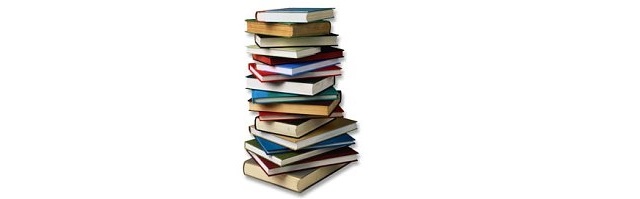
**Pilhas**

**6.1 - Introdução**

A pilha é uma das estruturas de dados e trabalha com o formato LIFO (o último a entrar é o primeiro a sair, “Last In, First Out”, em inglês). Lembre-se da pilha como uma pilha de livros, em que o primeiro livro que foi inserido na pilha, normalmente é o último que sai dela, enquanto o último adicionado é o primeiro a ser retirado, como vemos na **Figura 1**.

**Figura 1**. Pilhas em Java

A estrutura da pilha, segundo Farias “são estruturas de dados do tipo LIFO (last-in first-out), onde o último elemento a ser inserido, será o primeiro a ser retirado. Assim, uma pilha permite acesso a apenas um item de dados - o último inserido. Para processar o penúltimo item inserido, deve-se remover o último”.

A pilha é considerada uma estrutura de dados simples, sendo fácil de implementar. Em uma análise simples, poderia ser utilizada, por exemplo, em um carregamento de um caminhão, pois se o caminhão tiver 4 entregas, a última entrega colocada dentro do caminhão deve ser a primeira a sair, caso contrário, pode dar mais trabalho para descarregar.

Um outro caso de pilha simples de se entender é o caso do entregador de pizza.

O entregador deve entregar três pizzas em locais diferentes, se ele colocar na ordem de entrega, o que vai ocorrer é que a primeira pizza colocada no baú é a primeira pizza a ser entregue, de forma que todas as outras pizzas estão sobre a primeira pizza, então qual a melhor lógica?

Mudar a ordem: a primeira pizza no baú deve ser a última a ser entregue e a última pizza do baú, a primeira a ser entregue. Neste caso, ao chegar na casa do cliente, o entregador apenas pega a primeira pizza que está no baú e entrega ao cliente.

Este exemplo foi proposto inicialmente por Takai, apresentando uma rota de entrega de quatro pizzas, sendo a seguinte ordem:

1. 1° entrega: Portuguesa
2. 2° entrega: Frango com catupiry
3. 3° entrega: Calabresa
4. 4º entrega: Quatro queijos

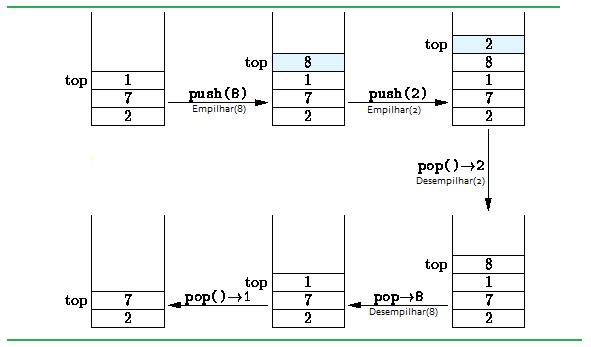
Assim, para armazenar no baú, a ordem deve ser invertida, ficando da seguinte forma:

* Portuguesa (topo do baú)
* Frango com catupiry
* Calabresa
* Quatro queijos

Para as tarefas temos: a criação da pilha, o empilhamento - push (ato de colocar uma caixa de pizza sobre a outra), o ato de desempilhar - pop (na hora que o entregador tira a caixa de pilha para entregar ao cliente), além de uma verificação se a pilha está cheia ou vazia (ato que o entregador faz ao verificar o baú).

**Pontos Principais de uma pilha**

* **PUSH, que insere um dado no topo da pilha.**
* **POP, que remove o item no topo da pilha.**
* **TOP, mostra o item no topo da pilha.**

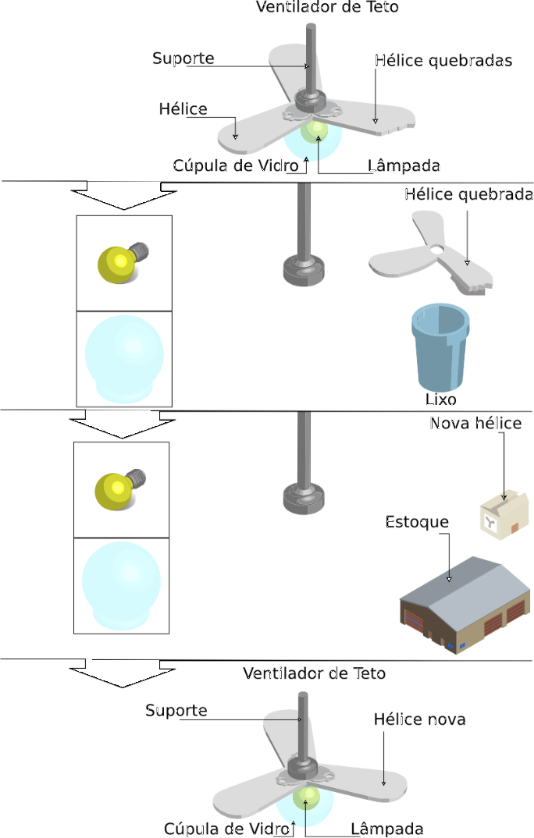
****

**Implementando nossa pilha em Java**

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.



*Figura 6.1: 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 problema 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 que 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 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 saberíamos 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 e mais 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 na 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: 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.

**6.4 - Operações em pilhas: 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.

**6.5 - Operações em pilhas: 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.

**6.6 - 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 classes 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 classe 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("Adalberto");

pilha2.insere("Maria");

String maria = pilha2.remove();

String adalberto = pilha2.remove();

System.out.println(maria);

System.out.println(adalberto);

}

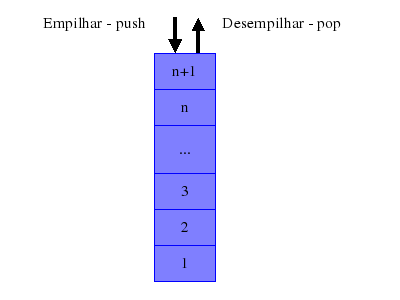
}

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.

**6.7 - API do Java**

**Pilhas - Stack**

São estruturas de dados do tipo LIFO (last-in first-out), onde o último elemento a ser inserido, será o primeiro a ser retirado. Assim, uma pilha permite acesso a apenas um item de dados - o último inserido. Para processar o penúltimo item inserido, deve-se remover o último.



São exemplos de uso de pilha em um sistema:

* Funções recursivas em compiladores;
* Mecanismo de desfazer/refazer dos editores de texto;
* Navegação entre páginas Web;
* etc.

A implementação de pilhas pode ser realizada através de vetor (alocação do espaço de memória para os elementos é contígua) ou através de listas encadeadas.

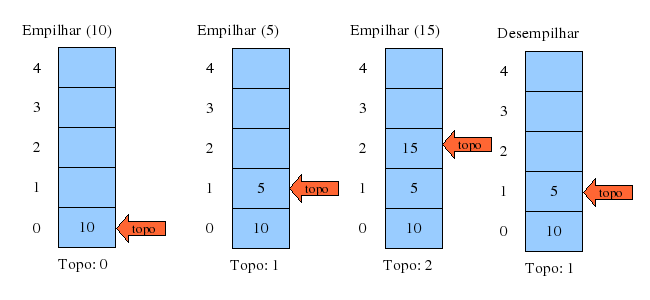
Numa pilha, a manipulação dos elementos é realizada em apenas uma das extremidades, chamada de topo, em oposição a outra extremidade, chamada de base.

Operações com Pilha:

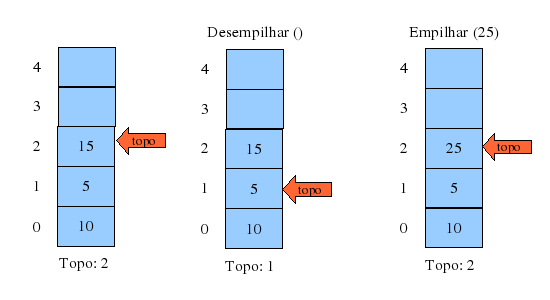
Todas as operações em uma pilha podem ser imaginadas como as que ocorre numa pilha de pratos em um restaurante ou como num jogo com as cartas de um baralho:

* criação da pilha (informar a capacidade no caso de implementação sequencial - vetor);
* empilhar (push) - o elemento é o parâmetro nesta operação;
* desempilhar (pop);
* mostrar o topo (top);
* verificar se a pilha está vazia (isEmpty);
* verificar se a pilha está cheia (isFull - implementação sequencial - vetor).

Supondo uma pilha com capacidade para 5 elementos (5 nós).



Na realidade a remoção de um elemento da pilha é realizada apenas alterando-se a informação da posição do topo.



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 LinkedList<> 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");

}

}

}

**Exercícios**

1. É possível implementar a nossa Pilha utilizando internamente uma ArrayList em vez de Stack? Teremos algum ganho ou perda no consumo de tempo de alguma das operações?
2. Uma mensagem é criptografada invertendo cada palavra do texto. O texto "Uma mensagem confidencial" criptografado fica "amU megasnem laicnedifnoc". Implemente a criptografia e a descriptografia de mensagem. Faça isso utilizando Pilha.
3. Fazer um programa que testa se uma sequência de caracteres fornecida pelo usuário é um palíndromo, ou seja, é uma palavra cuja primeira metade é simétrica à segunda metade.

Veja os exemplos:

AABCCBAA - sim

ADDFDDA - sim

ABFFBB – não

1. Fazer um programa que trate o seguinte menu:

  1 - Empilhar elemento

  2 - Desempilhar elemento

  3 - Mostrar o topo

  4 - Mostrar tudo

5 - Zerar a pilha

  6 - Sair

**6.8 - Escapando do Labirinto**

Ao percorrer um labirinto para fugir do Minotauro, existe um algoritmo bastante simples: você sempre escolhe 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!

**6.9 - Exercícios: Pilha**

1. Implemente a classe Peca

public class Peca {

private String nome;

public String getNome() {

return nome;

}

public void setNome(String nome) {

this.nome = nome;

}

}

Implemente a classe Pilha para peças vista neste capítulo.

import java.util.LinkedList;

import java.util.List;

import 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.

import 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.

Implemente a classe PilhaGenerica para objetos (genérica) vista neste capítulo.

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.

import.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 compilação. 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.

1. Implemente a classe PilhaParametrizada utilizando o recurso do **Generics**.

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:

Import 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("Manoel");

pilhaDeString.insere("Zuleide");

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

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

}

}

1. 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?
2. Uma mensagem é criptografada invertendo 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.
3. Implementar uma pilha genérica usando internamento a ListaSE genérica que você criou
4. Fazer um programa que trate o seguinte menu:

  1 - Empilhar elemento

  2 - Desempilhar elemento

  3 - Mostrar o topo

  4 - Imprimir tudo zerando a pilha

  5 - Sair

1. PROBLEMA 1: Fazer um programa que testa se uma sequência de caracteres fornecida pelo usuário é um palíndromo, ou seja, é uma palavra cuja primeira metade é simétrica à segunda metade. Veja os exemplos:

AABCCBAA - sim

ADDFDDA - sim

ABFFBB – não

**EXERCÍCIO 1 - CLASSE Node.java**

package util;

public class Node<T> {

public T valor;

public Node<T> next;

}

**EXERCÍCIO 1 - CLASSE ListaSE.java**

package util;

public class ListaSE<T> {

protected Node<T> head;

protected int size;

public ListaSE() {

size = 0;

head = null;

}

public boolean isEmpty() {

return size == 0;

}

public int size() {

return size;

}

public void inserirNaCabeca(Node<T> x) {

x.next = head;

size = size + 1;

head = x;

}

public Node<T> deletarPrimeiro() {

Node<T> aux = head;

if (!isEmpty()) {

head = head.next;

size = size - 1;

}

return aux;

}

public Node<T> buscar(T x) {

Node<T> p = head;

while (p!=null && !x.equals(p.valor)) {

p = p.next;

}

return p;

}

public Node<T> anterior(Node<T> v) {

if (v == head) {

return null;

}

else {

Node<T> p = head;

while (p.next != v) {

p = p.next;

}

return p;

}

}

public String toString() {

String s = "[";

Node<T> p = head;

while (p!=null) {

s = s + p.valor.toString();

p = p.next;

if (p!=null) {

s = s + "; ";

}

}

s = s + "]";

return s;

}

}

**EXERCÍCIO 2 - CLASSE Pilha.java:**

package util;

public class Pilha<T> {

private ListaSE<T> lista;

public Pilha() {

lista = new ListaSE<T>();

}

public boolean isEmpty() {

return lista.isEmpty();

}

public int size() {

return lista.size();

}

public void push(T valor) {

Node<T> aux = new Node<T>();

aux.valor = valor;

lista.inserirNaCabeca(aux);

}

public T pop() {

Node<T> aux = lista.deletarPrimeiro();

return aux.valor;

}

public T topo() {

return lista.head.valor;

}

}

**EXERCÍCIO 3 - CLASSE Exercicio3.java:**

package programa;

import java.util.Scanner;

import util.Pilha;

public class Exercicio3 {

public static void main(String[] args) {

Scanner sc = new Scanner(System.in);

int op;

Pilha<String> p = new Pilha<String>();

do {

System.out.println("\n1 - Empilhar elemento");

System.out.println("2 - Desempilhar elemento");

System.out.println("3 - Mostrar topo");

System.out.println("4 - Imprimir tudo zerando a pilha");

System.out.println("5 - Sair");

System.out.print("Digite uma opção: ");

op = Integer.parseInt(sc.nextLine());

if (op==1) {

System.out.print("\nDigite um String a ser empilhado: ");

String aux = sc.nextLine();

p.push(aux);

}

if (op==2) {

if (p.isEmpty()) {

System.out.println("A pilha está vazia!");

}

else {

String aux = p.pop();

System.out.println("Elemento desempilhado: " + aux);

}

}

if (op==3) {

if (p.isEmpty()) {

System.out.println("A pilha está vazia!");

}

else {

String aux = p.topo();

System.out.println("Elemento do topo: " + aux);

}

}

if (op==4) {

System.out.println("Desempilhando tudo:");

while (!p.isEmpty()) {

String aux = p.pop();

System.out.println(aux);

}

}

} while (op != 5);

}

}