uma grande vantagem pois a classe `Fila` poderá ser reaproveitada em diversas ocasiões. Mas, há uma desvantagem, quando removemos um elemento da Fila não podemos garantir qual é o tipo de objeto que virá.

A solução para este problema é utilizar o recurso do Generics. A nossa classe `Fila` vai ser uma classe parametrizada. Assim, quando criarmos uma Fila poderemos definir com qual tipo de objetos ela deve trabalhar.

Algo deste tipo:

```
Fila de alunos fila = new Fila de alunos();
```

Traduzindo este código para Java, ficaria assim:

```
Fila<Aluno> fila = new Fila<Aluno>();
```

Agora, precisamos parametrizar a classe Fila.

```
public class Fila<T> {
```

```
    private List<T> objetos = new LinkedList<T>();

    public void insere(T t) {
        this.objetos.add(t);
    }

    public T remove() {
        return this.objetos.remove(0);
    }

    public boolean vazia() {
        return this.objetos.size() == 0;
    }
}
```

Vamos criar duas Filas, uma para `Aluno` e outra para `String`.

```
public class Teste {

    public static void main(String[] args) {
        Fila<Aluno> fila = new Fila<Aluno>();

        Aluno aluno = new Aluno();
        fila.insere(aluno);

        Aluno alunoRemovido = fila.remove();

        if (fila.vazia()) {
            System.out.println("A fila está vazia");
        }

        Fila<String> filaDeString = new Fila<String>();
        filaDeString.insere("Rafael Cosentino");
        filaDeString.insere("Paulo Silvera");

        String paulo = filaDeString.remove();
        String rafael = filaDeString.remove();

        System.out.println(paulo);
        System.out.println(rafael);
    }
}
```

Em tempo de compilação, é verificado o tipo de objetos que estão sendo adicionados na Fila. Portanto, se você tentar inserir um objeto do tipo `Aluno` em uma Fila de `String` um erro de compilação será gerado.

```
public class Teste {

    public static void main(String[] args) {
        Aluno aluno = new Aluno();
        Fila<String> filaDeString = new Fila<String>();

        // este código não compila
        filaDeString.insere(aluno);
    }
```

```
}
```

7.8 - API do Java

Na biblioteca do Java, existe uma interface que define a estrutura de dados Fila. Essa interface chama-se `Queue`, umas das classes que implementam `Queue` é a `LinkedList`. O funcionamento fica estremamente parecido com a implementação que fizemos neste capítulo.

```
public class Teste {
    public static void main(String[] args) {

        Queue fila = new LinkedList();

        Aluno aluno = new Aluno();
        fila.offer(aluno);

        Aluno alunoRemovido = (Aluno)fila.poll();

        if(fila.isEmpty()){
            System.out.println("A fila está vazia");
        }
    }
}
```

Para evitar fazer casting de objetos, podemos utilizar o recurso de Generics aqui também.

```
public class Teste {
    public static void main(String[] args) {

        Queue<Aluno> fila = new LinkedList<Aluno>();

        Aluno aluno = new Aluno();
        fila.offer(aluno);

        Aluno alunoRemovido = fila.poll();

        if(fila.isEmpty()){
            System.out.println("A fila está vazia");
        }
    }
}
```

7.9 - Exercícios: Fila

1) Implemente a classe `Fila` para alunos vista neste capítulo. Coloque a classe no pacote **br.com.caelum.ed.filas**

```
package br.com.caelum.ed.filas;

import java.util.LinkedList;
import java.util.List;
```

```
import br.com.caelum.ed.Aluno;

public class Fila {

    private List<Aluno> alunos = new LinkedList<Aluno>();

    public void insere(Aluno aluno) {
        this.alunos.add(aluno);
    }

    public Aluno remove() {
        return this.alunos.remove(0);
    }

    public boolean vazia() {
        return this.alunos.size() == 0;
    }
}
```

Faça alguns testes.

```
package br.com.caelum.ed.filas;

import br.com.caelum.ed.Aluno;

public class Teste {

    public static void main(String[] args) {
        Fila fila = new Fila();

        Aluno aluno = new Aluno();
        fila.insere(aluno);

        Aluno alunoRemovido = fila.remove();

        if (aluno != alunoRemovido) {
            System.out.println("Erro: o aluno removido não é igual ao que foi inserido");
        }

        if (!fila.vazia()) {
            System.out.println("Erro: A fila não está vazia");
        }
    }
}
```

Se não for impresso nenhuma mensagem de erro significa que a Fila está funcionando.

2) Implemente a classe `FilaGenerica` para objetos (genérica) vista neste capítulo. Coloque a classe no pacote **br.com.caelum.ed.filas**

```
package br.com.caelum.ed.filas;

import java.util.LinkedList;
import java.util.List;
```

```
public class FilaGenerica {

    private List<Object> objetos = new LinkedList<Object>();

    public void insere(Object objeto) {
        this.objetos.add(objeto);
    }

    public Object remove() {
        return this.objetos.remove(0);
    }

    public boolean vazia() {
        return this.objetos.size() == 0;
    }
}
```

Faça alguns testes.

```
package br.com.caelum.ed.filas;

import br.com.caelum.ed.Aluno;

public class TesteFilaGenerica {

    public static void main(String[] args) {
        FilaGenerica filaDeAlunos = new FilaGenerica();

        Aluno aluno = new Aluno();
        filaDeAlunos.insere(aluno);

        Aluno alunoRemovido = filaDeAlunos.remove();

        if (aluno != alunoRemovido) {
            System.out
                    .println("Erro: o aluno removido não é igual ao que foi inserido");
        }

        if (!filaDeAlunos.vazia()) {
            System.out.println("Erro: A fila não está vazia");
        }
    }
}
```

Perceba que a classe `TesteFilaGenerica` contém um erro de compilção. Quando você remove um elemento da `FilaGenerica` você recebe uma referência do tipo `Object` e não do tipo `Aluno`.

Altere a linha:

```
Aluno alunoRemovido = filaDeAlunos.remove();
```

Por:

```
Object alunoRemovido = filaDeAlunos.remove();
```

Isso faz o código compilar mas agora você não tem mais a garantia de tipo. Não sabe se a referência que você recebeu realmente está apontado para um objeto do tipo `Aluno`.

3) Implemente a classe `FilaParametrizada` utilizando o recurso do **Generics**. Coloque a classe no pacote **br.com.caelum.ed.filas**

```
package br.com.caelum.ed.filas;

import java.util.LinkedList;
import java.util.List;

public class FilaParametrizada<T> {

    private List<T> objetos = new LinkedList<T>();

    public void insere(T t) {
        this.objetos.add(t);
    }

    public T remove() {
        return this.objetos.remove(0);
    }

    public boolean vazia() {
        return this.objetos.size() == 0;
    }
}
```

Faça alguns testes:

```
package br.com.caelum.ed.filas;

import br.com.caelum.ed.Aluno;

public class TesteFilaGenerica {

    public static void main(String[] args) {
        FilaParametrizada<Aluno> filaDeAlunos = new FilaParametrizada<Aluno>();

        Aluno aluno = new Aluno();
        filaDeAlunos.insere(aluno);

        Aluno alunoRemovido = filaDeAlunos.remove();

        if (aluno != alunoRemovido) {
            System.out
                    .println("Erro: o aluno removido não é igual ao que foi inserido");
        }

        if (!filaDeAlunos.vazia()) {
            System.out.println("Erro: A fila não está vazia");
        }

        FilaParametrizada<String> filaDeString = new FilaParametrizada<String>();
```

```
        filaDeString.insere("Rafael Cosentino");
        filaDeString.insere("Paulo Silveira");

        System.out.println(filaDeString.remove());
        System.out.println(filaDeString.remove());
    }
}
```

4) (**opcional**) É possível implementar a nossa Fila utilizando internamente uma `ArrayList` em vez de `LinkedList`? Teremos algum ganho ou perda no consumo de tempo de alguma das operações? Mostre a diferença através de um código que adiciona e remova muita gente da fila.

CAPÍTULO 8

Armazenamento sem repetição com busca rápida

"Não é a montanha que nos faz desanimar, mas a pedrinha que trazemos no sapato"
– Autor desconhecido

8.1 - Motivação

A impressão digital é a marca deixada pelas papilas dos dedos ao tocar em algum objeto. A íris é uma parte de olho humano que pode ser usada para identificar uma pessoa. Acredita-se que ambos, impressão digital e íris, são únicos para cada indivíduo, ou seja, não há duas impressões digitais iguais ou duas íris iguais.

Fig.: Impressão Digital

Fig.: Íris

É importante perceber também que manter as digitais em seqüência pode não ajudar em nada. Por exemplo, suponha que a polícia coloque as digitais em ordem alfabética do nome das pessoas. Quando um policial, na cena de um crime, recolhe uma digital, ele não sabe o nome da pessoa que tem aquela digital pois se soubesse

já teria resolvido o crime. Logo, o fato das digitais estarem em alguma seqüência não serve de nada para o policial.

Além de armazenarmos essas impressões digitais, uma operação chave sobre elas é poder consultar se determinar digital está contida em um grupo particular de digitais. Aqui não podemos fazer uma busca sequencial, comparando cada digital da base com a digital que está sendo procurada: isso consumiria tempo linear, o que seria inviavel para nossa aplicação, que possui uma base de dados grande. Queremos evitar o consumo de tempo linear durante a busca dessa digital.

As cores são a interpretação que o cérebro faz das ondas de luz. O comprimento de uma onda de luz varia de 7.5 X 10^-7m (vermelho) e 4 X 10^-7m (violeta). Cada comprimento de luz é interpretado pelo cérebro como uma cor diferente e única. Em outras palavras, não há cores repetidas.

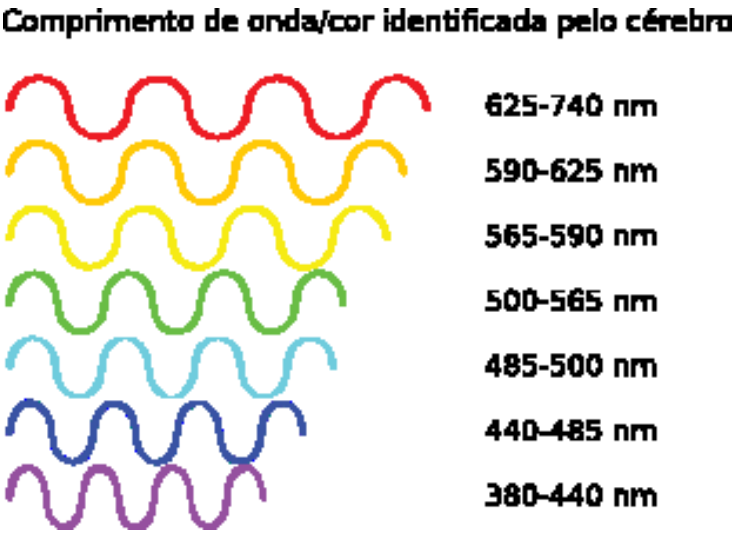


Fig.: Cores

Os mapas são a representação diagramática da Terra em uma superfície plana, em um pedaço de papel por exemplo. Alguns mapas destacam os países e um fato interessante que ocorre é que os países não se repetem, ou seja, são únicos.

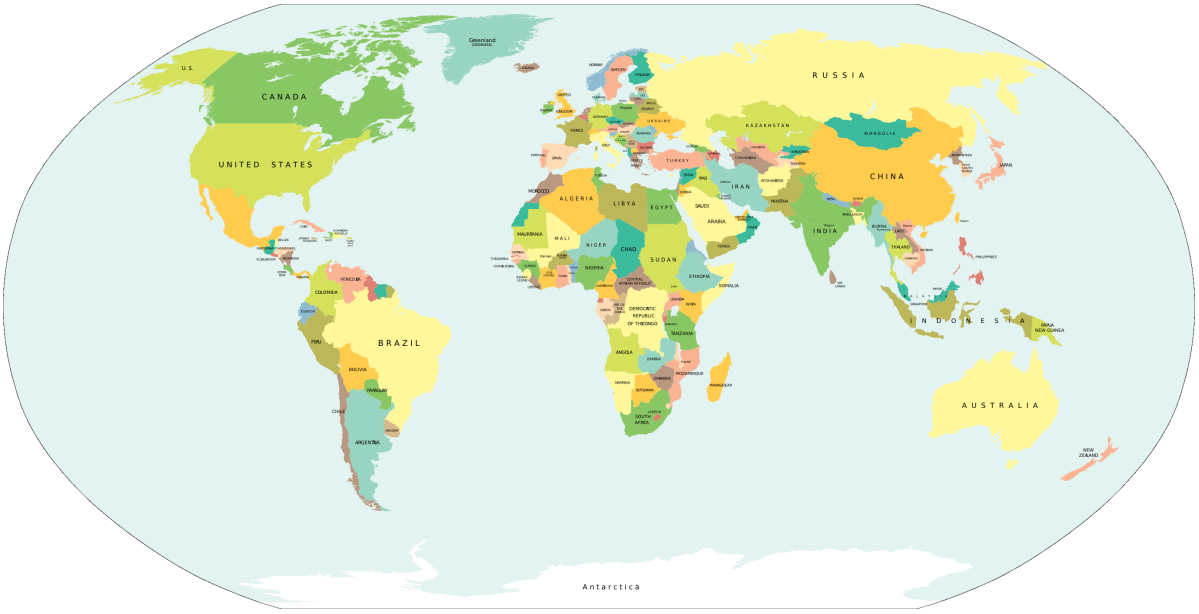


Fig.: Países

São muitos os exemplos em que há a necessidade de não permitir elementos repetidos dentro de um mesmo grupo, e em especial a finalidade de poder consultar se determinar elemento se encontra neste grupo, e que esta consulta seja computacionalmente eficiente.

8.2 - O problema do vocabulário

Um vocábulário é formado por um conjuto de palavras usadas na comunicação de um grupo de pessoas (um povo, uma nação, um país, etc).

Um requisito sobre as palavras que formam um vocabulário é que cada uma seja única. Não tem sentido existir palavras repetidas. Por outro lado, não há a necessidade das palavras ficarem em uma determinada seqüência. Se as palavras estiverem espalhadas sem estar numa seqüência específica, elas ainda podem formar o vocabulário língua.

Outro exemplo em que há a necessidade dos elementos serem únicos mas não obrigatoriamente estarem em seqüência são os baralhos. Imagine as cartas de um baralho, não pode existir cartas repetidas, elas são únicas.

Primeiro, suponha que todas as cartas estão dentro da caixa em que o baralho foi vendido. Desta forma, elas estão em alguma seqüência. Depois, suponha que todas as cartas estejam espalhadas sobre a mesa sem nenhuma seqüência. Em ambos os casos, as cartas continuam formando o baralho, independetemente se estão em seqüência ou não.

Queremos construir um sistema para armazenar as palavras de um vocabulário. Este sistema deve tomar as devidas providências para não permitir palavras repetidas.

As pessoas ao se comunicarem costumam inventar novas palavras que muitas vezes são provinientes de gírias ou inspiradas em outra língua. Algumas destas palavras, por serem tão usadas, acabam sendo incorporadas ao vocabulário de uma determinada língua. O sistema deve então fornecer uma operação para inserir palavras.

Na inserção de uma nova palavra na língua, o sistema deve verificar se a palavra já não existia pois não queremos repetir palavras. Esta verificação precisa ser rápida senão o sistema não será eficiente. Algum tipo especial de organização precisa ser usado para poder obter uma boa performance.

Da mesma forma que as palavras podem ser adicionadas na língua elas podem ser removidas. O sistema deve ser capaz de encotrar rapidamente as palavras que sairem do vocabulário da língua para remové-las.

Provavelemente precisaremos em alguns momentos percorrer todas as palavras armazenadas eficientemente. O sistema também deve fornecer uma funcionalidade para tal.

Outra operação útil seria alguma que informasse o número total de palavras da língua.

8.3 - Conjuntos

Vamos definir uma estrutura de dados para o sistema que armazenará o conjunto de palavras do vocabulário de alguma língua. Chamaremos esta estrutura de **Conjunto**.

Nesta seção, estamos interessados apenas em definir a interface de uso de um Conjunto. Mais adiante cuidaremos de implementar esta estrutura de dados.

As funcionalidades de um Conjunto são básicamente estas:

1) Adicionar um dado elemento.
2) Verificar se dado elemento está ou não no Conjunto.
3) Recuperar todos os elementos.
4) Informar o número de elementos.

CAPÍTULO 9

Tabelas de Espalhamento

"O primeiro passo para conseguirmos o que queremos na vida é decidirmos o que queremos"
– Ben Stein

Neste capítulo, implementaremos a estrutura de dados **Conjunto**.

9.1 - Introdução

Poderíamos simplesmente criar uma implementação desta estrutura que internamente encapsule uma `LinkedList` ou `ArrayList` como já fizemos anteriormente no caso da Fila e da Pilha.

Essa implementação é muito ineficiente computacionalmente: a cada inserção de palavra teremos que buscar dentro da Lista pela que está sendo adicionada, para evitar vocábulos repetidos. Também teremos de buscar dentro da Lista quando formos verificar se determinada palavra se encontra naquele vocabulário.

O problema de buscar em uma Lista é que fatalmente temos que percorrer ela por inteira e isso pode demorar muito: o consumo de tempo é linear em relação ao número de elementos que ela possui. O espaço de busca pode ser muito grande, a Lista pode ter milhões de elementos. Imagine uma Lista com as digitais de todos os brasileiros: são muitas digitais!

Em vez de buscar em todos os elementos do grupo, seria mais interessante conseguir restringir o espaço de busca, eliminando o maior número possível de elementos sem precisar "olhar" para eles.

Diferentemente das Listas, Pilhas e Filas, os Conjutos não precisam manter os elementos em seqüência. Então tentaremos uma abordagem bem diferente aqui. Vamos armazenar os elementos do Conjunto espalhados! Isso mesmo, espalhados, mas não aleatoriamente e sim com uma certa lógica que facilitará buscá-los.

A idéia é separar os elementos em categorias de acordo com certas características chaves do próprio elemento. Desta forma, para buscar um elemento, basta verificar qual é a categoria dele e assim eliminar todos os outros que estão nas outras categorias.

Essa idéia é fantástica e todo mundo já utilizou algo parecido no dia a dia. Um exemplo clássico em que essa técnica aparece é a agenda de telefones.

Nas agendas de telefones, os contatos são separados por categorias, normalmente, o que define a categoria do contato é a primeira letra do nome dele. Há uma página para cada letra do alfabeto. Desta forma, quando é preciso buscar um contato, vamos direto para a página referente a primeira letra do nome dele e assim eliminamos todos os contatos das outras páginas.

A técnica que estamos falando aqui é a técnica de **Espalhamento**. Ela consiste basicamente de duas partes: a **Função de Espalhamento** ("Função de Hash") e a **Tabela de Espalhamento** ("Tabela de Hash").

A Função de Espalhamento deve descobrir qual é a categoria de um determinado elemento. Para isso, ela deve analisar as caracteríticas chaves do próprio elemento. Por exemplo, na agenda de telefone, a Função deveria ver qual é a primeira letra do nome do contato e devolver o número da página referente àquela letra.

A Tabela de Espalhamento deve armazenar as categorias e cada categoria deve ter um índice. Este índice, gerado com a Função de Espalhamento, é usado para recuperar a categoria rapidamente.

A Função deve ser determinística, ou seja, toda vez que aplicada a um mesmo elemento ela deve gerar o mesmo índice, caso contrário procuraríamos um nome na página errada da agenda.

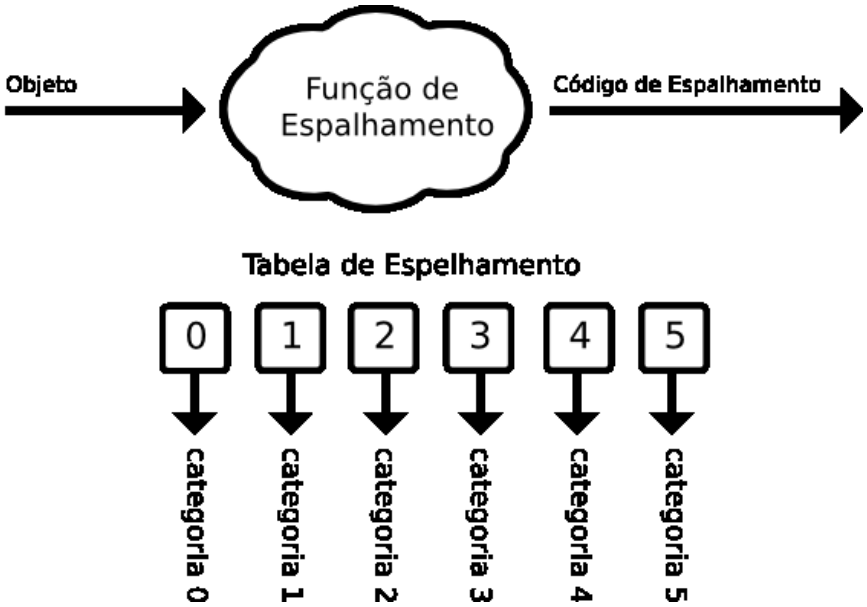


Fig.: Técnica de Espalhamento

No mundo ideal, a Função deveria gerar índices diferentes para elementos diferentes. Mas, na prática, ela pode, eventualmente, gerar o mesmo índice para elementos distintos. Quando isto ocorre, dizemos que houve uma **colisão**. No exemplo da agenda de contatos, uma colisão acontece quando o nome de dois contatos distintos iniciam com mesma letra.

Para resolver o problema da colisão, muitas vezes, a Tabela é combinada com outra estrutura de dados. Isso ocorre da seguinte forma: cada posição da Tabela pode armazenar uma Lista, uma Árvore ou outra estrutura de dados em vez de apenas um elemento. Desta forma conseguimos armazenas mais de um elemento na mesma posição da Tabela.

Cuidados devem ser tomados para não permitir a repetição de elementos nas estruturas que forem combinadas com a Tabela.

Vamos criar uma classe para implementar um Conjunto de palavras para uma língua utilizando a técnica de Espalhamento.

```
package br.com.caelum.ed.conjuntos;

public class ConjuntoEspalhamento {

    public void adiciona(String palavra) {
        // implementação
```

```
    }

    public void remove(String palavra) {
        // implementação
    }

    public boolean contem(String palavra) {
        // implementação
    }

    public List<String> pegaTodas(){
        // implementação
    }

    public int tamanho() {
        // implementação
    }
}
```

Podemos também definir alguns testes:

```
package br.com.caelum.ed.conjuntos;

public class Teste {
    public static void main(String[] args) {
        ConjuntoEspalhamento conjunto = new ConjuntoEspalhamento();

        conjunto.adiciona("palavra");
        conjunto.adiciona("computador");
        conjunto.adiciona("apostila");
        conjunto.adiciona("instrutor");
        conjunto.adiciona("mesa");
        conjunto.adiciona("telefone");

        if (!conjunto.contem("apostila")) {
            System.out.println("Erro: não tem a palavra: apostila");
        }

        conjunto.remove("apostila");

        if (conjunto.contem("apostila")) {
            System.out.println("Erro: tem a palavra: apostila");
        }

        if (conjunto.tamanho() != 5) {
            System.out.println("Erro: o tamanho do conjunto deveria ser 5");
        }
    }
}
```

9.2 - Tabela de Espalhamento

A Tabela de Espalhamento pode ser implementada através de uma Lista de Strings. Neste caso, cada

posição da Tabela poderá guardar somente uma palavra da língua. No mundo ideal, isto seria ótimo.

Mas, na prática, haverá colisões causadas pela Função de Espalhamento. Devemos combinar a Tabela com outra estrutura de dados para contornar as colisões.

Para combinar a Tabela com Listas, poderíamos definir um atributo do tipo Lista de Listas de Strings na classe `ConjuntoEspalhamento`.

Seria interessante que a Lista principal, aquela que armazena Listas, fosse um Vetor pois a princípio ela não sofrerá adição nem remoção de elementos.

As Listas secundárias, aquelas armazenadas na Lista principal, guardarão as palavras e elas sim sofrerão adição e remoção de elementos então seria interessante que estas elas fossem Listas Ligadas.

Lembrando que as classes do Java que implementam Vetor e Lista Ligada são `ArrayList` e `LinkedList` respectivamente.

```
package br.com.caelum.ed.conjuntos;

import java.util.ArrayList;
import java.util.LinkedList;

public class ConjuntoEspalhamento {

    private ArrayList<LinkedList<String>> tabela = new ArrayList<LinkedList<String>>();

    ...
}
```

Essa declaração java pode parecer complicada, mas você pode imaginar que aí criamos algo similar a uma matriz bidimensional, só que como são Collections do java, temos grandes vantagens e facilidades através de seus métodos.

Precisamos definir quantas posições a Tabela terá, ou seja, quantas Listas secundárias devemos ter, e inicializar cada uma delas. Isso pode ser feito no construtor da nossa classe. Inicialmente, teremos 26 posições na Tabela, uma para cada letra do alfabeto. Aqui vamos separar as palavras de acordo com a primeira letra delas: essa será nossa Função de Espalhamento. Assim como nas agendas de contatos de antigamente.

```
public class ConjuntoEspalhamento {

    private ArrayList<LinkedList<String>> tabela = new ArrayList<LinkedList<String>>();

    public ConjuntoEspalhamento() {
        for (int i = 0; i < 26; i++) {
            LinkedList<String> lista = new LinkedList<String>();
            tabela.add(lista);
        }
    }

    ...
}
```

Nós definimos o tipo do atributo `tabela` como `ArrayList` de `LinkedList`. Desta forma, estamos "amarrando" nossa implementação a tipos específicos de Listas. Se depois quisermos trocar os tipos das Listas teremos problemas. Podemos declarar e inicializar o atributo de forma mais genérica e elegante:

```
private List<List<String>> tabela = new ArrayList<List<String>>();
```

9.3 - Função de Espalhamento

A nossa Função deve observar a primeira letra das palavras e calcular o índice correto na Tabela. Por exemplo, palavras começadas com "a" devem ir para uma posição, palavras começadas com "b" devem ir para outra `1` e assim por diante até a letra "z".

Vamos fazer um método para implementar a Função de Espalhamento. Este método deve receber uma String (palavra) e devolver um int (índice associado a palavra).

```
private int calculaIndiceDaTabela(String palavra){
    return palavra.toLowerCase().charAt(0) % 26;
}
```

Observe que foram utilizados os métodos `toLowerCase()` e `chatAt(int)` da classe `String`. O primeiro devolve a palavra em letras minúsculas. O segundo devolve o caracter na posição passada como parâmetro. Além disso, foi feito uma operação de Resto da Divisão. Isso parece estranho mas faz todo o sentido visto que cada caracter tem um valor inteiro positivo.

Em vez de pegar o resto da divisão, poderíamos simplesmente subtrair o valor 65 desse caractere, que é o caractere `'A'`. Mas o que aconteceria se o primeiro caractere da String fosse um número ou um caractere especial?

Esta operação de resto ajusta o resultado da função de hash para não sair do intervalo fechado de `0` até `25`.

9.4 - Operações necessárias

Na seqüência, vamos implementar as operações que um Conjunto de palavras deve ter.

9.5 - Adicionar uma palavra

Para adicionar uma palavra no Conjunto, devemos aplicar a Função de Espalhamento para descobrir em qual posição da Tabela devemos adicionar. Depois, recuperamos a Lista que está nesta posição para guardar a palavra.

```
public void adiciona(String palavra) {
    int indice = this.calculaIndiceDaTabela(palavra);
    List<String> lista = this.tabela.get(indice);
    lista.add(palavra);
}
```

Este método ficou simples mas contém um erro. O requisito fundamental de um Conjunto é não ter elementos repetidos. Como as Listas permitem elementos repetidos se adcionarmos duas vezes a mesma palavra ela será inserida duas vezes.

Podemos evitar a repetição de palavras fazendo uma pequena verificação antes de adicionar uma palavra.

```
public void adiciona(String palavra) {
    if (!this.contem(palavra)) {
```

```
        int indice = this.calculaIndiceDaTabela(palavra);
        List<String> lista = this.tabela.get(indice);
        lista.add(palavra);
    }
}
```

Utilizamos o método `contem(String)` para saber se o Conjunto já contém a palavra. Se contém o `adiciona(String)` não faz nada.

9.6 - Remover uma palavra

Analogamente ao `adiciona(String)`, o método `remove(String)` deve achar a posição da Tabela onde está a Lista na qual a palavra a ser removida estaria. Depois, basta remover a palavra da Lista.

Antes de tentar remover poderíamos verificar se a palavra está no Conjunto. Nas Listas da API do Java, existe uma sobrecarga do método de `remover` que recebe o próprio elemento, além do o qual recebe um índice. Isto auxilia no nosso método:

```
public void remove(String palavra) {
    if (this.contem(palavra)) {
        int indice = this.calculaIndiceDaTabela(palavra);
        List<String> lista = this.tabela.get(indice);
        lista.remove(palavra);
    }
}
```

9.7 - Verificar se uma palavra está ou não no Conjunto

Esta operação é simples, basta achar o índice da Tabela aplicando a Função de Espalhamento da palavra desejada e verificar se ela está na Lista correspondente.

```
public boolean contem(String palavra) {
    int indice = this.calculaIndiceDaTabela(palavra);
    List<String> lista = this.tabela.get(indice);

    return lista.contains(palavra);
}
```

Aqui está o grande truque para deixar nosso Conjunto mais rápido que uma simples lista: buscamos apenas nos elementos que se encontram naquela "página da agenda". Se o elemento não estiver lá, com certeza ele não se encontra em nenhuma outra página da agenda. O nosso espalhamento tem uma certa organização que facilita as buscas!

9.8 - Recuperar todas as palavras do Conjunto

As palavras estão armazenadas na Tabela. Então, para recuperar todas as palavras, precisamos percorrer todas as posições da Tabela. Em cada posição, há uma Lista, pegaremos todos os elementos de cada Lista e armazenaremos em uma única Lista e a devolveremos.

```
public List<String> pegaTodas() {
```

```
    List<String> palavras = new ArrayList<String>();

    for (int i = 0; i < this.tabela.size(); i++) {
        palavras.addAll(this.tabela.get(i));
    }

    return palavras;
}
```

Este método cria uma Lista do tipo `ArrayList`. Porém, observe que a referência é do tipo `List`. Isso é possível por causa do **Polimorfismo** (capacidade de referênciar de várias maneiras um mesmo objeto). Todo objeto do tipo `ArrayList` também é do tipo `List`.

Todas as Listas do Java disponibilizam uma operação que permite adicionar diversos elementos de uma vez só. Esta funcionalidade é implementada pelo método `addAll(Collection)`. Este método foi utilizado dentro do `for` para inserir na Lista de todas as palavras do Conjunto todas as palavras da Lista da posição `i` da Tabela.

9.9 - Informar o tamanho do Conjunto de palavras

Para esta operação, temos duas alternativas ou em cada vez que a operação for executada nós percorremos todas as Listas contanto os elementos ou fazemos de uma forma mais eficiente que é guardar em um atributo a quantidade de palavras presentes no Conjunto. Este atributo deve ser incrementado toda vez que uma palavra for adicionada e decrementado toda vez que uma palavra for removida.

```
public class ConjuntoEspalhamento {

    ...

    private int tamanho = 0;

    ...

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

9.10 - Exercícios: Tabela de Espalhamento 1

1) Crie a classe `ConjuntoEspalhamento` para implementar Conjunto utilizando a técnica de Espalhamento. Faça esta classe no pacote **br.com.caelum.ed.conjuntos**. Coloque o atributo que implementa a tabela e o construtor que inicializa cada Lista da Tabela.

```
package br.com.caelum.ed.conjuntos;

import java.util.ArrayList;
import java.util.LinkedList;

public class ConjuntoEspalhamento {

    private List<List<String>> tabela = new ArrayList<List<String>>();
```

```
    public ConjuntoEspalhamento() {
        for (int i = 0; i < 26; i++) {
            LinkedList<String> lista = new LinkedList<String>();
            tabela.add(lista);
        }
    }
}
```

2) Faça o método que calcula qual é o índice da Tabela onde uma palavra deve ser armazenada. Este método deve classificar as palavras de acordo com a primeira letra.

```
private int calculaIndiceDaTabela(String palavra) {
    return palavra.toLowerCase().charAt(0) % 26;
}
```

3) Vamos implementar o método de adicionar palavras no Conjunto assim como foi discutido neste capítulo.

```
public void adiciona(String palavra) {
    if (!this.contem(palavra)) {
        int indice = this.calculaIndiceDaTabela(palavra);
        List<String> lista = this.tabela.get(indice);
        lista.add(palavra);
        this.tamanho++;
    }
}
```

Não esqueça de acrescentar o atributo `tamanho` para contabilizar o total de palavras.

```
private int tamanho = 0;
```

4) Para visualizar melhor o que está acontecendo como o Conjunto, vamos implementar agora o método que recupera todas as palavras da mesma forma que fizemos neste capítulo.

```
public List<String> pegaTodas() {
    List<String> palavras = new ArrayList<String>();

    for (int i = 0; i < this.tabela.size(); i++) {
        palavras.addAll(this.tabela.get(i));
    }

    return palavras;
}
```

Faça testes adicionando e imprimindo algumas palavras:

```
package br.com.caelum.ed.conjuntos;

import java.util.List;

public class TesteAdiciona {
    public static void main(String[] args) {
        ConjuntoEspalhamento conjunto = new ConjuntoEspalhamento();
```

```
        conjunto.adiciona("Rafael");
        conjunto.adiciona("Ana");
        conjunto.adiciona("Paulo");

        List<String> palavras = conjunto.pegaTodas();

        for (String palavra : palavras) {
            System.out.println(palavra);
        }
    }
}
```

Este teste deve imprimir o nome das três pessoas.

5) Implemente o `remove(String)`

```
public void remove(String palavra) {
    if (this.contem(palavra)) {
        int indice = this.calculaIndiceDaTabela(palavra);
        List<String> lista = this.tabela.get(indice);
        lista.remove(palavra);
        this.tamanho-;
    }
}
```

E teste:

```
public class TesteRemove {
    public static void main(String[] args) {
        ConjuntoEspalhamento conjunto = new ConjuntoEspalhamento();
        conjunto.adiciona("Rafael");
        conjunto.adiciona("Ana");
        conjunto.adiciona("Paulo");

        List<String> palavras = conjunto.pegaTodas();

        System.out.println("antes de remover");
        for (String palavra : palavras) {
            System.out.println(palavra);
        }

        conjunto.remove("Rafael");

        palavras = conjunto.pegaTodas();

        System.out.println("depois de remover");
        for (String palavra : palavras) {
            System.out.println(palavra);
        }
    }
}
```

6) Implemente o `contem(String)`

```
public boolean contem(String palavra) {
```

```
        int indice = this.calculaIndiceDaTabela(palavra);
        List<String> lista = this.tabela.get(indice);

        return lista.contains(palavra);
    }
```

Faça um teste adicionando algumas palavras e testando se método devolve `true` para elas. Teste também com palavras que não foram adicionadas. Neste caso, o método deve devolver `false`.

7) Implemente o método `tamanho()` e faça testes.

O Conjunto que implementamos não deve permitir repetição de palavras. Então, experimente adicionar palavras repetidas e depois verificar o tamanho do Conjunto.

9.11 - Diminuindo Colisões

Até o momento, não estamos tratando eficientemente as colisões. Devemos considerar dois problemas: o desbalanceamento da Tabela e a sobrecarga das Listas da Tabela.

O primeiro problema acontece quando a Função de Espalhamento não "espalha" direito os elementos. Por exemplo, na língua portuguesa há muito mais palavras começando com "a" do que com "z" então a nossa Função que considera apenas a primeira letra sobrecarrega muito mais a Lista da letra "a" do que a da letra "z" que fica praticamente vazia.

O segundo problema ocorre quando adicionamos muitas palavras. A Tabela tem tamanho fixo, ou seja, o número de Listas é constante. Ao adicionar `100000` palavras, calculando a melhor média possível, cada Lista interna da Tabela terá `100000/26 = 3846.2` elementos. Conforme as Listas vão ficando mais sobrecarregadas pior será o desempenho do sistema, chegando próximo do consumo de tempo linear.

Para resolver o primeiro problema, vamos fazer a Função de Espalhamento considerar melhor as características dos elementos do Conjunto. No nosso caso, a Função irá olhar para todas as letras da palavra em vez de olhar só para a primeira.

Para resolver o segundo, vamos fazer o tamanho da Tabela de Espalhamento ser dinâmico. Assim, se o Conjunto estiver com muitas palavras aumentamos este tamanho caso contrário diminuimos.

9.12 - Espalhando Melhor

As tarefas de analisar as características das palavras e de gerar um índice válido para a Tabela de Espalhamento estão "amarradas" em um único lugar, no método `calculaIndiceDaTabela(String)`.

Desta forma, será difícil fazer alterações independentes em cada uma destas duas importantes tarefas. Então, vamos separá-las em dois métodos distintos. Estes métodos funcionarão da seginte forma: o primeiro ficará encarregado de analisar uma dada palavra e gerar um código genérico que chamaremos de **Código de Espalhamento**, o segundo estará incumbido de a partir do Código de Espalhamento calcular um índice válido para a Tabela.

```
private int calculaCodigoDeEspalhamento(String palavra){
    int codigo = 1;
    for (int i = 0; i < palavra.length(); i++) {
        codigo = codigo + palavra.charAt(i);
    }
    return codigo;
```

```
}
```

Acima, o método que calcula o Código de Espalhamento faz um calculo sobre **todos** os caracteres que formam a String. Agora a palavra “caelum” não terá mais o mesmo resultado da função de hash que a palavra “campeão”.

Apesar dessa significativa melhora, ainda existem outras colisões básicas que podem acontecer, como “caelum” com “caemul": uma simples troca de ordem das letras gera uma colisão. Não é escopo deste texto, mas aqui para melhorar ainda mais a função de hash, podemos dar um peso para cada posição da String, assim os anagramas não resultarão em colisão, como por exemplo:

```
private int calculaCodigoDeEspalhamento(String palavra){
    int codigo = 1;
    for (int i = 0; i < palavra.length(); i++) {
        codigo = 31 * codigo + palavra.charAt(i);
    }
    return codigo;
}
```

Agora, o método `calculaIndiceDaTabela(String)` simplesmente usa o método `calculaCodigoDeEspalhamento` para obter o Código de Espalhamento e a partir dele calcula um índice válido para a Tabela (isto é, dentro dos índices da tabela):

```
private int calculaIndiceDaTabela(String palavra) {
    int codigoDeEspalhamento = this.calculaCodigoDeEspalhamento(palavra);
    return codigoDeEspalhamento % this.tabela.size();
}
```

Observe como o Código de Espalhamento é ajustado para gerar um índice da Tabela. A operação Resto da Divisão devolve um número no intervalo fechado `[0, this.tabela.size() - 1]`.

Como foi dito, o Código de Espalhamento de um elemento é genérico e deve considerar apenas as características do próprio elemento. Eventualmente, este código poderia ser até um número negativo. Então, devemos tomar cuidado com este fato.

Do jeito que está, o método `calculaIndiceDaTabela(String)` poderia gerar um índice negativo para a tabela. Isso ocorreria se o Código de Espalhamento fosse negativo. Com uma pequena modificação evitamos este problema.

```
private int calculaIndiceDaTabela(String palavra) {
    int codigoDeEspalhamento = this.calculaCodigoDeEspalhamento(palavra);
    codigoDeEspalhamento = Math.abs(codigoDeEspalhamento);
    return codigoDeEspalhamento % tabela.size();
}
```

Através do método `Math.abs(int)` obtemos o valor absoluto do Código de Espalhamento.

9.13 - Exercícios: Tabela de Espalhamento 2

1) Vamos verificar se realmente a nossa Função de Espalhamento está melhor que a antiga. Para isso, vamos fazer um método na classe `ConjuntoEspalhamento` que imprime o estado da Tabela.

```
public void imprimeTabela(){
    for (List<String> lista : this.tabela) {
        System.out.print("[");
        for (int i = 0; i < lista.size(); i++) {
            System.out.print("*");
        }
        System.out.println("]");
    }
}
```

E o seguinte teste:

```
package br.com.caelum.ed.conjuntos;

public class TesteEspalhamento {

    public static void main(String[] args) {

        ConjuntoEspalhamento conjunto = new ConjuntoEspalhamento();

        for (int i = 0; i < 100; i++) {
            conjunto.adiciona("" + i);
        }

        conjunto.imprimeTabela();
    }
}
```

Observe que as palavras ficam concentradas em algumas Listas.

Agora, faça as modificações para melhorar a Função de Espalhamento.

```
private int calculaCodigoDeEspalhamento(String palavra) {
    int codigo = 1;
    for (int i = 0; i < palavra.length(); i++) {
        codigo = 31 * codigo + palavra.charAt(i);
    }
    return codigo;
}

private int calculaIndiceDaTabela(String palavra) {
    int codigoDeEspalhamento = this.calculaCodigoDeEspalhamento(palavra);
    codigoDeEspalhamento = Math.abs(codigoDeEspalhamento);
    return codigoDeEspalhamento % tabela.size();
}
```

Execute novamente o teste e veja a diferença.

9.14 - Tabela Dinâmica

O tamanho da Tabela será dinâmico, ou seja, quanto mais elementos no Conjunto maior a Tabela e quanto menos elementos menor a Tabela. Precisamos de algum critério para decidir quando aumentar e quando diminuir o tamanho da Tabela.

A capacidade da Tabela é a capacidade das suas Listas. A Tabela está sobrecarregada quando o número de elementos é muito maior do que a capacidade. Então, vamos definir a carga da Tabela como sendo a razão entre o número de elementos e o número de Listas.

```
carga = tamanho / n\ de Listas
```

Se esta carga ultrapassar o valor `0.75` significa que há uma alta chance de ocorrer colisões. Neste caso, aumentaremos o tamanho da Tabela.

Se a carga ficar menor do que o valor `0.25` significa que a Tabela tem muito espaço vazio. Neste caso, diminuiremos o tamanho dela.

Este valores não foram chutados, eles são obtidos através de análise estatística avançada. Não faremos estas análises aqui pois foge do escopo deste texto.

Quando aumetarmos ou diminuirmos o tamanho da Tabela temos que redistribuir os elementos. Vamos criar um método que redimensiona a Tabela dado uma nova capacidade.

```
private void redimensionaTabela(int novaCapacidade){
    List<String> palavras = this.pegaTodas();
    this.tabela.clear();

    for (int i = 0; i < novaCapacidade; i++) {
        this.tabela.add(new LinkedList<String>());
    }

    for (String palavra : palavras) {
        this.adiciona(palavra);
    }
}
```

O que é feito neste método é o seguinte:

1) Guarda em uma Lista todas as palavras do Conjunto.

2) Limpa a tabela através do método `clear()` que toda Lista do Java tem.

3) Inicializa a Tabela com a quantida de Lista da novaCapacidade.

4) Guarda novamente as palavras no Conjunto.

Agora ao adicionar ou remover um elemento do Conjunto, verificamos se a carga está acima ou abaixo do aceitável. Caso estiver, redimensionamos adequadamente.

```
private void verificaCarga() {
    int capacidade = this.tabela.size();
    double carga = (double) this.tamanho / capacidade;

    if (carga > 0.75) {
        this.redimensionaTabela(capacidade * 2);
    } else if (carga < 0.25) {
        this.redimensionaTabela(Math.max(capacidade / 2, 10));
    }
}
```

O método acima calcula a carga atual do conjunto. Se a carga for maior que `0.75` a novaCapacidade será o dobro da antiga. Se a carga for menor que `0.25` a novaCapacidade será a metade da antiga mas para evitar ficar com uma capacidade muito pequena definimos que o mínimo é 10.

O `verficaCarga()` deve ser utilizado antes de adicionar e após remover uma palavra.

9.15 - Exercícios: Tabela de Espalhamento 3

1) Neste ponto, nossa implementação deve estar sobrecarregando as Listas da Tabela deixando o desempenho ruim. Para verificar este fato faça o seguinte teste:

```
package br.com.caelum.ed.conjuntos;

public class TesteDesempenho {

    public static void main(String[] args) {

        long inicio = System.currentTimeMillis();

        ConjuntoEspalhamento conjunto = new ConjuntoEspalhamento();

        for (int i = 0; i < 50000; i++) {
            conjunto.adiciona("palavra" + i);
        }

        for (int i = 0; i < 50000; i++) {
            conjunto.contem("palavra" + i);
        }
    }
}
```

Marque o tempo gasto!

Faça as melhorias adicionado o recurso de Tabela de Espalhamento dinâmica.

```
private void redimensionaTabela(int novaCapacidade) {
    List<String> palavras = this.pegaTodas();
    this.tabela.clear();

    for (int i = 0; i < novaCapacidade; i++) {
        this.tabela.add(new LinkedList<String>());
    }

    for (String palavra : palavras) {
        this.adiciona(palavra);
    }
}

private void verificaCarga() {
    int capacidade = this.tabela.size();
    double carga = (double) this.tamanho / capacidade;

    if (carga > 0.75) {
        this.redimensionaTabela(capacidade * 2);
```

```
        } else if (carga < 0.25) {
            this.redimensionaTabela(Math.max(capacidade / 2, 10));
        }
    }
```

Não esqueça de invovar o `verificaCarga()` antes de adicionar uma palavra e depois de remover.

```
    public void adiciona(String palavra) {
        if (!this.contem(palavra)) {
            this.verificaCarga();
            // RESTO DO CÓDIGO
        }
    }

    public void remove(String palavra) {
        if (this.contem(palavra)) {
            // RESTO DO CÓDIGO
            this.verificaCarga();
        }
    }
```

Execute novamente o teste e veja a diferença.

9.16 - Generalização

A nossa implementação de Conjunto usando a técnica de Espalhamento funciona apenas para armazenar palavras (Strings). Estamos amarrando a estrutura de dados ao tipo de dado que ela vai guardar. Seria melhor generalizar o nosso Conjunto.

Assim como fizemos em capítulos anteriores, vamos trocar os tipos específicos por referências da classe `Object`. Diferentemente das outras estruturas que implementamos até o momento, a generalização do Conjunto será um pouco mais delicada.

O problema é o calculo do Código de Espalhamento. Este calculo depende do tipo do objeto. Para gerar um bom Código de Espalhamento o Conjunto deveria conhecer as características de todos os tipo de objetos. Isso seria inviável.

A solução é deixar alguém que conhece muito bem as características do objeto a ser armazenado no Conjunto gerar o Código de Espalhamento. Quem melhor conhece um objeto e a classe dele. Então, as classes dos objetos que precisam ser armazenados devem definir como gerar o Código de Espalhamento.

No Java, a classe `Object` define um método para gerar o Código de Espalhamento. Este método é o `hashCode()`. Eventualmente, você pode reescrever este método para implementar o seu Código de Espalhamento ou pode utilizar o que está feito em `Object`.

```
private int calculaIndiceDaTabela(Object objeto) {
    int codigoDeEspalhamento = objeto.hashCode();
    codigoDeEspalhamento = Math.abs(codigoDeEspalhamento);
    return codigoDeEspalhamento % this.tabela.size();
}
```

O método `calculaCodigoDeEspalhamento(Object)` não é mais necessário na classe que implementa o Conjunto.

```
package br.com.caelum.ed.conjuntos;

import java.util.ArrayList;
import java.util.LinkedList;
import java.util.List;

public class ConjuntoEspalhamentoGenerico {

    private List<List<Object>> tabela = new ArrayList<List<Object>>();

    private int tamanho = 0;

    public ConjuntoEspalhamentoGenerico() {
        for (int i = 0; i < 26; i++) {
            LinkedList<Object> lista = new LinkedList<Object>();
            tabela.add(lista);
        }
    }

    public void adiciona(Object objeto) {
        if (!this.contem(objeto)) {
            this.verificaCarga();
            int indice = this.calculaIndiceDaTabela(objeto);
            List<Object> lista = this.tabela.get(indice);
            lista.add(objeto);
            this.tamanho++;
        }
    }

    public void remove(Object objeto) {
        if (this.contem(objeto)) {
            int indice = this.calculaIndiceDaTabela(objeto);
            List<Object> lista = this.tabela.get(indice);
            lista.remove(objeto);
            this.tamanho-;
            this.verificaCarga();
        }
    }

    public List<Object> pegaTodos() {
        List<Object> objetos = new ArrayList<Object>();
        for (int i = 0; i < this.tabela.size(); i++) {
            objetos.addAll(this.tabela.get(i));
        }
        return objetos;
    }

    public boolean contem(Object objeto) {
        int indice = this.calculaIndiceDaTabela(objeto);
        List<Object> lista = this.tabela.get(indice);

        return lista.contains(objeto);
    }

    public int tamanho() {
```

```
        return this.tamanho;
    }

    private int calculaIndiceDaTabela(Object objeto) {
        int codigoDeEspalhamento = objeto.hashCode();
        codigoDeEspalhamento = Math.abs(codigoDeEspalhamento);
        return codigoDeEspalhamento % tabela.size();
    }

    private void redimensionaTabela(int novaCapacidade) {
        List<Object> objetos = this.pegaTodos();
        this.tabela.clear();

        for (int i = 0; i < novaCapacidade; i++) {
            this.tabela.add(new LinkedList<Object>());
        }

        for (Object objeto : objetos) {
            this.adiciona(objeto);
        }
    }

    private void verificaCarga() {
        int capacidade = this.tabela.size();
        double carga = (double) this.tamanho / capacidade;

        if (carga > 0.75) {
            this.redimensionaTabela(capacidade * 2);
        } else if (carga < 0.25) {
            this.redimensionaTabela(Math.max(capacidade / 2, 10));
        }
    }

    public void imprimeTabela() {
        for (List<Object> lista : this.tabela) {
            System.out.print("[");
            for (int i = 0; i < lista.size(); i++) {
                System.out.print("*");
            }
            System.out.println("]");
        }
    }
}
```

9.17 - equals e hashCode

Repare que a nossa busca pelo elemento usa o método `contains` da `List`. A implementação de lista que estamos usando (assim como a maioria absoluta das coleções da API do Java) utiliza do método `equals` para saber se determinado elemento está contido naquela lista ou não.

Um problema aparece quando reescrevemos o `equals` e não reescrevemos o `hashCode`: pode ser que existam dois objetos, digamos `a` e `b`, que são considerados iguais pelo nosso método, mas possuem códigos de espalhamento diferentes. Isto vai fazer com que a pequisa por `a`, no caso de só existir `b` dentro do conjunto

retorne `false`, sendo que na verdade deveria retornar `true` pois estamos levando em consideração o que o `equals` diz, e não o operador `==`.

Em outras palavras, devemos seguir o **contrato da classe Object**, que diz que se:

```
a.equals(b)
```

então:

```
a.hashCode() == b.hashCode()
```

Essa relação não é bidirecional! Se dois elementos têm o mesmo `hashCode`, eles não precisam (mas podem) necessariamente ser o mesmo... é o caso em que há colisão! Já sabemos que quanto menos colisão, melhor para uma distribuição e carga mais balanceada em nossa tabela.

Se não reescrevemos o `equals` não há problema: a implementação padrão do `hashCode` definido na classe `Object` é a de devolver um identificador único para aquele objeto. Como o `equals` definido na classe `Object` se comporta como o operador `==`, é fácil de perceber que o contrato da classe `Object` definido acima será respeitado.

9.18 - Parametrizando o Conjunto

Podemos utilizar o recurso de Generics para parametrizar a nossa classe.

```
package br.com.caelum.ed.conjuntos;

import java.util.ArrayList;
import java.util.LinkedList;
import java.util.List;

public class ConjuntoEspalhamentoParametrizado<T> {

    private List<List<T>> tabela = new ArrayList<List<T>>();

    private int tamanho = 0;

    public ConjuntoEspalhamentoParametrizado() {
        for (int i = 0; i < 26; i++) {
            LinkedList<T> lista = new LinkedList<T>();
            tabela.add(lista);
        }
    }

    public void adiciona(T objeto) {
        if (!this.contem(objeto)) {
            this.verificaCarga();
            int indice = this.calculaIndiceDaTabela(objeto);
            List<T> lista = this.tabela.get(indice);
            lista.add(objeto);
            this.tamanho++;
        }
    }

    public void remove(T objeto) {
```

```
        if (this.contem(objeto)) {
            int indice = this.calculaIndiceDaTabela(objeto);
            List<T> lista = this.tabela.get(indice);
            lista.remove(objeto);
            this.tamanho-;
            this.verificaCarga();
        }
    }

    public List<T> pegaTodos() {
        List<T> objetos = new ArrayList<T>();
        for (int i = 0; i < this.tabela.size(); i++) {
            objetos.addAll(this.tabela.get(i));
        }
        return objetos;
    }

    public boolean contem(T objeto) {
        int indice = this.calculaIndiceDaTabela(objeto);
        List<T> lista = this.tabela.get(indice);

        return lista.contains(objeto);
    }

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

    private int calculaIndiceDaTabela(T objeto) {
        int codigoDeEspalhamento = objeto.hashCode();
        codigoDeEspalhamento = Math.abs(codigoDeEspalhamento);
        return codigoDeEspalhamento % tabela.size();
    }

    private void redimensionaTabela(int novaCapacidade) {
        List<T> objetos = this.pegaTodos();
        this.tabela.clear();

        for (int i = 0; i < novaCapacidade; i++) {
            this.tabela.add(new LinkedList<T>());
        }

        for (T objeto : objetos) {
            this.adiciona(objeto);
        }
    }

    private void verificaCarga() {
        int capacidade = this.tabela.size();
        double carga = (double) this.tamanho / capacidade;

        if (carga > 0.75) {
            this.redimensionaTabela(capacidade * 2);
        } else if (carga < 0.25) {
            this.redimensionaTabela(Math.max(capacidade / 2, 10));
```

```
        }
    }

    public void imprimeTabela() {
        for (List<T> lista : this.tabela) {
            System.out.print("[");
            for (int i = 0; i < lista.size(); i++) {
                System.out.print("*");
            }
            System.out.println("]");
        }
    }
}
```

9.19 - API do Java

Na biblioteca do Java há uma classe que faz exatamente o que fizemos neste capítulo, implementar um Conjunto genérico utilizando a técnica de espalhamento. Esta classe é a `HashSet`.

Exemplo de uso:

```
package br.com.caelum.ed.conjuntos;

import java.util.HashSet;

public class TesteHashSet {
    public static void main(String[] args) {
        HashSet conjunto = new HashSet();
        conjunto.add("Rafael");
        conjunto.add("Rafael");
        conjunto.add("Ana");
        conjunto.add("Paulo");

        System.out.println(conjunto);
    }
}
```

9.20 - Exercícios: Tabela de Espalhamento 4

1) (Gereralizando) Faça o método `calculaIndiceDaTabela` utilizar o método `hashCode` e substitua as referências do tipo `String` por `Object`. Assim como foi feito na seção de Generalização.

```
private int calculaIndiceDaTabela(Object objeto) {
    int codigoDeEspalhamento = objeto.hashCode();
    codigoDeEspalhamento = Math.abs(codigoDeEspalhamento);
    return codigoDeEspalhamento % tabela.size();
}
```

Execute o teste:

```
package br.com.caelum.ed.conjuntos;
```

```
public class TesteConjuntoGenerico {
    public static void main(String[] args) {
        ConjuntoEspalhamentoGenerico conjunto = new ConjuntoEspalhamentoGenerico();
        conjunto.adiciona("Rafael");
        conjunto.adiciona("Rafael");
        conjunto.adiciona("Ana");
        conjunto.adiciona("Paulo");

        System.out.println(conjunto.pegaTodos());
    }
}
```

2) Teste a classe `HashSet`.

```
package br.com.caelum.ed.conjuntos;

import java.util.HashSet;

public class TesteHashSet {
    public static void main(String[] args) {
        HashSet conjunto = new HashSet();
        conjunto.add("Rafael");
        conjunto.add("Rafael");
        conjunto.add("Ana");
        conjunto.add("Paulo");

        System.out.println(conjunto);
    }
}
```

CAPÍTULO 10

Armazenamento Associativo

"A cooperação é a convicção plena de que ninguém pode chegar à meta se não chegarem todos"
– Virginia Burden

10.1 - Motivação

Hoje em dia, devido a correria do dia a dia, estamos acostumados com os restaurantes "fast food". Como o próprio nome diz o principal atrativo destes restaurantes é ou deveria ser a rapidez no atendimento. Normalmente, para tornar mais ágil o atendimento, os pedidos são feitos por número. O cliente escolhe o que deseja comer e informa o número correspondete ao seu pedido ao atendente. Cada número está associado a um prato ou a um pedido.

Para controlar a circulação de carros e motos nas ruas das cidades, algum orgão governamental associa uma placa a cada veículo. Para emitir uma multa ou denunciar um roubo à polícia, informamos a placa que identifica o veículo. A placa é formada de letras e números e funciona como uma chave para encontrar o veículo desejado.

A televisão está presente na vida de todas as pessoas. Ela é o principal meio de comunicação nos dias atuais e é dividida em canais. Cada canal escolhe a sua própria programação de acordo com o público alvo. Para escolher qual canal assistir, uma pessoa digita no controle ou no próprio aparelho de TV o número que está associado ao canal desejado.

Além destes, há muitos outros exemplos em que um elemento está associado a outro. Por isso, vamos estudar alguma estrutura de dados que permita associações deste tipo.

10.2 - Mapa

A primeira funcionalidade que precisamos na nossa estrutura e uma que faça as associações entre os elementos. Esta operação deve receber uma chave e um valor para criar a associação entre eles.

É necessário recuperar o valor associado a uma chave rapidamente. Uma operação que faça isso deve ser implementada. Ela receberá uma chave e devolverá o valor associado se ele existir.

Provavelmente, será necessário remover uma associação da nossa estrutura em alguns casos. Deverá existir uma funcionalidade para fazer isso. Dado uma chave remove a associação que a chave participa.

Outra operação interessante seria poder verificar se uma determinada chave está ou não associada a algum valor.

Além disso, precisamos saber quantas associações existem na estrutura.

Um requisito fundamental para o Mapa é que as chaves sejam únicas. Não pode existir dois carros com a mesma placa, dois canais de televisão no mesmo número ou dois pedidos de números iguais.

Os Mapas são as estruturas de dados que implementam o tipo de situação que estamos discutindo aqui.

1) Adicionar uma associação.

2) Pegar um valor dado uma chave.

3) Remover uma associação dado uma chave.

4) Verificar se existe uma associação para uma determinada chave.

5) Informar a quantidade de associações.

Queremos modelar uma estrutura de dados para ser utilizada em um sistema de armazenamento de registros de veículos. Este sistema poderia ser útil para o Detran.

As informações sobre um determinado carro são obtidas a partir da placa do mesmo. Então, o mapa que queremos fazer aqui vai associa as placas aos carros. As placas podem ser `String` e para os carros podemos implementar uma classe `Carro`.

```
class Carro {

    private String nome;
    private String marca;
    private String cor;
    private int ano;

    // getters e setters
}
```

O ponto fundamental do Mapa é a associação das chaves com os valores. Para modelar uma associação, vamos definir uma classe.

```
class Associacao {
    private String placa;
    private Carro carro;

    public Associacao(String placa, Carro carro){
        this.placa = placa;
        this.carro = carro;
    }
    // getters
}
```

10.3 - Exercícios: Armazenamento Associativo

1) Implemente a classe `Carro` no pacote **br.com.caelum.ed**.

```
public class Carro {
    private String nome;

    public Carro(String nome) {
        this.nome = nome;
```

```
    }

    public String getNome() {
        return nome;
    }

    @Override
    public String toString() {
        return "Carro: " + this.nome;
    }
}
```

2) Implemente a classe `Associacao` que modela o relacionamento entre as chaves e os valores do Mapa no pacote **br.com.caelum.ed.mapas**.

```
public class Associacao {
    private String placa;
    private Carro carro;

    public Associacao(String placa, Carro carro) {
        this.placa = placa;
        this.carro = carro;
    }

    public String getPlaca() {
        return placa;
    }

    public Carro getCarro() {
        return carro;
    }
}
```

CAPÍTULO 11

Mapas com Lista

"Os nossos desejos são como crianças pequenas: quanto mais lhes cedemos, mais exigentes se tornam"
– Provérbio Chinês

11.1 - Introdução

A maneira mais simples de implementar um Mapa é guardar as associações pertencentes a ele em uma Lista.

Como sabemos quais operações queremos podemos definir o "esqueleto" da classe que vai implementar o Mapa.

```
public class MapaLista {

    private List<Associacao> associacoes = new ArrayList<Associacao>();

    public void adiciona(String placa, Carro carro) {
    }

    public Carro pega(String placa) {
    }

    public void remove(String placa) {
    }

    public boolean contemChave(String placa) {
    }
}
```

Antes de adicionar uma associação no Mapa devemos verificar se a chave da nova associação não pertence a alguma associação da Lista.

Para remover uma associação, verificar se uma chave está associada a um valor ou recuperar o valor associado a uma determinada chave é necessário percorrer a Lista.

11.2 - Operações em mapas

Vamos ver como implementar as operações nos mapas.

11.3 - Adicionar uma associação

O Mapa não pode permitir duas associações com a mesma chave. Então, fazemos uma verificação para saber se a chave já está no Mapa. Utilizamos o método `contemChave(String)` para este teste.

```
public void adiciona(String placa, Carro carro) {
    if (!this.contemChave(placa)) {
        Associacao associacao = new Associacao(placa, carro);
        this.associacoes.add(associacao);
    }
}
```

Esta operação é bem eficiente porque adicionar no fim de qualquer tipo de Lista é muito rápido.

Poderíamos ter escolhido uma outra decisão aqui: se a chave já existisse, trocamos o valor associado para este novo Carro. A API do Java trata isso desta maneira.

11.4 - Recuperar o valor associado a uma dada chave

Da maneira que foi implementado devemos percorrer todas as associações para achar a desejada. Se a chave não estiver presente no Mapa uma exceção é lançada.

```
public Carro pega(String placa) {
    for (Associacao associacao : this.associacoes) {
        if (placa.equals(associacao.getPlaca())) {
            return associacao.getCarro();
        }
    }
    throw new IllegalArgumentException("chave não existe");
}
```

Este método também deve percorrer a Lista de associações. Então o consumo de tempo é linear.

11.5 - Remover a associação que contem uma determinada chave

Comparamos a chave recebida no parâmetro com as chaves de todas as associações da Lista. Se alguma for igual então marcamos a associação para remover. Não podemos remover dentro do **for** por causa da concorrência. Se o Mapa não tem uma associação com a chave procurada então uma exceção é lançada.

```
public void remove(String placa) {

    if (this.contemChave(placa)) {

        for (int i = 0; i < this.associacoes.size(); i++) {
            Associacao associacao = this.associacoes.get(i);

            if (placa.equals(associacao.getPlaca())) {
                this.associacoes.remove(i);
                break;
            }
        }

    } else {
        throw new IllegalArgumentException("chave não existe");
    }
```

```
}
```

O consumo de tempo deste método também é linear.

11.6 - Verificar se uma dada chave está em alguma associação

Esta operação é simples, basta percorrer a Lista e comparar as chaves. Logo o consumo de tempo será linear.

```
public boolean contemChave(String placa) {
    for (Associacao associacao : this.associacoes) {
        if (placa.equals(associacao.getPlaca())) {
            return true;
        }
    }
    return false;
}
```

11.7 - Informar o tamanho do Mapa

Como todas as associações estão armazenadas em uma Lista, o tamanho do Mapa é o tamanho da Lista.

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

11.8 - Exercícios: Mapas

1) Implemente a classe `Carro` no pacote **br.com.caelum.ed**.

```
public class Carro {
    private String nome;

    public Carro(String nome) {
        this.nome = nome;
    }

    public String getNome() {
        return nome;
    }

    @Override
    public String toString() {
        return "Carro: " + this.nome;
    }
}
```

2) Faça o Mapa com Listas. Assim como vimos neste capítulo.

```
public class MapaLista {

    private List<Associacao> associacoes = new ArrayList<Associacao>();

    public void adiciona(String placa, Carro carro) {
        if (!this.contemChave(placa)) {
            Associacao associacao = new Associacao(placa, carro);
            this.associacoes.add(associacao);
        }
    }

    public Carro pega(String placa) {
        for (Associacao associacao : this.associacoes) {
            if (placa.equals(associacao.getPlaca())) {
                return associacao.getCarro();
            }
        }
        throw new IllegalArgumentException("chave não existe");
    }

    public void remove(String placa) {
        if (this.contemChave(placa)) {

            for (int i = 0; i < this.associacoes.size(); i++) {
                Associacao associacao = this.associacoes.get(i);

                if (placa.equals(associacao.getPlaca())) {
                    this.associacoes.remove(i);
                    break;
                }
            }

        } else {
            throw new IllegalArgumentException("chave não existe");
        }
    }

    public boolean contemChave(String placa) {
        for (Associacao associacao : this.associacoes) {
            if (placa.equals(associacao.getPlaca())) {
                return true;
            }
        }
        return false;
    }
}
```

3) Faça um teste para medir a performance do nosso Mapa.

```
public class TesteTempoMapaLista {

    public static void main(String[] args) {

        MapaLista mapaLista = new MapaLista();
        int numeroDeElementos = 15000;
```

```
        long inicio = System.currentTimeMillis();

        for (int i = 0; i < numeroDeElementos; i++) {
            mapaLista.adiciona("" + i, new Carro("c" + i));
        }

        for (int i = 0; i < numeroDeElementos; i++) {
            mapaLista.pega("" + i);
        }

        for (int i = 0; i < numeroDeElementos; i++) {
            mapaLista.contemChave("" + i);
        }

        for (int i = 0; i < numeroDeElementos; i++) {
            mapaLista.remove("" + i);
        }


        long fim = System.currentTimeMillis();

        System.out.println("Tempo: " + (fim - inicio)/1000.0);
    }
}
```

CAPÍTULO 12

Mapas com Espalhamento

"Se está muito difícil encontrar o caminho, faça-o"
– Autor desconhecido

12.1 - Introdução

A implementação de Mapa usando Listas não é eficiente pois em todas as operações precisamos percorrer todo Conjunto de associações e isso pode ficar muito custoso a medida que o número de associações cresce.

Novamente podemos utilizar a técnica de Espalhamento para obter resultados melhores. É importante notar que os Mapas são semelhantes aos Conjuntos. Nos Conjuntos, os elementos não podem se repetir. Nos Mapas, as chaves das associações não podem se repetir.

Para utilizar a técnica de Espalhamento precisamos definir basicamente a Função de Espalhamento e a Tabela de Espalhamento.

Vimos que o Código de Espalhamento não é gerado pela estrutura de dados, no caso pelo Mapa, e sim pelo próprio elemento que vamos trabalhar, no caso as chaves. No Java, temos o método `hashCode()` para calcular este código.

A Tabela de Espalhamento é implementada como uma Lista de Lista de Associações. Analogamente ao que fizemos para implementar Conjuntos.

```
public class MapaEspalhamento {
    private List<List<Associacao>> tabela = new ArrayList<List<Associacao>>();
}
```

Além disso, precisamos inicializar cada posição da Tabela. Isso será feito no construtor.

```
public MapaEspalhamento() {
    for (int i = 0; i < 100; i++) {
        this.tabela.add(new LinkedList<Associacao>());
    }
}
```

Para ajustar o código gerado pelo método `hashCode()` e gerar um índice válido para a Tabela, vamos definir o método `calculaIndiceDaTabela(String)`

```
private int calculaIndiceDaTabela(String placa) {
    return Math.abs(placa.hashCode()) % this.tabela.size();
}
```

12.2 - Operações

Vamos ver as operações em mapas.

12.3 - Verificando se uma chave existe

Como estamos utilizando a técnica de Espalhamento, para verificar se uma chave existe no Mapa, basta calcular o índice correto da Tabela e procurar na Lista correspondente.

```
public boolean contemChave(String placa) {
    int indice = this.calculaIndiceDaTabela(placa);
    List<Associacao> lista = this.tabela.get(indice);

    for (int i = 0; i < lista.size(); i++) {
        Associacao associacao = lista.get(i);
        if (associacao.getPlaca().equals(placa)) {
            return true;
        }
    }
    return false;
}
```

12.4 - Removendo uma associação dado uma chave

Este procedimento é simples, calculamos o índice e procuramos a chave na Lista correspondente. Ao achar a chave, removemos a associação, se a chave não for achada podemos lançar uma exceção ao usuário.

```
public void remove(String placa) {
    int indice = this.calculaIndiceDaTabela(placa);
    List<Associacao> lista = this.tabela.get(indice);

    for (int i = 0; i < lista.size(); i++) {
        Associacao associacao = lista.get(i);
        if (associacao.getPlaca().equals(placa)) {
            lista.remove(i);
            return;
        }
    }

    throw new IllegalArgumentException("A chave não existe");
}
```

12.5 - Adicionando uma associação dado uma chave

Ao adicionar um associação nova pode ser que a chave da mesma já exista no Mapa. Neste caso, vamos retirar a associação antiga antes de colocar a nova. Isso deve ser feito porque o Mapa não permite chaves repetidas.

```
public void adiciona(String placa, Carro carro) {
    if (this.contemChave(placa)) {
        this.remove(placa);
```

```
    }

    int indice = this.calculaIndiceDaTabela(placa);
    List<Associacao> lista = this.tabela.get(indice);
    lista.add(new Associacao(placa, carro));
}
```

12.6 - Recuperando o valor associado a uma determinada chave

O principal objetivo do Mapa é oferecer uma forma rápida de acessar o valor de uma chave dada.

Então, procuramos a associação pela chave na Lista adequada e devolvemos o valor correspondente. Se a chave não existe então lançamos uma exceção para o usuário.

```
public Carro pega(String placa) {
    int indice = this.calculaIndiceDaTabela(placa);
    List<Associacao> lista = this.tabela.get(indice);

    for (int i = 0; i < lista.size(); i++) {
        Associacao associacao = lista.get(i);
        if (associacao.getPlaca().equals(placa)) {
            return associacao.getCarro();
        }
    }

    throw new IllegalArgumentException("A chave não existe");
}
```

12.7 - Performance das operações

Utilizando a técnica de Espalhamento podemos obter um consumo de tempo médio constante para todas as operações. Isso é muito melhor do que obtivemos implementando Mapas com Listas.

Porém ainda precisaríamos tratar o problema da sobrecarga das Listas da Tabela de Espalhamento. Assim como fizemos no capítulo de Conjuntos com Espalhamento.

Deveríamos transformar a nossa Tabela de Espalhamento em uma tabela dinâmica, ou seja, uma tabela que aumenta e diminui de tamanho conforme a carga do Mapa.

12.8 - Generalização e Parametrização

O nosso Mapa está atrelado fortemente a associações entre `String` e `Carro`. Podemos generalizá-lo e parametrizá-lo para reutilizá-lo em diversas situações.

O primeiro passo é tornar as associações genéricas e parametrizadas.

```
public class Associacao<C, V> {
    private C chave;
    private V valor;

    public Associacao(C chave, V valor) {
        this.chave = chave;
```

```
        this.valor = valor;
    }

    public C getChave() {
        return chave;
    }

    public V getValor() {
        return valor;
    }

    @Override
    public String toString() {
        return "{" + this.chave + " -> " + this.valor + "}";
    }
}
```

Depois modificamos a classe `MapaEspalhamento`.

```
public class MapaEspalhamento<C, V> {

    private List<List<Associacao<C, V>>> tabela = new ArrayList<List<Associacao<C, V>>>();

    public MapaEspalhamento() {
        for (int i = 0; i < 100; i++) {
            this.tabela.add(new LinkedList<Associacao<C, V>>());
        }
    }

    public boolean contemChave(C chave) {
        int indice = this.calculaIndiceDaTabela(chave);
        List<Associacao<C, V>> lista = this.tabela.get(indice);

        for (int i = 0; i < lista.size(); i++) {
            Associacao<C, V> associacao = lista.get(i);
            if (associacao.getChave().equals(chave)) {
                return true;
            }
        }
        return false;
    }

    public void remove(C chave) {
        int indice = this.calculaIndiceDaTabela(chave);
        List<Associacao<C, V>> lista = this.tabela.get(indice);

        for (int i = 0; i < lista.size(); i++) {
            Associacao<C, V> associacao = lista.get(i);
            if (associacao.getChave().equals(chave)) {
                lista.remove(i);
                return;
            }
        }

        throw new IllegalArgumentException("A chave não existe");
```

```
    }

    public void adiciona(C chave, V valor) {
        if (this.contemChave(chave)) {
            this.remove(chave);
        }

        int indice = this.calculaIndiceDaTabela(chave);
        List<Associacao<C, V>> lista = this.tabela.get(indice);
        lista.add(new Associacao<C, V>(chave, valor));
    }

    public V pega(C chave) {
        int indice = this.calculaIndiceDaTabela(chave);
        List<Associacao<C, V>> lista = this.tabela.get(indice);

        for (int i = 0; i < lista.size(); i++) {
            Associacao<C, V> associacao = lista.get(i);
            if (associacao.getChave().equals(chave)) {
                return associacao.getValor();
            }
        }

        throw new IllegalArgumentException("A chave não existe");
    }

    private int calculaIndiceDaTabela(C chave) {
        return Math.abs(chave.hashCode()) % this.tabela.size();
    }

    private List<Associacao<C, V>> pegaTodas() {
        ArrayList<Associacao<C, V>> associacoes = new ArrayList<Associacao<C, V>>();
        for (List<Associacao<C, V>> lista : this.tabela) {
            associacoes.addAll(lista);
        }
        return associacoes;
    }

    @Override
    public String toString() {
        return this.pegaTodas().toString();
    }
}
```

Agora, quando utilizarmos a nossa implementação de Mapa podemos escolher o tipo das chaves e o tipo dos valores.

```
MapaEspalhamento<String, Carro> mapa = new MapaEspalhamento<String, Carro>();
```

12.9 - API do Java

Na biblioteca de coleções do Java existem algumas implementações de Mapa. Há duas que utilizam a técnica de Espalhamento (`HashMap` e `Hashtable`).

O uso destas classes é simples e bem parecido com o que fizemos aqui.

```
public class TesteHashMap {
    public static void main(String[] args) {

        HashMap<String, Carro> mapa = new HashMap<String, Carro>();
        mapa.put("abc1234", new Carro("a"));
        System.out.println(mapa);
        mapa.put("abc1234", new Carro("b"));
        System.out.println(mapa);
        mapa.put("def1234", new Carro("c"));
        System.out.println(mapa);

        System.out.println(mapa.containsKey("abc1234"));
        System.out.println(mapa.get("abc1234"));
        mapa.remove("abc1234");
        System.out.println(mapa);
    }
}
```