

Estrutura de dados

Conteúdo

- Estrutura de dados pilha dinâmica encadeada



Autores

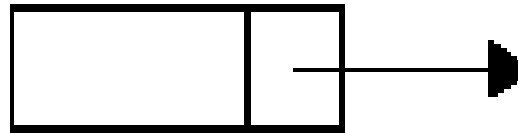
Prof. Manuel F. Paradela **Ledón**
Profª Cristiane P. Camilo Hernandez
Prof. Amilton Souza Martha
Prof. Daniel Calife

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

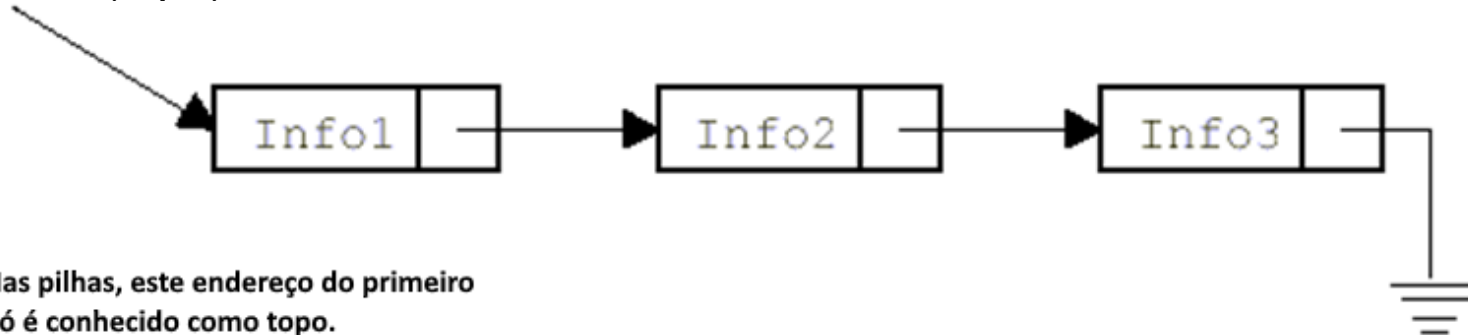
- Para implementar uma **pilha dinâmica encadeada** (enlaçada, ligada), vamos pensar que cada elemento da Pilha será um **nó** ou **nodo**, sendo que teremos acesso somente ao topo da pilha (estrutura com comportamento LIFO) e cada nó 'conhece' o endereço do próximo nó.



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

- Para isso, implementaremos uma classe que descreve um **nó** (**nodo**) da Pilha. Para cada inserção alocaremos espaço para mais um nó, até o limite da heap, ou seja, cada elemento da Pilha é um nó.
- Para a implementação de uma Pilha Dinâmica, vamos primeiramente entender como funciona uma **estrutura encadeada**.

prim (topo)



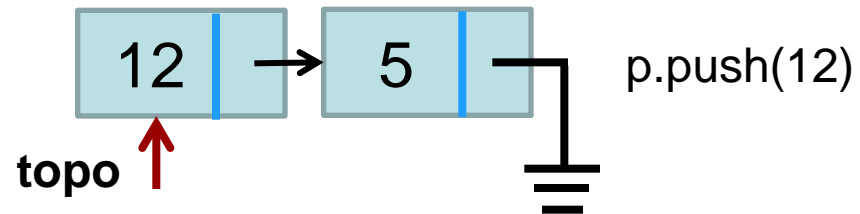
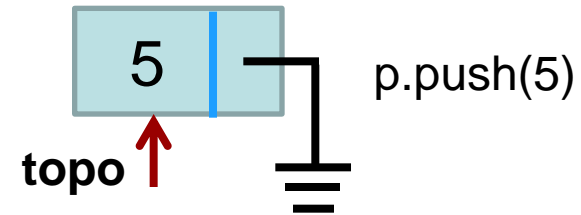
Nas pilhas, este endereço do primeiro nó é conhecido como topo.

Null (fim da lista)

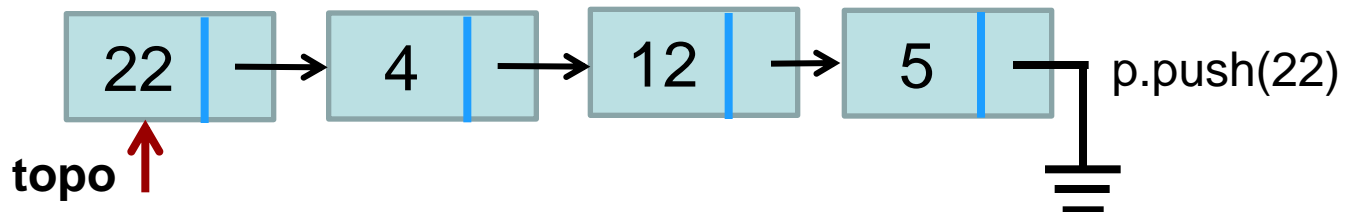
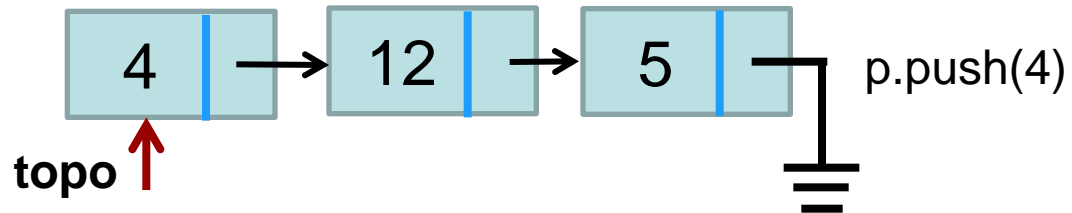
Exemplo

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)

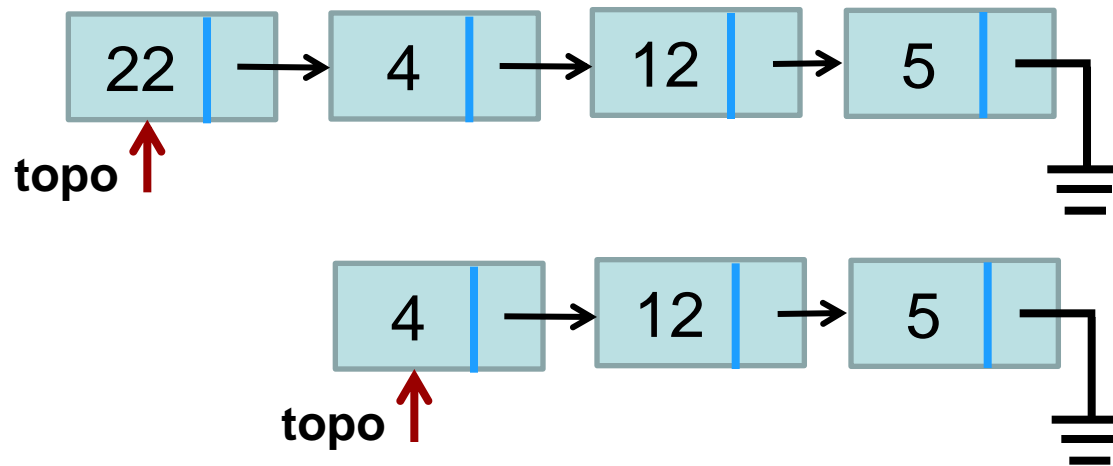


p.push()



Exemplo

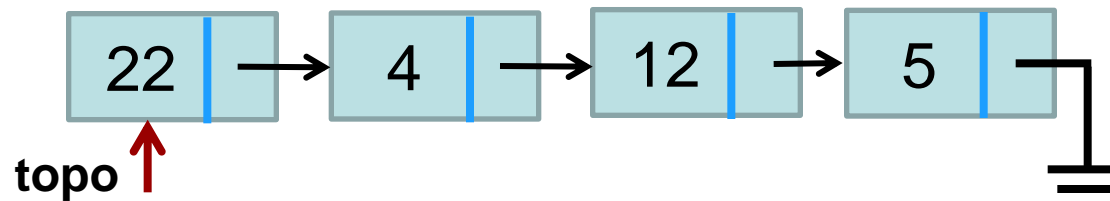
Retirar um elemento de uma pilha dinâmica encadeada *p* (operação **pop**). O objeto no topo (22) será eliminado da pilha e retornado. O topo avançará e apontará para o objeto inteiro, que é o 4 no exemplo.



p.pop()

Exemplo

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 aterado, continuará apontando para o nodo com o objeto inteiro 22.



p.top()

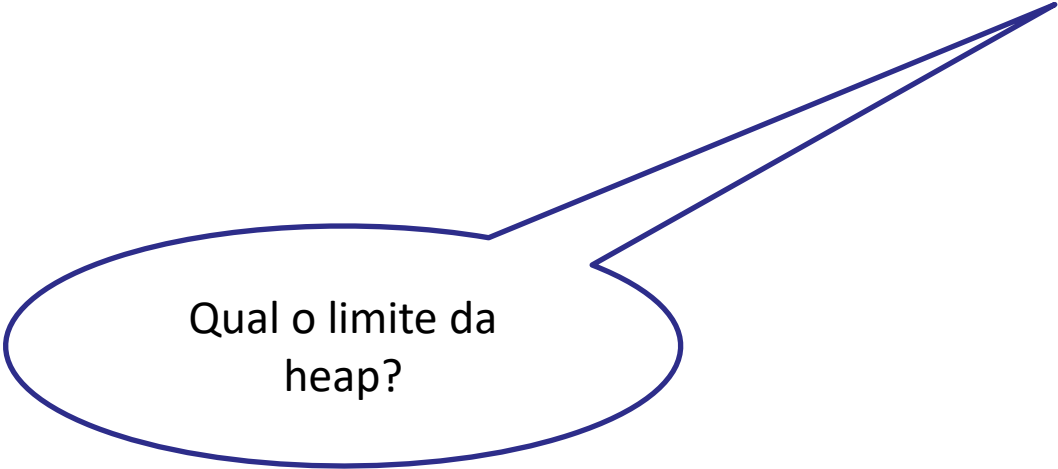
(operação que não altera a pilha)

```
public class Node {  
    private Object value; // valor do nodo  
    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 nodo  
        this.next = next;  
    }  
}
```



Um nodo (classe Node)

- Como na 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*, pois a Pilha (ou o vetor) não estará "cheia" e sim não teremos mais memória na *heap* para alocar.



Qual o limite da
heap?

Curiosidade: teste da heap (overflow)

```
public class testePilhaDin{  
    public static void main(String args[]){  
        int cont=0;  
        try{  
            PilhaDin P = new PilhaDin();  
            while(true){  
                P.push("Teste");  
                P.print();  
                cont++;  
            }  
        }  
        catch(OutOfMemoryError e){  
            System.out.println("Com "+cont+" registros deu o erro "+e.toString());  
        }  
    }  
}
```

Implementação de um pilha dinâmica encadeada

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

class Pilha implements TAD_Pilha {

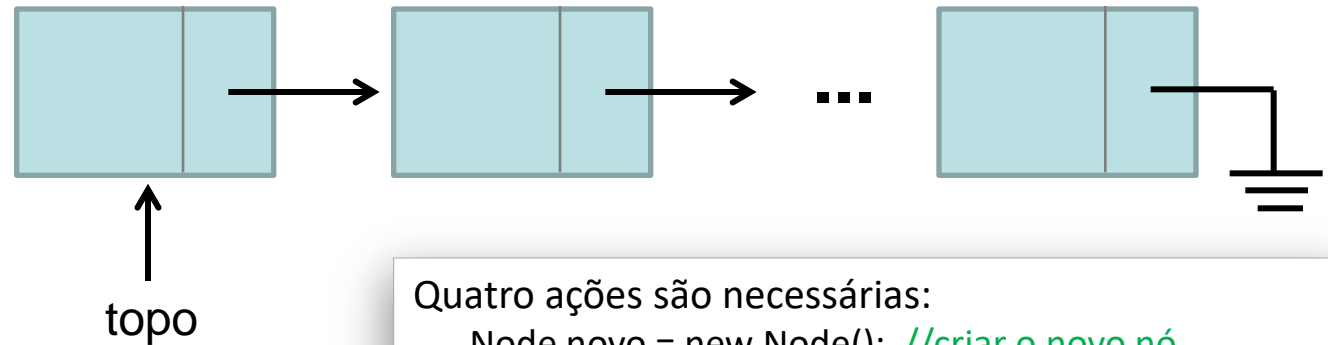
    private Node topo = null;

    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)

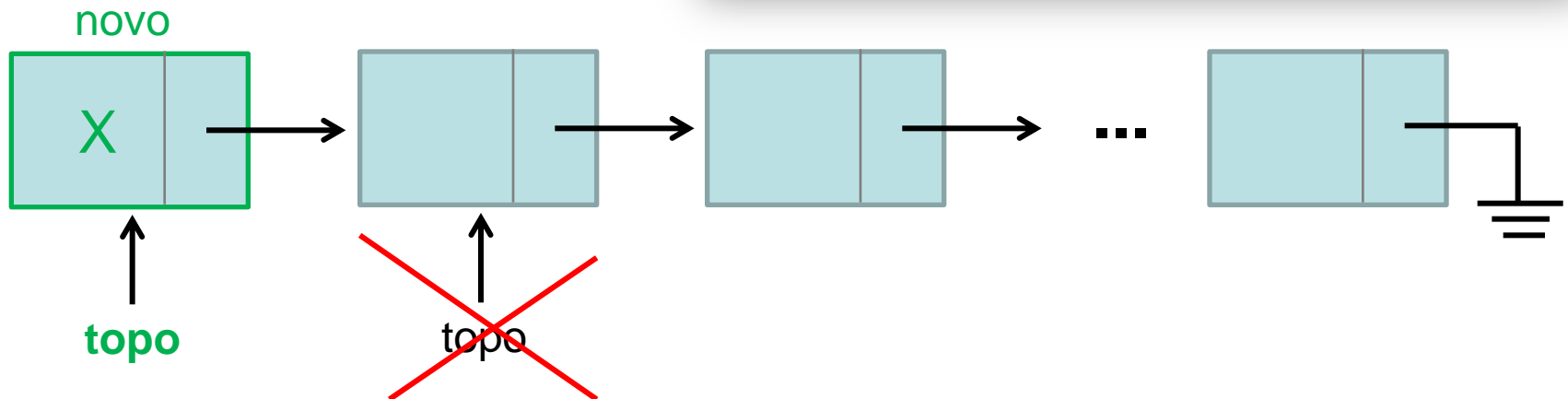
antes:



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:

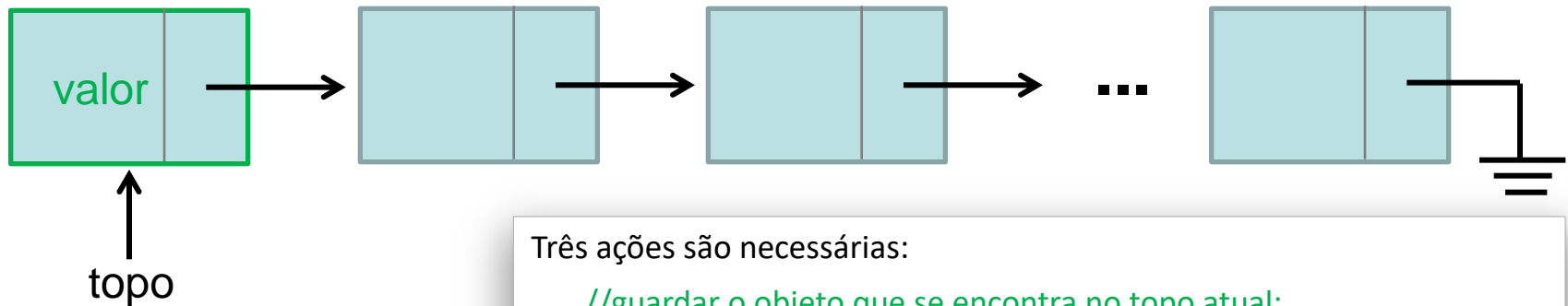


```
public Node push(Object x) {  
    try {  
        if(x == null) return null; //não permitimos novo objeto nulo  
        Node novo = new Node(); //alocamos memória para um novo nodo  
        novo.setValue(x); // atribuímos valor para o novo nó  
        novo.setNext(topo); // no caso de pilha vazia (topo == null) também funciona  
        topo = novo;  
        return novo;  
    } catch(Exception ex) {  
        return null; // memória insuficiente  
    }  
}
```

p.push()

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

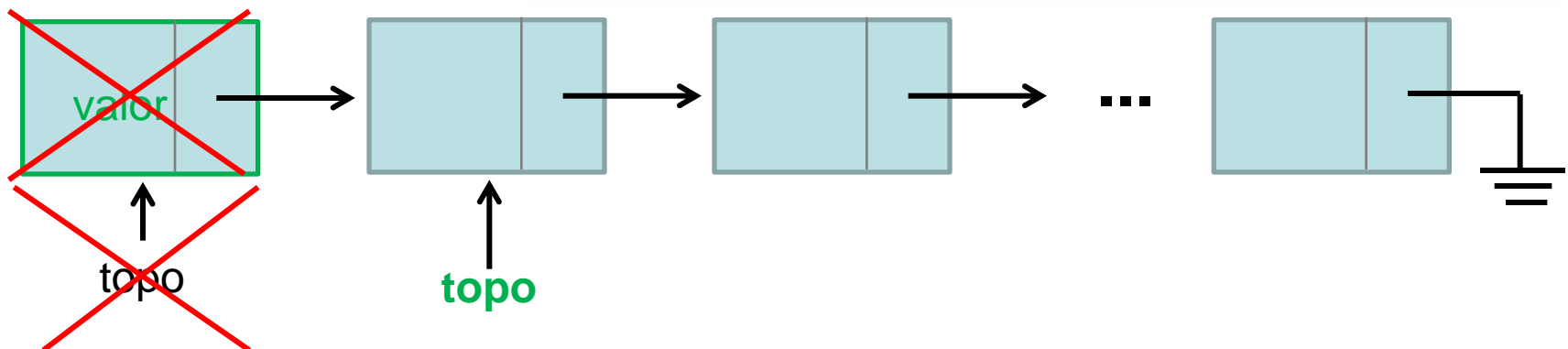
antes:



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:

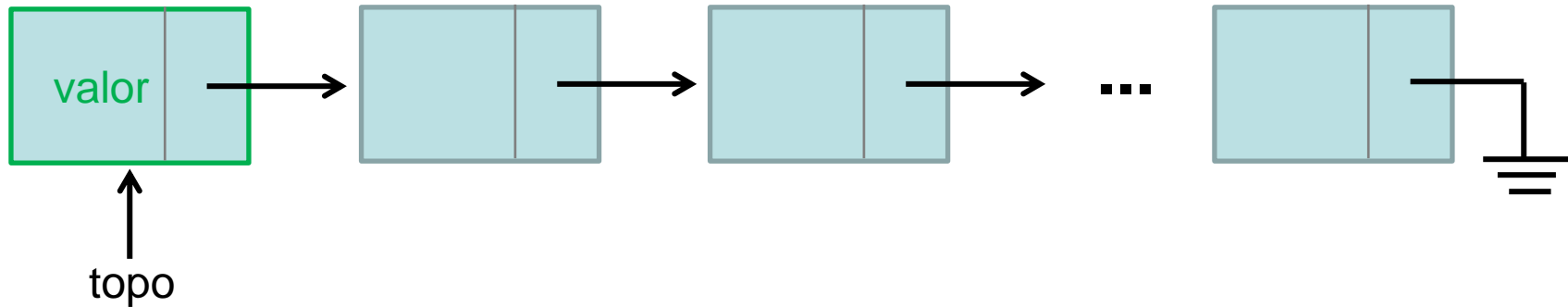


```
public Object pop() {  
    if (topo == null) return null; // se a pilha estiver vazia retornamos null  
    Object valor = topo.getValue();  
    Node temp = topo; // isto é opcional  
    topo = topo.getNext(); // avançar o topo para o próximo da pilha  
    temp = null; // isto é opcional  
    return valor; // retornamos o valor do elemento que estava no topo  
}
```

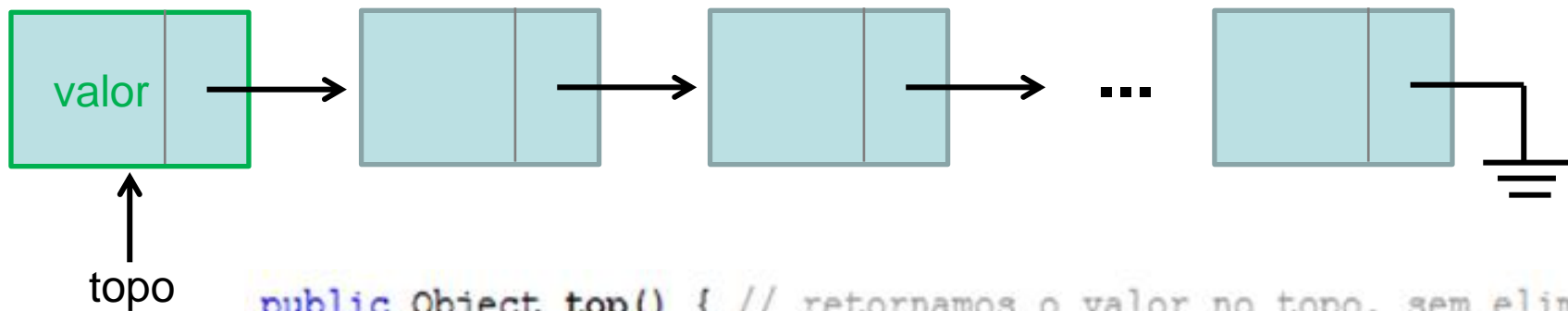
p.pop()

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;  
            aux = aux.getNext();  
            if(aux != null) resp = ", " + resp;  
        }  
        return ( "P: [ " + resp + " ]" );  
    }  
    else return ( "Pilha Vazia!" );  
}
```

Obs: 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 zip **PilhaEncadeada**.

Outro exemplo completo de implementação de pilha dinâmica enlaçada, utilizando **genéricos**, se encontra no projeto NetBeans na pasta e arquivo zip **PilhaComGenericos**.

Mais um exemplo completo, de ordenação utilizando genéricos, se encontra no projeto NetBeans na pasta **OrdenacaoBubbleSortGenericos**.

Exercício para praticar e entregar

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

- Inserir (vários) trabalhadores na pilha.
- Listar os trabalhadores guardados na pilha.
- Retirar um objeto trabalhador da pilha.
- Extrair os elementos guardados na pilha e visualizá-los na ordem em que foram extraídos. Vá guardando os elementos extraídos em um vetor.
- Ordene (pelos nomes) os elementos no vetor anterior utilizando o método de ordenação Bubble Sort.
- Por último, mostre na tela os objetos já ordenados.

Bibliografia para a disciplina

BIBLIOGRAFIA BÁSICA

CORMEN, Thomas H.; CORMEN, Thomas H.; LEISERSON, Charles E.; RIVEST, Ronald L.; STEIN, Clifford. Algoritmos: teoria e prática. Rio de Janeiro: Elsevier, Campus, 2002.

GOODRICH, Michael T.; TAMASSIA, Roberto. Estruturas de dados e algoritmos em Java. 2. ed. Porto Alegre: São Paulo: Bookman, 2002.

PREISS, B. R. Estruturas de Dados e Algoritmos: Padroes de Projetos Orientados a Objetivos Com Java. Rio de Janeiro: Campus, 2001.

BIBLIOGRAFIA COMPLEMENTAR

ASCENCIO, A. F. G.; ARAÚJO, G. S. Estruturas de Dados. São Paulo: Pearson, 2011. [eBook]

EDELWEISS, N.; GALANTE, T. Estruturas de Dados. Porto Alegre: Bookman, 2009. [eBook]

MORIN, P. Open Data Structures (in Java) Creative Commons, 2011. Disponível em <http://opendatastructures.org/ods-java.pdf> [eBook]

PUGA, S.; RISSETTI, G. Estruturas de Dados com aplicações em Java, 2a ed. São Paulo: Pearson, 2008. [eBook]

SHAFFER, C. A.; Data Structures and Algorithm Analysis. Virginia Tech, 2012. Disponível em