

Estrutura de dados

Conteúdo

- Estrutura de dados: pilha dinâmica encadeada



Autores

Prof. Manuel F. Paradela **Ledón**

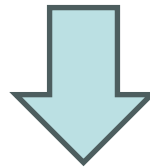
Profª Cristiane P. Camilo Hernandez

Prof. Amilton Souza Martha

Prof. Daniel Calife

Reflexões sobre alocação dinâmica de memória (1)

```
Trabalhador trab1 = new Trabalhador("Bruno Costa", 6290);  
Trabalhador trab2 = new Trabalhador("Ana Lopes", 6500);
```



Com as referências `trab1` e `trab2` podemos acessar os atributos (dados privados e seus métodos get/set públicos associados) e os restantes métodos públicos da classe `Trabalhador`.

Reflexões sobre alocação dinâmica de memória (2)

Trabalhador trab1;

//erro porque a "variable trab1 might not have been initialized":

//trab1.setNome("Ana Lopes");

//trab1.setSalario(3500);

//utilizando o construtor sem parâmetros

Trabalhador trab2 = new Trabalhador();

trab2.setNome("Ana Lopes"); //com a referência trab2 executamos setNome

trab2.setSalario(6500); //com a referência trab2 executamos setSalario

System.out.println(trab2.toString()); //com a ref. trab2 executamos toString

//utilizando o construtor com parâmetros

Trabalhador trab3 = new Trabalhador("Bruno Costa", 6290);

System.out.println(trab3.toString()); //com a ref. trab3 executamos toString

```
run:
```

```
Ana Lopes, salario: R$6500.0
```

```
Bruno Costa, salario: R$6290.0
```

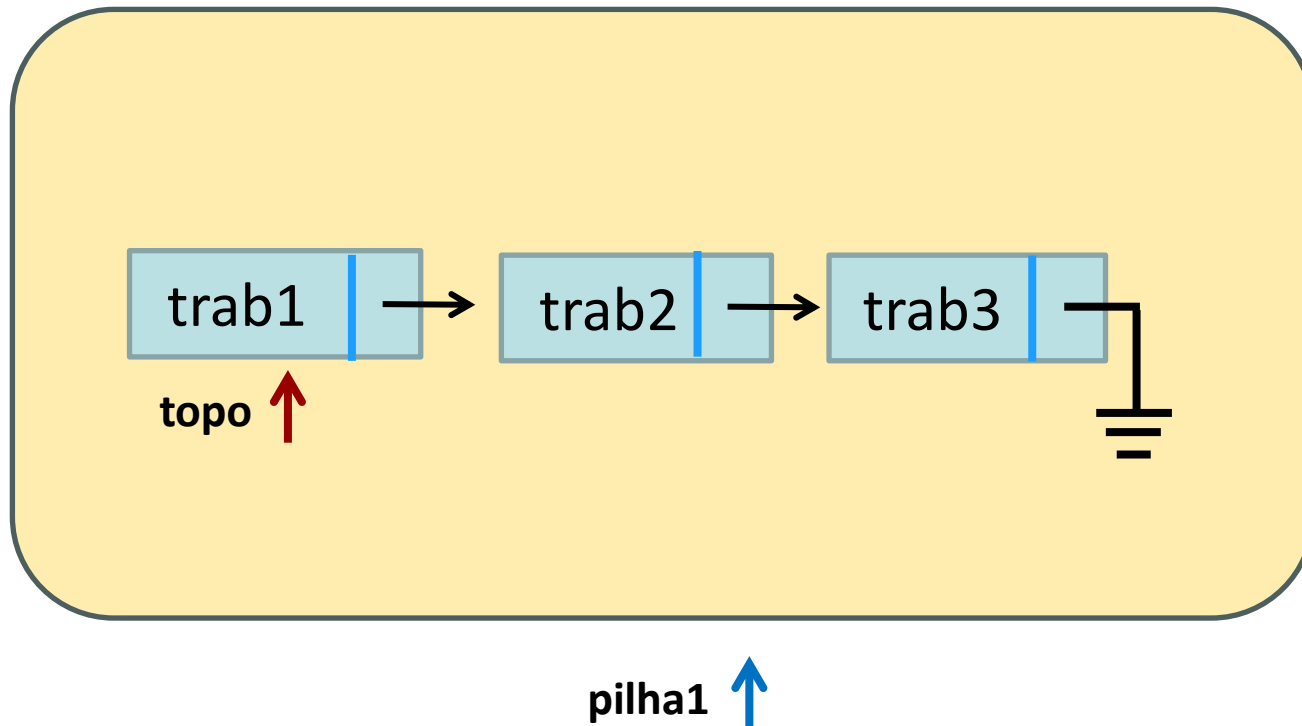
```
BUILD SUCCESSFUL (total time: 0 seconds)
```

Reflexões sobre alocação dinâmica de memória (3)

```
//se não fornecemos um método toString() e executamos:  
Trabalhador trab2 = new Trabalhador("Ana Lopes", 6500);  
System.out.println(trab2);
```

```
//será mostrado na tela algo assim:  
testesalocdinamica.Trabalhador@5a39699c
```

Uma pilha com alocação dinâmica de memória e dados encadeados



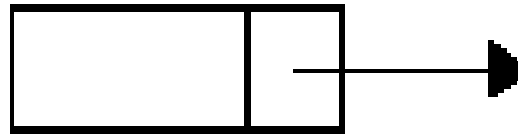
A memória da pilha, dos nodos encadeados (enlaçados) da pilha e dos objetos da classe Trabalhador que guardamos nesta estrutura de dados será alocada dinamicamente!

Pilhas dinâmicas

- Um problema encontrado nas pilhas estáticas é a limitação da inserção de elementos na pilha, **limitada pelo tamanho do vetor declarado**.
- Tanto em termos de superdimensionamento quanto subdimensionamento, uma **pilha dinâmica** faz uma requisição à **heap*** toda vez que um novo elemento precisa ser inserido na pilha, assim como desaloca (libera) a memória usada por um elemento que foi excluído da pilha.
- Com isso, usamos somente a quantidade de memória exata que o programa necessita.

*Memória Heap (bulto, pacote) é onde os objetos ficam de forma não organizada. Understanding memory management:
https://docs.oracle.com/cd/E13150_01/jrockit_jvm/jrockit/geninfo/diagnos/garbage_collect.html#wp1086087

- Para implementar uma **pilha dinâmica encadeada** (enlaçada, ligada), vamos pensar que cada elemento da pilha será um **nó** ou **nodo**, sendo que teremos acesso somente ao topo da pilha (estrutura com comportamento LIFO) e cada nó 'conhece' o endereço do próximo nó.

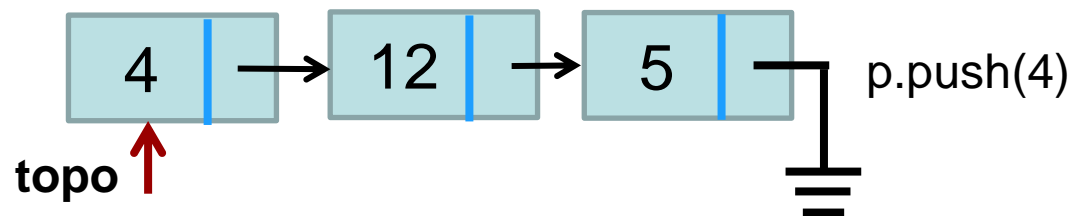
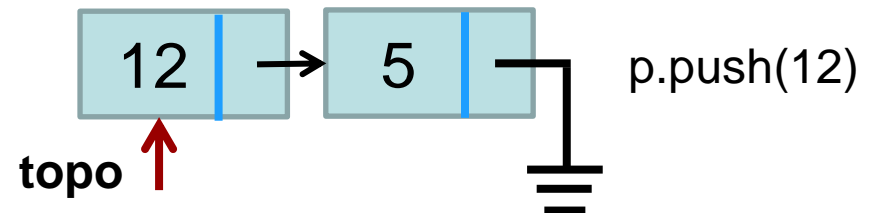
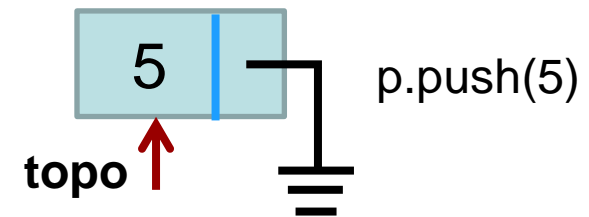


- Esta pilha dinâmica usa organização encadeada dos itens, ou seja, os elementos não necessariamente estão dispostos na ordem física da memória, portanto, **cada nodo deverá conter o endereço do próximo elemento da pilha.**

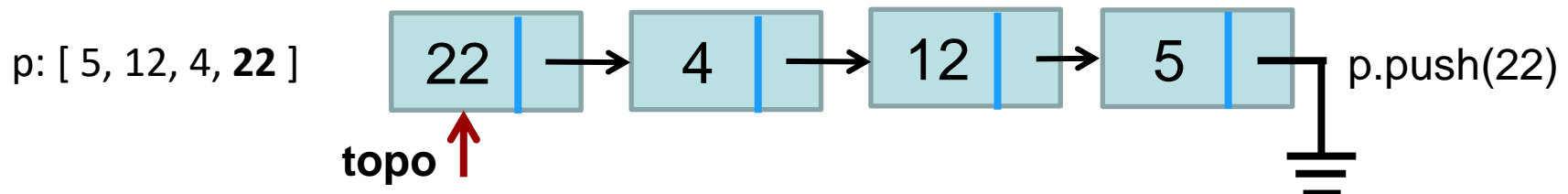
Exemplo gráfico

Inserir em uma pilha dinâmica encadeada *p* (inicialmente vazia) os objetos inteiros 5, 12, 4 e 22, nesta ordem. A cada inserção (operação **push**) o topo será modificado, de forma a apontar para o novo objeto inserido.

topo=null (pilha *p* vazia)

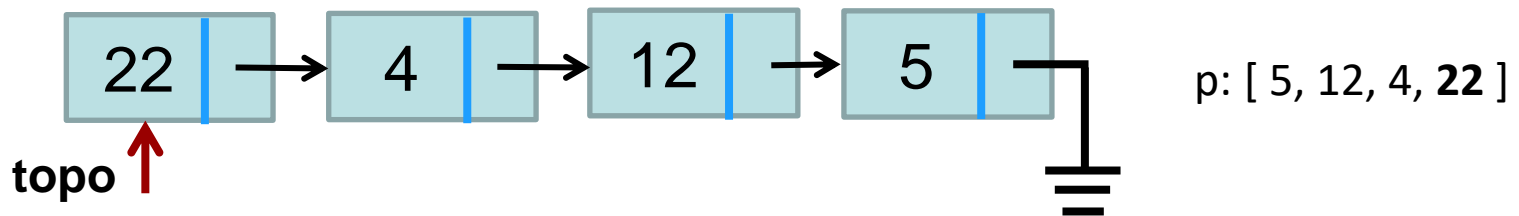


p.push()

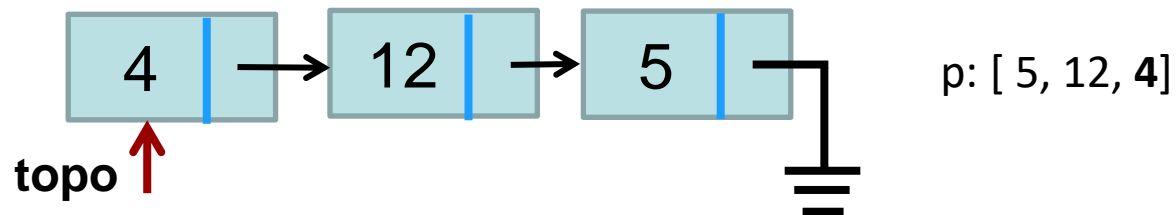


Exemplo gráfico

Retirar um elemento de uma pilha dinâmica encadeada *p* (operação **pop**). O objeto no topo (22 na figura) será eliminado da pilha e retornado para processamento posterior. O topo avançará e apontará para o próximo nodo, que guarda o objeto 4 no exemplo abaixo.

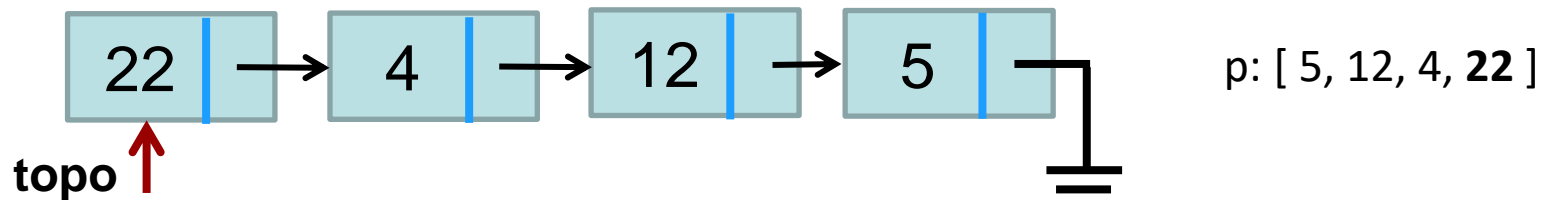


p.pop()



Exemplo gráfico

Retornar, sem eliminar, o elemento que se encontra no topo de uma pilha dinâmica encadeada *p* (operação **top**). O objeto no topo (22 nesta figura) será retornado. O topo não será alterado, continuará apontando para o nodo que guarda o objeto inteiro 22.



p.top()

(operação que não altera a pilha)

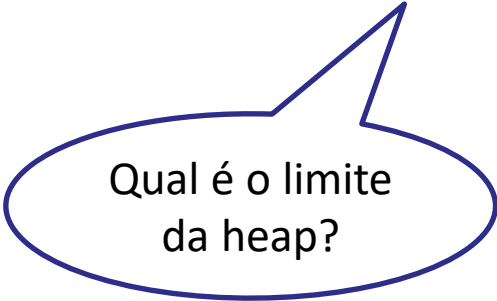
```
public class Node {  
    private Object value; // valor do nodo (um objeto guardado na pilha)  
    private Node next; // enlace, endereço para acessar o próximo item  
  
    public Object getValue() { // retorna o valor do nodo  
        return value;  
    }  
  
    public void setValue(Object value) { // para alterar o valor do nodo  
        this.value = value;  
    }  
  
    public Node getNext() { // retorna o endereço do próximo item  
        return next;  
    }  
  
    public void setNext(Node next) { // para alterar o endereço do próximo nodo  
        this.next = next;  
    }  
}
```



Um nodo (classe Node)

Implementaremos esta classe **Node** que descreve um **nó (nodo)** da pilha. Para cada inserção alocaremos espaço para mais um nó, ou seja, cada elemento da pilha é um nó, que guarda um objeto e um endereço de memória.

- Como em uma pilha só manipulamos uma extremidade denominada **topo**, precisamos ter a referência somente de apenas um nó.
- Como a alocação será dinâmica, não teremos a implementação de **isFull** (é opcional), pois a pilha não estará "cheia" e sim não teremos mais memória na *heap* para alocar.



Qual é o limite da heap?

Curiosidade: teste da heap (overflow)

```
public void PilhaTesteMemoria () {  
    int q=1;  
    Pilha pilha = new Pilha(); // cria uma pilha vazia  
    while( true ) {  
        Object obj = pilha.push("objeto string");  
        if(obj == null) { //se provocou erro de memória insuficiente  
            System.out.println("Overflow depois de " + q + " inserções na pilha");  
            break;  
        }  
        q++;  
    }  
}
```

Exemplo:

Overflow depois de 83.971.716 inserções na pilha
(em um PC comum, com 8GB de RAM)

Implementação de um pilha dinâmica encadeada

```
// A classe Pilha implementa uma pilha dinâmica encadeada

class Pilha implements TAD_Pilha {

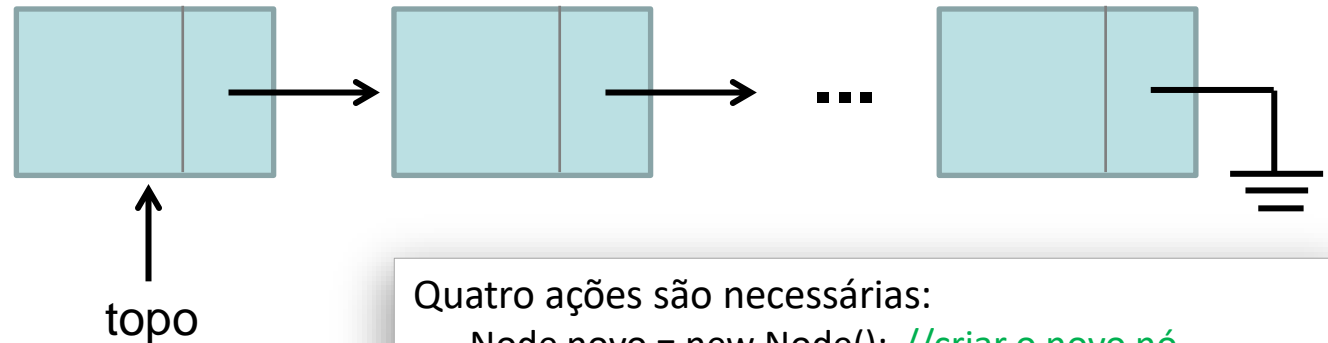
    private Node topo = null;

    public Pilha() {
        topo = null;
    }

    public boolean isEmpty() { //verifica se a pilha está vazia
        return (topo == null);
        // ou também:
        //      if(topo == null) return true; else return false;
    }
}
```

A operação **inserir** na pilha dinâmica (push)

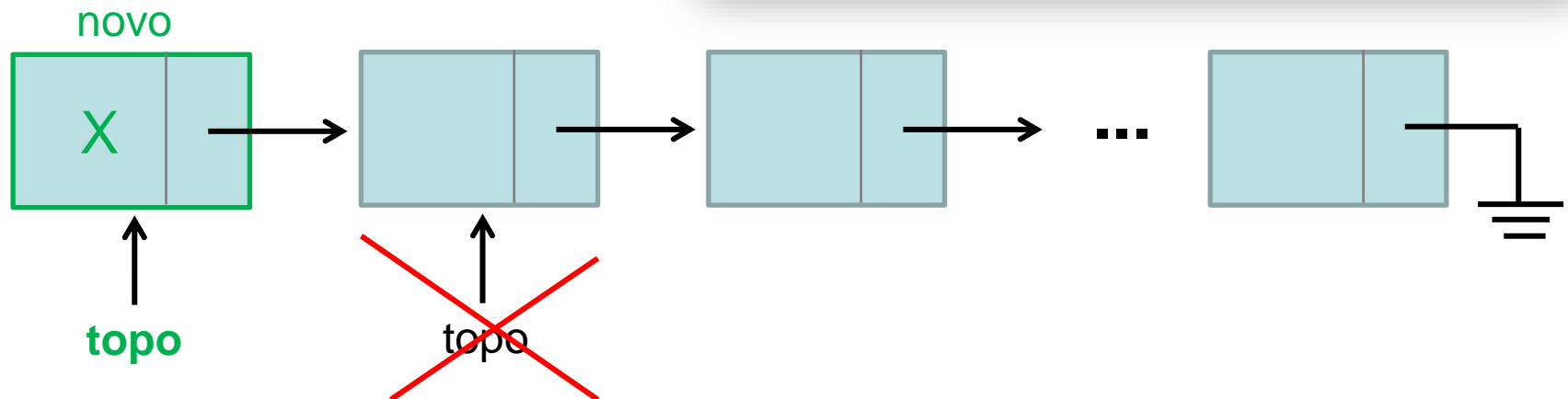
antes:



Quatro ações são necessárias:

```
Node novo = new Node(); //criar o novo nó  
novo.setValue(X); //colocar o objeto X no novo nó  
novo.setNext(topo); //enlaçar o novo nó  
topo = novo; //colocar o topo na nova posição
```

depois:

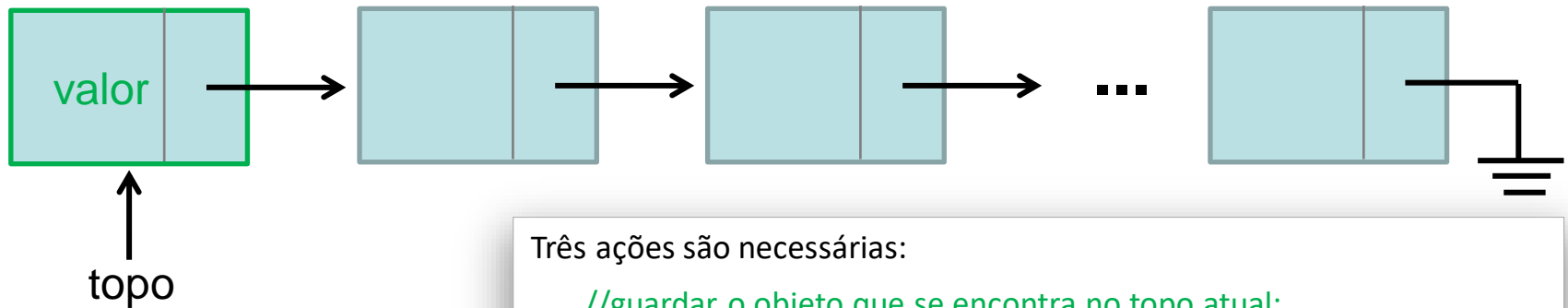


```
public Node push(Object x) {  
    try {  
        if(x == null) return null; //não permitimos um novo objeto x nulo  
        if(Runtime.getRuntime().freeMemory() < x.toString().length() + 1024) return null;  
        Node novo = new Node(); //alocamos memória para um novo nodo  
        novo.setValue(x); // atribuímos valor para o novo nó  
        novo.setNext(topo); // no caso de pilha vazia (topo == null) também funciona  
        topo = novo;  
        return novo;  
    } catch(Exception ex) {  
        return null; // memória insuficiente ou qualquer outro erro  
    }  
}
```

p.push()

A operação **retirar** da pilha dinâmica (pop)

antes:



Três ações são necessárias:

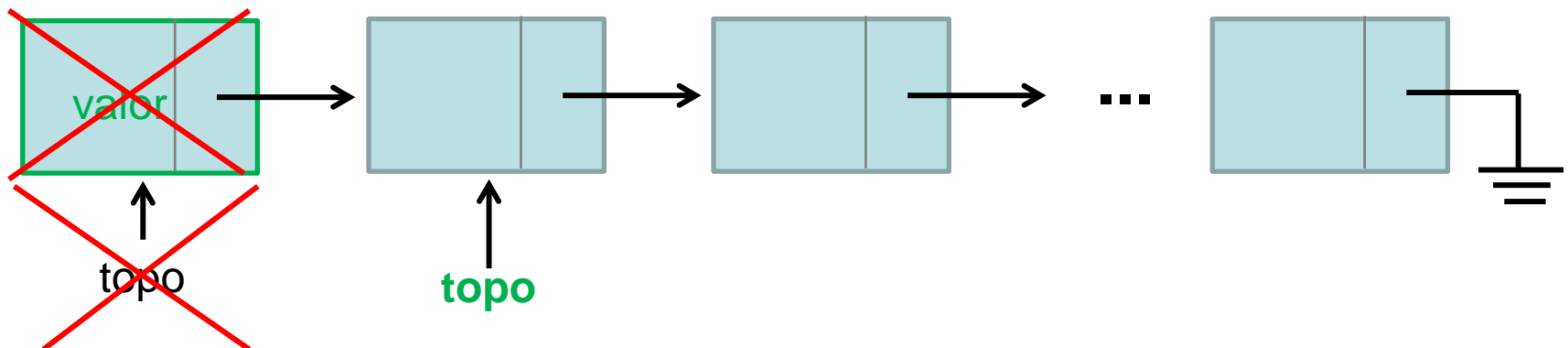
//guardar o objeto que se encontra no topo atual:

Object valor = topo.getValue();

topo = topo.getNext(); //mover o topo para o próximo nó

return valor; // retornar o objeto que estava no antigo topo

depois:

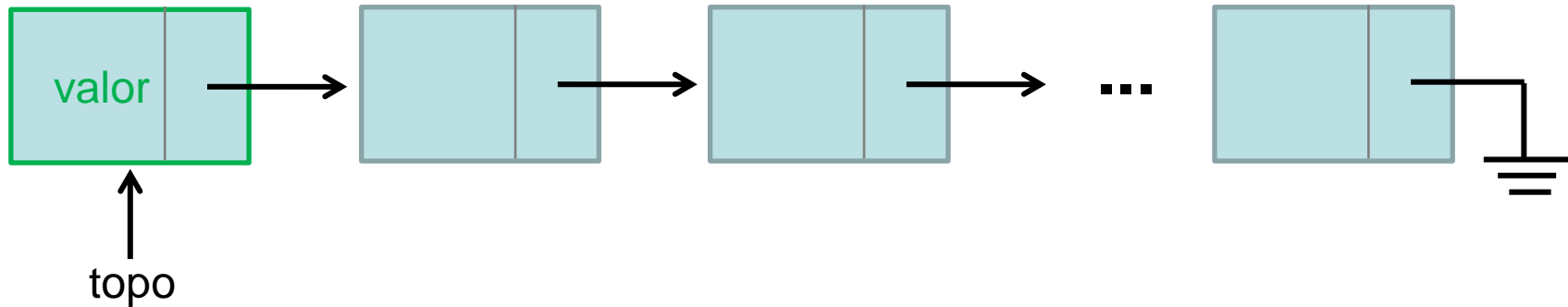


```
public Object pop() {  
    if (topo == null) return null; //se a pilha estiver vazia, retornamos null  
    Object valor = topo.getValue(); //pegamos o objeto no topo da pilha  
    topo = topo.getNext(); //avançar o topo para o próximo da pilha  
    return valor; //retornamos o objeto que estava no topo  
}
```

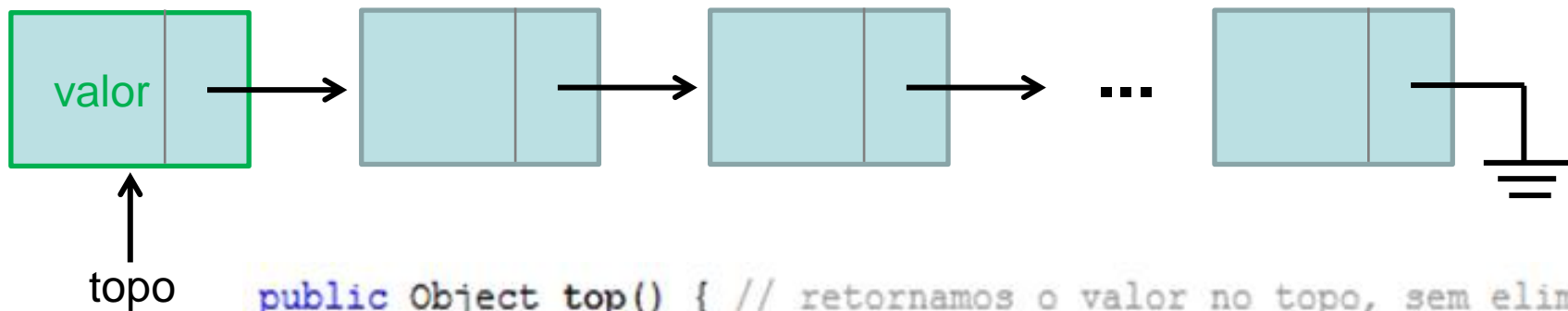
p.pop()

A operação (top)

antes:

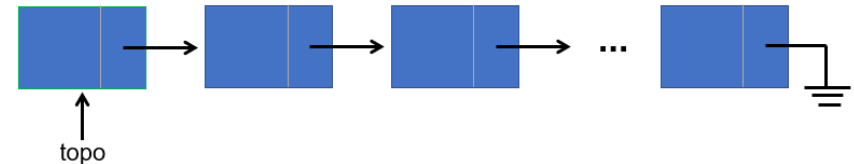


depois: (a pilha não será alterada)



```
public Object top() { // retornamos o valor no topo, sem eliminá-lo
    if(topo == null) return null; else return topo.getValue();
    // ou: if(isEmpty()) return null; else return topo.getValue();
}
```

```
public String toString() {
//Este método retorna os itens guardados na pilha, com a convenção P: [ a, b, c, topo ]
    if( !isEmpty() ) {
        String resp = "";
        Node aux = topo;
        while(aux!=null) {
            resp = aux.getValue().toString() + resp; //adicionamos em ordem invertida
            aux = aux.getNext(); //avançar o ponteiro aux
            if(aux != null) resp = ", " + resp;
        }
        return ( "P: [ " + resp + " ]" );
    }
    else return ( "Pilha Vazia!" );
}
```



Observações

- toString() retornará uma String com os objetos na pilha, no formato p: [a, b, c, d, **topo**]
- todos os métodos (operações) anteriores tem **O(1)**, mas esta operação toString() é de **O(n)**.

Exemplos completos resolvidos

Um exemplo completo de implementação de pilha dinâmica encadeada, se encontra no projeto NetBeans na pasta e arquivo **PilhaEncadeada.zip**.

Outro exemplo completo de implementação de pilha dinâmica enlaçada, utilizando **genéricos**, se encontra no projeto NetBeans na pasta e arquivo **PilhaComGenericos.zip**. Mas este assunto de genéricos será estudado em outra atividade posterior na disciplina.

O exemplo em **PilhaEncadeada2.zip** acrescenta na classe **Pilha** um contador de objetos `private int count`, uma função `public int size()` que retorna a quantidade de objetos guardados na pilha. Também, acrescenta a função `public Object [] toArray()` que retorna um vetor com os objetos guardados na pilha, o que poderá ser útil para ordenar ou processar os objetos em geral, mas sem alterar a pilha.

Exercício para praticar e entregar



Implemente uma **pilha dinâmica encadeada** que guarde elementos (objetos) da classe **Trabalhador** utilizada em aulas anteriores. Implemente, as ações a seguir (o ideal seria um programa Java SE com interface gráfica para cadastrar, consultar, retirar etc.):

- Inserir trabalhadores na pilha.
- Listar os trabalhadores guardados na pilha.
- Retirar um objeto trabalhador da pilha e mostrar seus dados.
- Recuperar os elementos guardados na pilha e guarda-los em um vetor, sendo que os mesmos devem permanecer na pilha. Sugestão: utilize a operação `toArray()` que se encontra implementada na classe `Pilha` do projeto `PilhaEncadeada2`. Ordene (pelos nomes) os elementos do vetor anterior utilizando o método de ordenação `Bubble Sort` ou qualquer outro e, por último, mostre na tela os objetos já ordenados.

Bibliografia (oficial) para a disciplina

BIBLIOGRAFIA BÁSICA	BIBLIOGRAFIA COMPLEMENTAR
<p>CORMEN, T. H.; et al. Algoritmos: teoria e prática. 3. ed. Rio de Janeiro: Elsevier, 2012.</p> <p>GOODRICH, M. T.; TAMASSIA, R. Estrutura de dados e algoritmos em java. 5. ed. Porto Alegre: Bookman, 2013. (livro físico e e-book)</p> <p>CURY, T. E., BARRETO, J. S., SARAIVA, M. O., et al. Estrutura de Dados (1. ed.) ISBN 9788595024328, Porto Alegre: SAGAH, 2018 (e-book)</p>	<p>ASCENCIO, A. F. G.; ARAÚJO, G. S. Estruturas de Dados: algoritmos, análise da complexidade e implementações em JAVA e C/C++. São Paulo: Pearson Prentice Hall, 2010. (eBook)</p> <p>PUGA, S.; RISSETTI, G. Lógica de programação e estruturas de dados, com aplicações em Java. 3. ed. São Paulo: Pearson Education do Brasil, 2016. (eBook)</p> <p>DEITEL, P.; DEITEL, H. Java como programar. 10. ed. São Paulo: Pearson Education do Brasil, 2017. (eBook)</p> <p>BARNES, D. J.; KOLLING, M. Programação Orientada a Objetos com Java: uma introdução prática usando o Blue J. São Paulo: Pearson Prentice Hall, 2004. (eBook)</p> <p>BORIN, V. POZZOBON. Estrutura de Dados. ISBN: 9786557451595, Edição: 1ª. Curitiba: Contentus, 2020 (e-book)</p>