Aula 5
Coleções
Sacos e Pilhas

Algoritmos e Estruturas de Dados

Saco (*bag*)

Especificação e Implementação

API do tipo Saco (bag)

- Coleção que permite adicionar vários elementos, percorrê-los, mas não apagar elementos

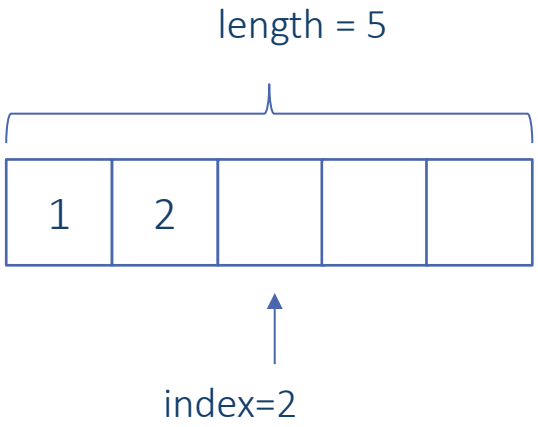
Bag

```
public class Bag<Item> implements Iterable<Item>
```

	`Bag()`	*create an empty bag*
`void`	`add(Item item)`	*add an item*
`boolean`	`isEmpty()`	*is the bag empty?*
`int`	`size()`	*number of items in the bag*

Implementação Tipo Saco

- Implementação como array redimensionável



index – Índice que aponta para onde o próximo elemento irá ser colocado no saco. Também pode ser usado para sabermos o número de elementos no saco.

Implementação Tipo Saco

- Implementação do Saco como um *Array* redimensionável

```
package aed.collections;

import java.util.Iterator;

public class Bag<T> implements Iterable<T> {
    private T[] items;
    private int index;

    private static final int INITIAL_ARRAY_LENGTH = 10;

    @SuppressWarnings("unchecked")
    public Bag()
    {
        items = (T[]) new Object[INITIAL_ARRAY_LENGTH];
        index = 0;
    }
    …
```

nome da package e subpackage

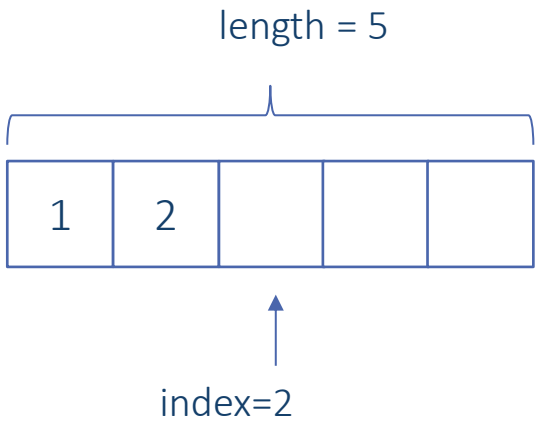
Constante

Necessário fazer por causa do cast. Caso contrário obtemos um warning

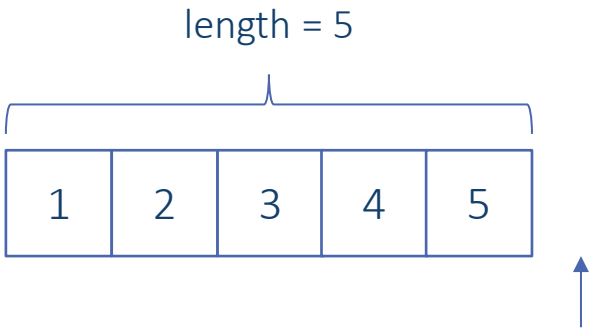
Por razões técnicas (covariância) não é possível criar um array genérico
Temos de criar um array de Objectos e fazer cast

Implementação Tipo Saco

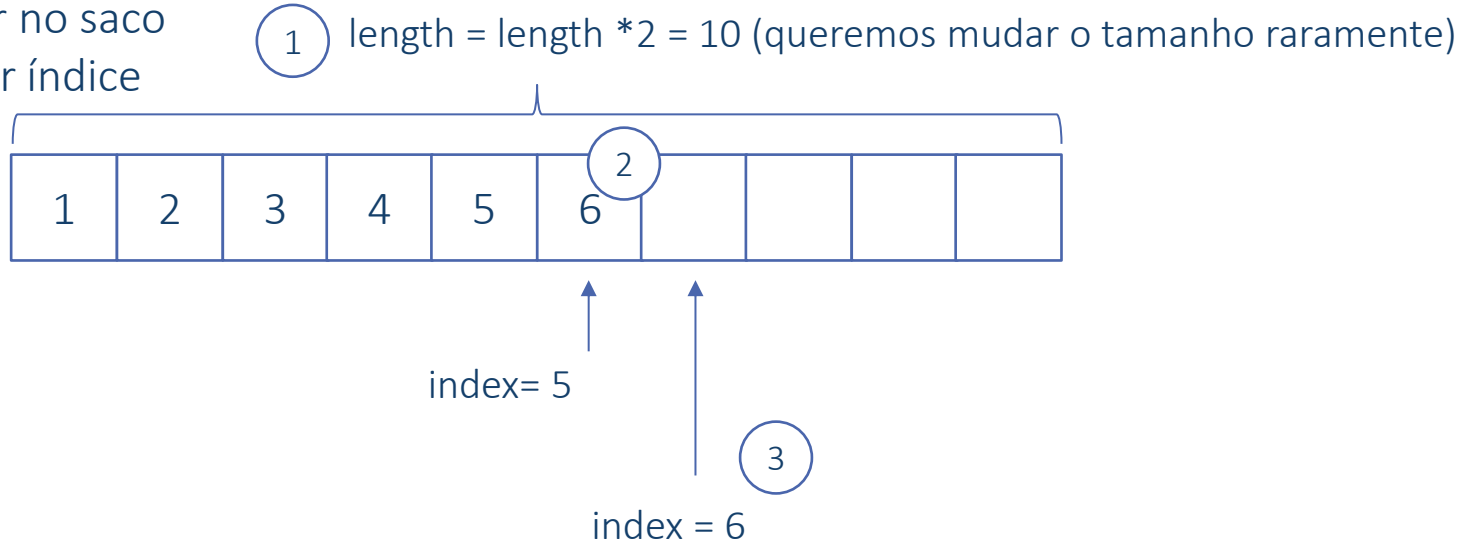
```
public boolean isEmpty()
{
    return index == 0;
}

public int size()
{
    return index;
}


@Override
public Iterator<T> iterator()
{
    return new BagIterator<T>();
}
```



Método *add* c/ *resize*

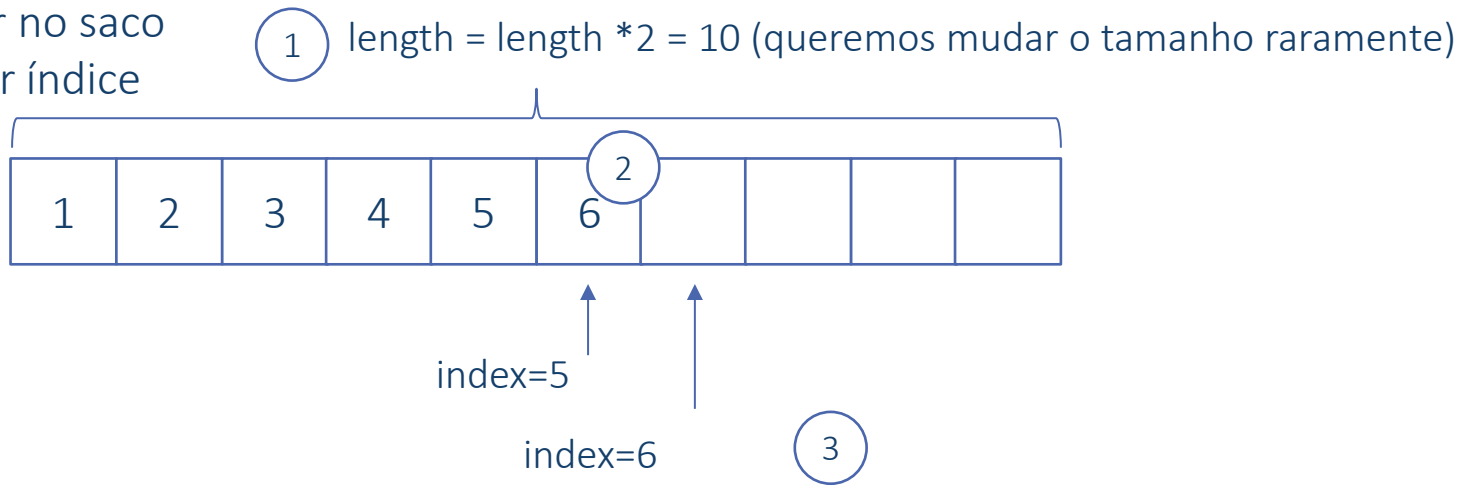


- add(6)
  1. Resize
  2. Colocar no saco
  3. Avançar índice



Método *add* c/ *resize*

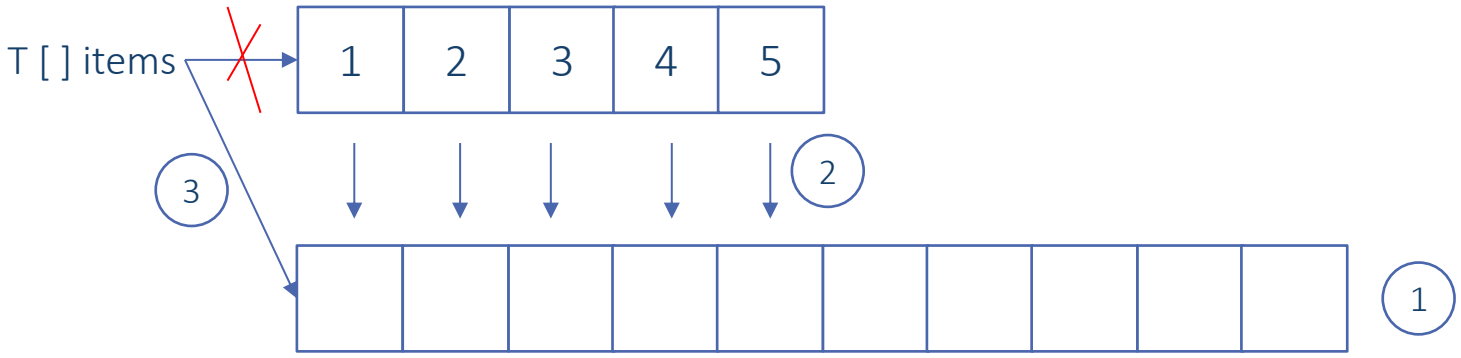
- add(6)
  1. Resize
  2. Colocar no saco
  3. Avançar índice



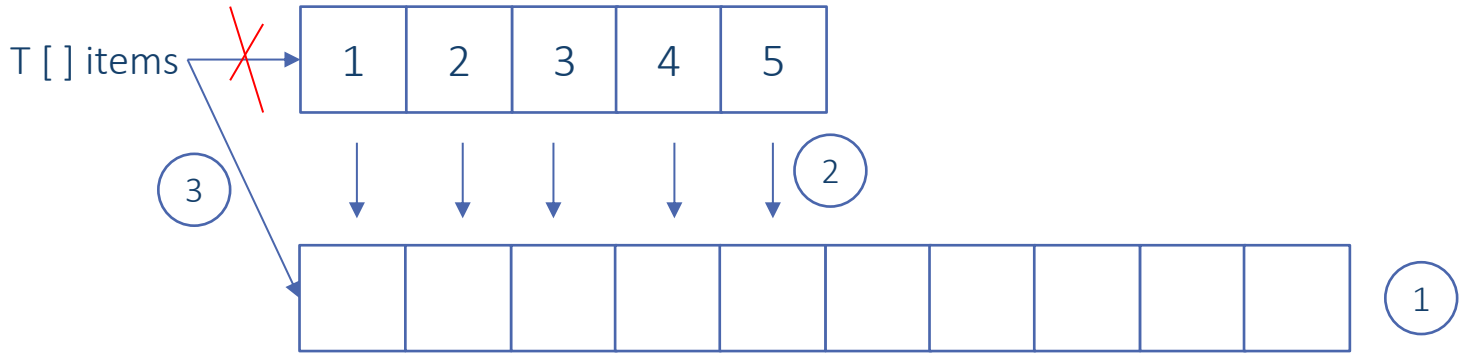

```
public void add(T item)
{
    if(this.index == this.items.length) {
        resize(2*this.items.length);
    }
    this.items[index++] = item;
}
```

Mudar o tamanho do array?

- Na realidade não é possível directamente
  1. Cria-se um novo array (com o tamanho maior)
  2. Copiam-se os elementos do actual para lá
  3. Muda-se a referência para o novo array



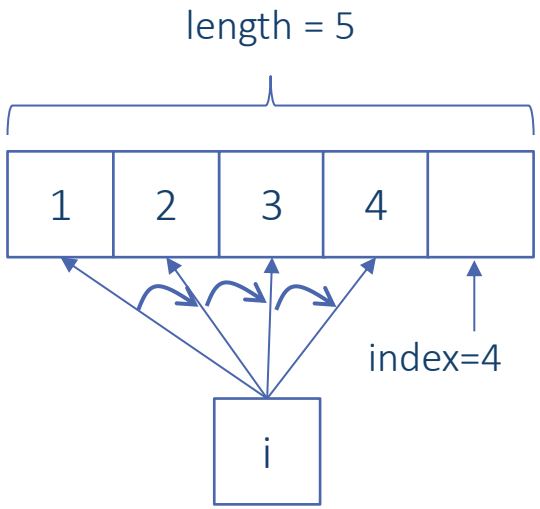
Mudar o tamanho do array?




```
@SuppressWarnings("unchecked")
private void resize(int newArrayLength)
{
    T[] resizedArray = (T[]) new Object[newArrayLength];
    for(int i = 0; i < this.index; i++)
    {
        resizedArray[i] = this.items[i];
    }
    this.items = resizedArray;
}
```

BagIterator

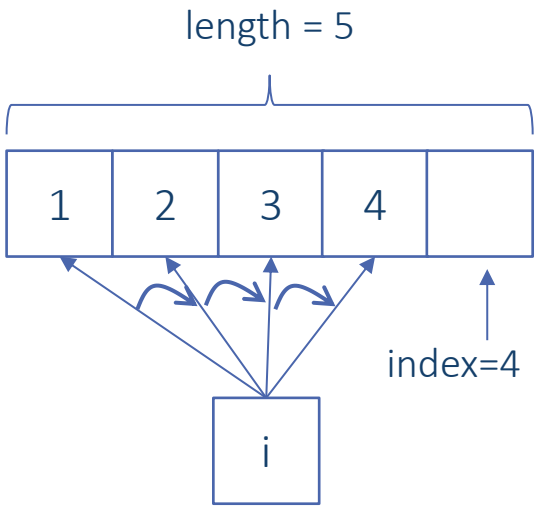
- Cria-se um índice independente com estado
  - Com acesso ao array do saco



Implementação Tipo Saco

```
private class BagIterator implements Iterator<T>
{
    private int i = 0;
    //we don't need the Bag because a innerclass has access to
    // the fields of the outerclass
    public BagIterator()
    {
        this.i = 0;
    }
    @Override
    public boolean hasNext() {
        return i < index;
    }
    @Override
    public T next() {
        return items[i++];
    }
    @Override
    public void remove() {
        throw new UnsupportedOperationException("Bag doesn't support el. removal");
    }
}
```

classe BagIterator definida dentro da classe Bag como innerclass



Exemplo de utilização de Sacos

```
public static void main(String[] args) {
    Bag<String> saco = new Bag<String>();

    saco.add("ola");
    saco.add("isto");
    saco.add("e");
    saco.add("um");
    saco.add("teste");
    for(String s : saco)
    {
        System.out.println(s);
    }
}
```

```
"C:\Program Files\Java\jdk-14.0.2\bin\java.exe"
ola
isto
e
um
teste

Process finished with exit code 0
```

Pilha (*Stack*)

Especificação e Implementação (como array)

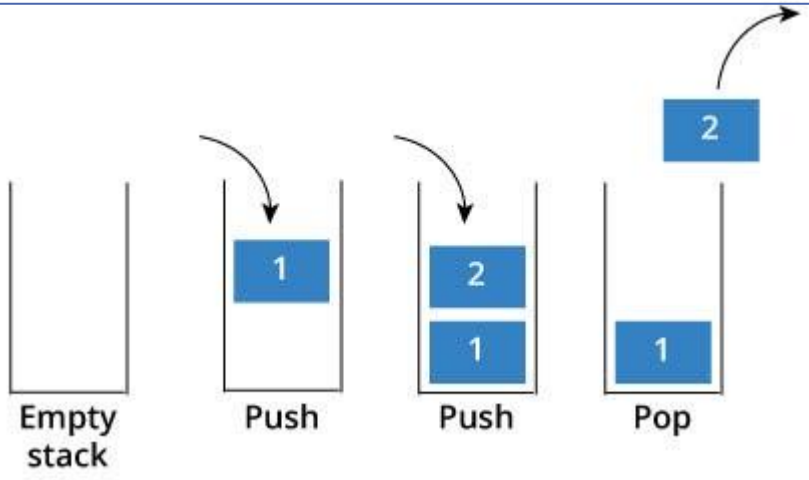
Pilha (*Stack*)

- Coleção em que se só se consegue aceder ao elemento no início (topo) da mesma

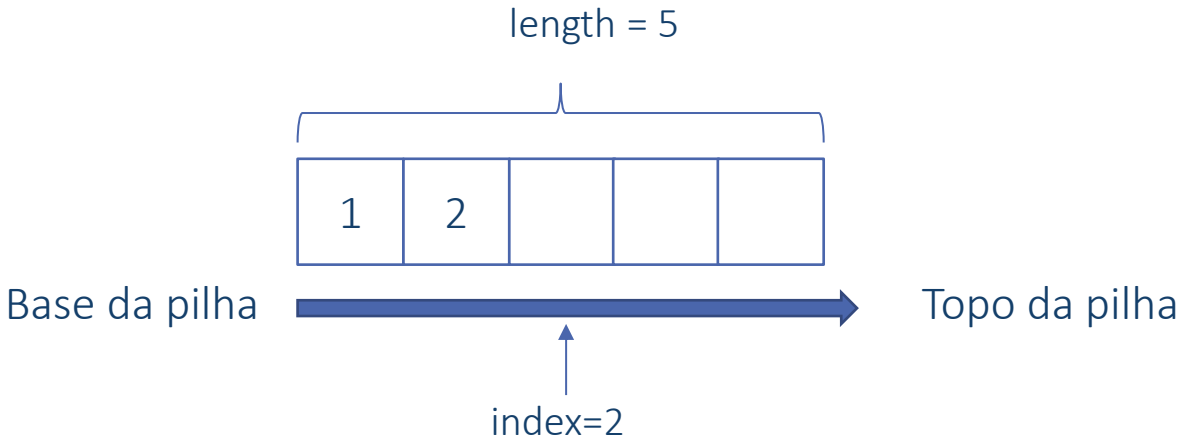
- Diz-se uma coleção *LIFO* (*Last In First Out*)

Especificação Interface Pilha

public interface IStack<Item> extends Iterable<Item>		
`void`	`push(Item item)`	coloca um item no topo da pilha
`Item`	`pop()`	remove e retorna o item no topo da pilha. Caso a pilha esteja vazia deverá retornar *null*
`Item`	`peek()`	retorna o item no topo da pilha, mas não o remove. Retorna null caso a pilha esteja vazia.
`boolean`	`isEmpty()`	retorna *true* se a pilha estiver vazia e *false* caso contrário
`Int`	`size()`	devolve o tamanho (número de elementos) da pilha
`Istack<Item>`	`shallowCopy()`	retorna uma cópia superficial da pilha. Uma cópia superficial copia a estrutura da pilha sem copiar cada item individualmente.



Pilha como *vetor* redimensionável




```
package aed.collections;
import java.util.Iterator;
public class StackArray<T> implements IStack<T>{
    private int index;
    private T[] items;
    private static final int INITIAL_ARRAY_LENGTH = 5;

    @SuppressWarnings("unchecked")
    public StackArray()
    {
        this.index = 0;
        this.items = (T[]) new Object[INITIAL_ARRAY_LENGTH];
    }
```

Métodos *empty, size, peek*

```
public int size(){
    return this.index;
}

public boolean isEmpty()
{
    return this.index == 0;
}

public T peek()
{
    if(this.index <= 0) return null;
    return this.items[this.index-1];
}
```

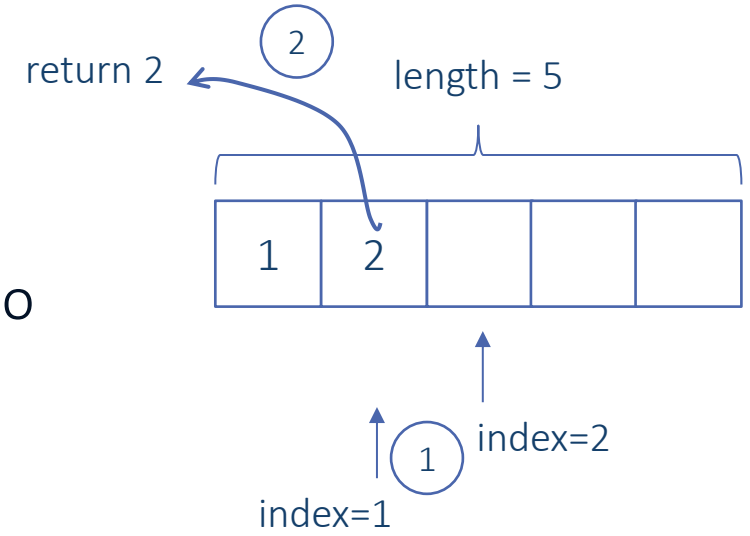
Método *push*

- Implementação igual ao *add* dos sacos

```
public void push(T item)
{
    if(this.index == this.items.length) {
        resize(2*this.items.length);
    }
    this.items[index++] = item;
}

@SuppressWarnings("unchecked")
private void resize(int newArrayLength)
{
    T[] resizedArray = (T[]) new Object[newArrayLength];
    for(int i = 0; i < this.index; i++)
    {
        resizedArray[i] = this.items[i];
    }
    this.items = resizedArray;
}
```

Método *pop*

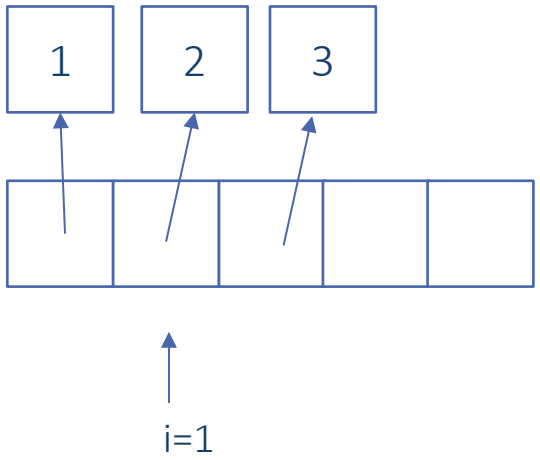
- Abordagem simplista
  1. diminuir índice
  2. retornar elemento do topo
  3. profit




```
public T pop()
{
    if(this.index == 0) return null;
    this.index--;
    T result = this.items[this.index];
    return result;
}
```

Problema: vadiagem

- Vadiagem (*Loitering*)
  - Uma espécie de fuga de memória



- *Ex:*
  push(1),push(2),push(3),pop(),pop()

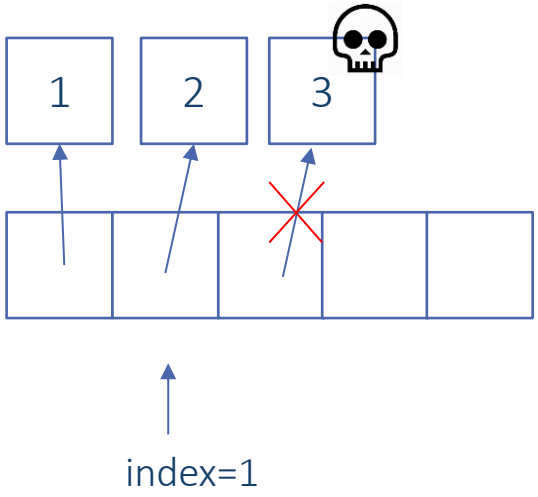
A posição do índice i=2 continua a referenciar o objecto 3

Este nunca será marcado para remoção pelo GC

A não ser que a pilha volte a crescer

Solução problema vadiagem

```
public T pop()
{
    if(this.index == 0) return null;
    this.index--;
    T result = this.items[this.index];
    this.items[this.index] = null;
    return result;
```



Problema 2: Desperdicio de memória

- Memória usada não se ajusta dinâmicamente
  - pior caso: 2*$index_{max}$
  - deviamos libertar memória quando já não precisamos dela

Solução 1

- Reduzir memória para metade quando $size < \frac{length}{2}$

- Problema
  - Imaginemos size = length

push → resize >

pop → resize <

push → resize >

push → resize <

pop → resize >

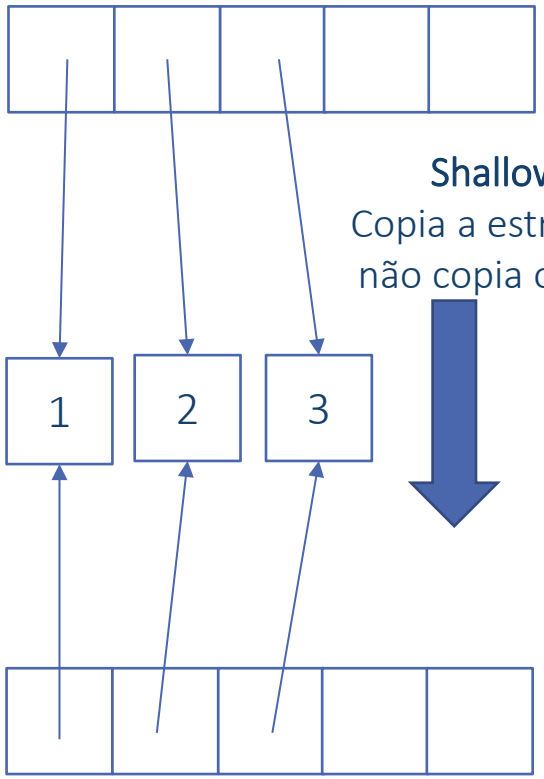
…

Método *pop* final

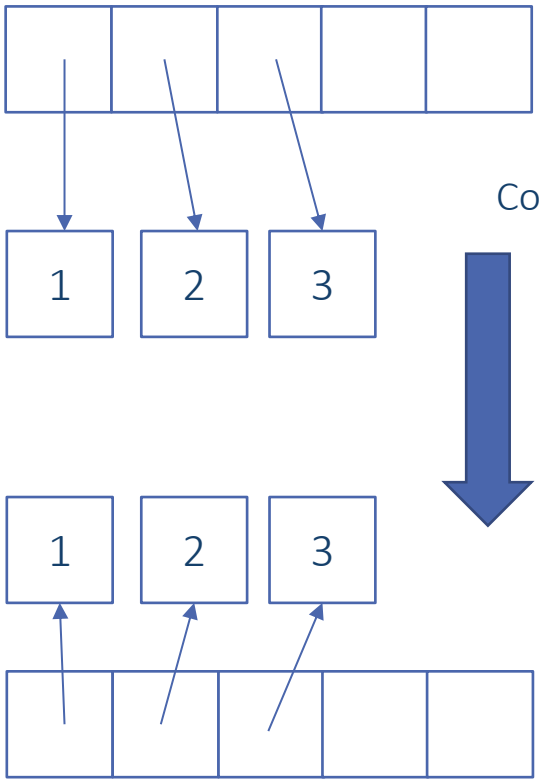
- Reduzir memória para metade quando $size < \frac{length}{4}$

```
public T pop()
{
    if(this.index == 0) return null;
    this.index--;
    T result = this.items[this.index];
    this.items[this.index] = null;
    if(this.index <= this.items.length/4)
    {
        resize(this.items.length/2);
    }
    return result;
}
```

ShallowCopy vs *DeepCopy*



Shallow Copy
Copia a estrutura, mas não copia os objectos



Deep Copy
Copia a estrutura, e os objectos

Método *shallowCopy*

- semelhante ao método resize, mas sem mudar de tamanho

```
@SuppressWarnings("unchecked")
public StackArray(int size)
{
    this.index = 0;
    this.items = (T[]) new Object[size];
}
public IStack<T> shallowCopy()
{
    StackArray<T> newStack = new StackArray<>(this.index);
    for(int i = 0; i < this.index;i++)
    {
        //shallowcopy - only copy the reference
        newStack.items[i] = this.items[i];
    }
    newStack.index = this.index;
    return newStack;
}
```

Métodos Iterador

- Semelhante ao iterador dos sacos, mas começa da direita para a esquerda
  - Queremos percorrer pela ordem natural da pilha (do topo para a base)

```
@Override
public Iterator<T> iterator() {return new StackArrayIterator();}

private class StackArrayIterator implements Iterator<T>
{
    private int it;

    StackArrayIterator(){this.it = index-1;}

    @Override
    public boolean hasNext() {return this.it >= 0;}

    @Override
    public T next() {return items[it--];}

    @Override
    public void remove() {
        throw new UnsupportedOperationException("Iterator doesn't support removal");
```

Eficiência pilha como vetor

- Complexidade temporal
  - Análise amortizada

    Def: *Começando de uma estrutura vazia, é a média do tempo de execução para o pior caso de uma sequência de operações*

	best	worst	amortized
construct	1	1	1
push	1	N	1
pop	1	N	1
size	1	1	1

doubling and halving operations

Eficiência pilha como vetor

- Complexidade espacial
  - Memória array

    = 16 bytes (header+tam) + 8 bytes x tamanho array

  - Sendo ***N*** o número de elementos na stack

    Casos extremos

    Exactamente antes de aumentar o tamanho da stack

    Taxa de ocupação 100% - 8**N*

    Exactamente antes de diminuir o tamanho da stack

    Taxa de ocupação 25% - 4*8**N*= 32**N*

  - Stack como vetor ocupa entre 8**N* e 32**N* bytes