

# ALGORITMOS E ESTRUTURAS DE DADOS 2023/2024

## TIPOS ABSTRATOS DE DADOS

---

Armanda Rodrigues

13 de setembro 2023

# Tipos Abstratos de Dados (TAD)

TIPOS DE DADOS

ABSTRAÇÃO

# Tipo de Dados

- Conceito aplicado originalmente aos tipos de dados primitivos disponíveis numa linguagem de programação
  - E.g. Em Java, `int` e `double`;
- Quando se fala num tipo primitivo referimo-nos a:
  - Um conjunto de Itens de dados com determinadas características (Domínio);
  - Um conjunto de operações que podem ser efetuadas sobre esses itens.
- As operações permitidas num tipo de dados são inseparáveis da sua identidade
- Para compreendermos o tipo precisamos de compreender que operações podem ser executadas sobre o mesmo

## Exemplo:

O tipo `int` em Java está associado aos números inteiros entre  $-2,147,483,648$  and  $+2,147,483,647$  e aos operadores `+`, `-`, `*`, `/`,...

# Tipos de Dados

- Quando usamos Programação Orientada a Objetos, os tipos de dados que pretendemos adicionar podem ser criados através de Classes.
- Os tipos de dados podem representar quantidades numéricas que são usadas de forma muito similar aos tipos primitivos
  - E.g. Uma classe representativa de fracções (com campos de numerador e denominador)
- Estas classes envolvem operações parecidas com as dos tipos primitivos mas que têm de ser aplicadas utilizando uma notação funcional
  - Por exemplo `add()` e `sub()` em vez de `+` e `-`
- O termo TIPO de DADOS aplica-se de forma natural a estas classes

# Tipos de Dados

- Existem classes que não incluem este aspeto quantitativo
- Qualquer classe representa uma implementação de um tipo de dados, tendo:
  - uma componente de dados (que pode ser realizada com um conjunto de atributos);
  - Um conjunto de operações permitidas sobre os dados (métodos)
- Assim, quando pretendemos representar um conceito do dia a dia, como uma Conta Bancária, ou uma Pilha de Livros, estes conceitos também podem ser tratados como Tipos de Dados



# Abstração

- “*Separação mental de um ou mais elementos concretos de uma entidade complexa desprezando outros que lhe são inerentes*” - Dicionário Porto Editora da Língua Portuguesa
- Uma abstração é a essência, as características importantes de algo
- Por exemplo, o Presidente da República Portuguesa é uma abstração, considerada à parte do indivíduo que ocupa o lugar num determinado momento.
  - Os poderes e as responsabilidades do cargo mantêm-se enquanto que os indivíduos vão e vêm



# Tipo Abstrato de Dados em OO

- Em Programação OO um TAD é composto de:
  - Descrição dos dados (atributos)
  - Lista de operações (métodos) que podem ser aplicadas aos atributos
  - Instruções sobre como utilizar as mesmas operações
- São propositadamente excluídos os detalhes relativos à maneira como os métodos executam as suas tarefas
- À especificação de um TAD chamamos um *interface*.

# Tipo Abstrato de Dados em OO

- Um Tipo Abstrato de Dados pode ser associado a várias implementações, instanciadas em classes
- Cada uma destas classes poderá encapsular a utilização de uma determinada estrutura de dados, para implementar o TAD, com diferentes performances:
  - Em termos do tempo necessário para executar as operações sobre a Estrutura de Dados
  - Em termos do espaço (em memória) necessário para guardar a informação associada ao Tipo de Dados



# TADs em AED

- Os exemplos e exercícios que vamos resolver em AED vão implicar a especificação de dois tipos de TADs
  - TADs diretamente relacionados com o domínio do problema que pretendemos resolver
    - Por exemplo, Conta Bancária ou Supermercado
  - TADs genéricos usados em Programação para resolver problemas com determinadas características
    - Por exemplo, Pilha, Fila ou Dicionário
- As implementações de TADs genéricos serão auxiliadas pela escolha de estruturas de dados adaptadas às características subjacentes aos tipos
  - E.g. Uma Pilha pode ser implementada em Vetor ou em Lista Ligada
- As classes resultantes de implementações de TADs genéricos podem contribuir para as implementações de TADs associados ao domínio de certos problemas
  - Uma fila de espera num Supermercado pode ser instanciada através de uma implementação do TAD Fila

# Exemplo da Biblioteca

- Vamos pensar quais seriam os TADs necessários para a implementação de um Sistema de uma Biblioteca
- Este sistema contém vários subsistemas
- Iremos debruçar-nos sobre eles de acordo com as oportunidades que nos aparecem relativamente aos TADs a especificar e às estruturas de dados disponíveis



# Biblioteca - TADs do Domínio

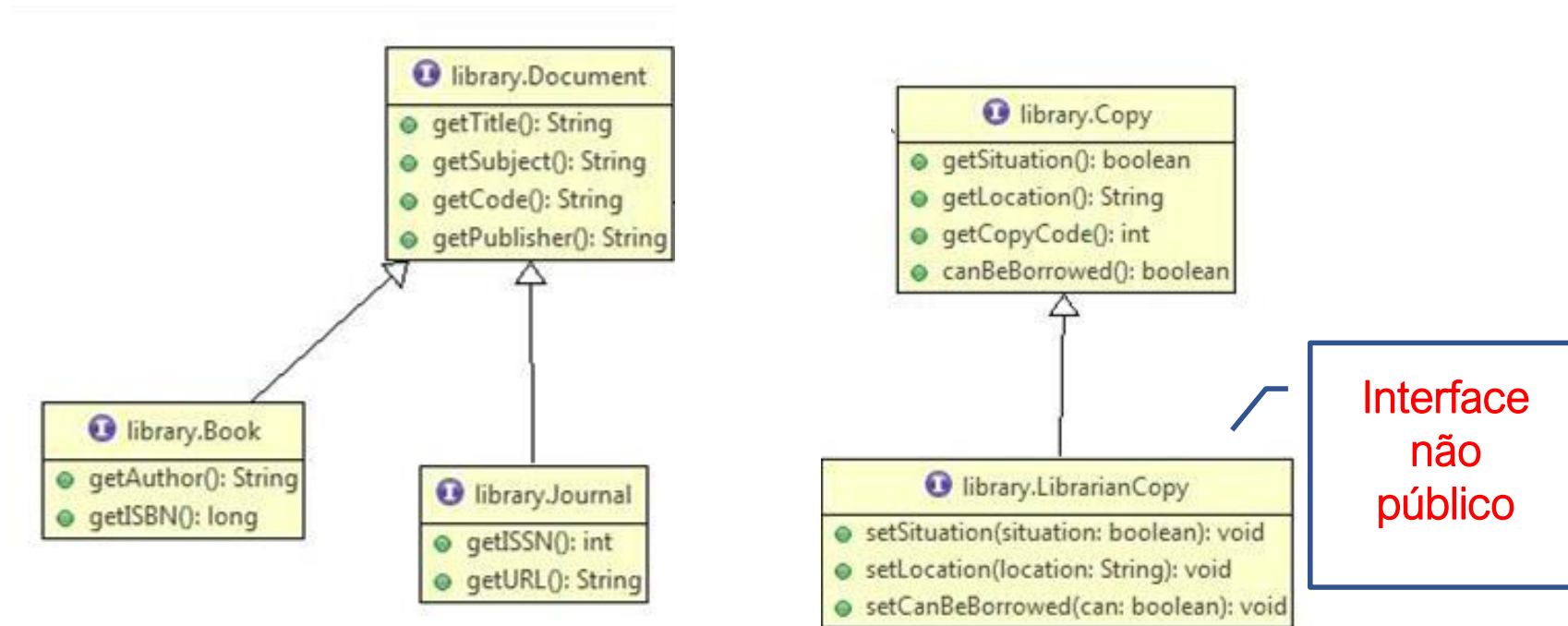
- O Sistema da Biblioteca deverá envolver o armazenamento e tratamento de informação relativa a Documentos existentes na mesma
- Existem vários tipos de **Documentos**: Livros, CDs, DVDs, Revistas, etc...
  - Para simplificar vamos cingir-nos a Livros e a Revistas
- Estes documentos podem existir na biblioteca em várias cópias, ou exemplares
- Como informação de base para um **Documento** existe o Título, o Assunto, a Cota e a Editora
- Um **Livro** terá informação adicional que irá incluir o seu Autor e o ISBN
- Quanto à **Revista**, esta irá incluir ISSN e URL, uma vez que estará disponível online
- Uma Revista está sempre disponível num único exemplar que não pode ser emprestado
- No caso do Livro, podem existir vários exemplares que podem ser emprestados se estiverem disponíveis, com Código, Situação (emprestado ou livre) e Localização na biblioteca

# Biblioteca - TADs do Domínio

- Como organizariam os TADs deste exemplo ?

# Biblioteca - TADs do Domínio

- Como organizariam os TADs deste exemplo ?



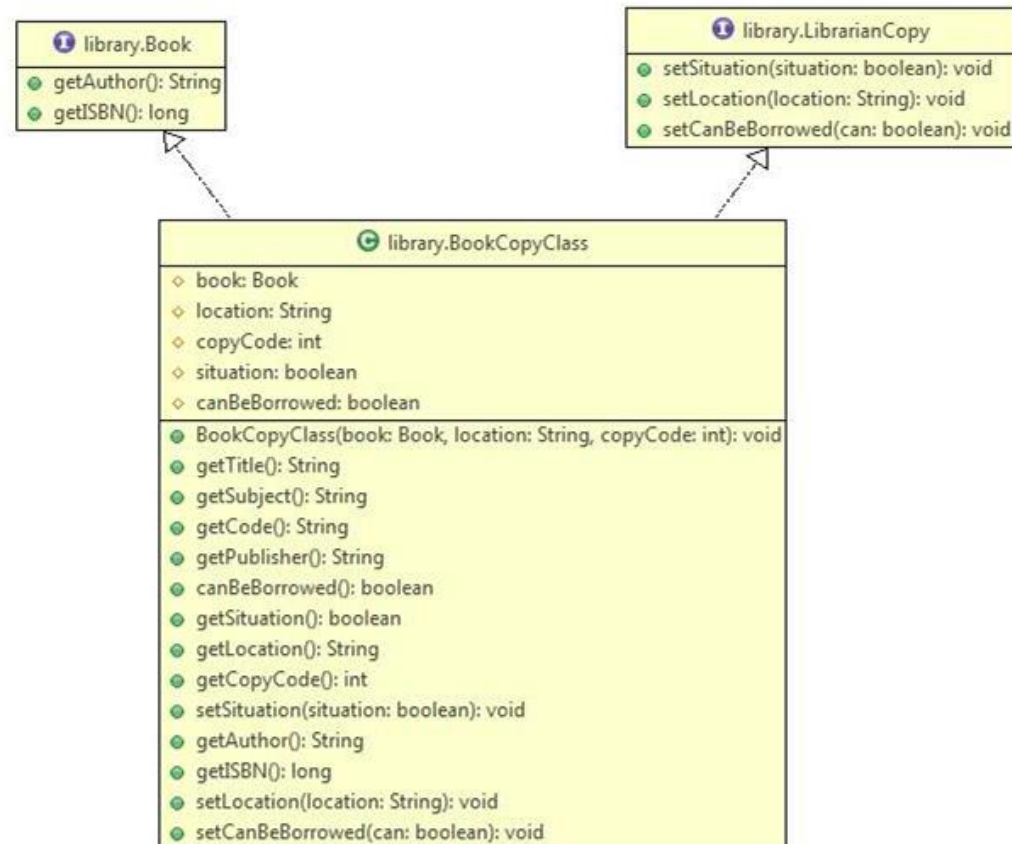
# Biblioteca - TADs do Domínio

- Como seria uma classe Cópia de Livro ?

# Biblioteca - TADs do Domínio

- Como seria uma classe Cópia de Livro ?

Esta classe poderia ser disponibilizada a aplicações que não alterassem a situação da cópia, através dos interfaces Copy ou Book



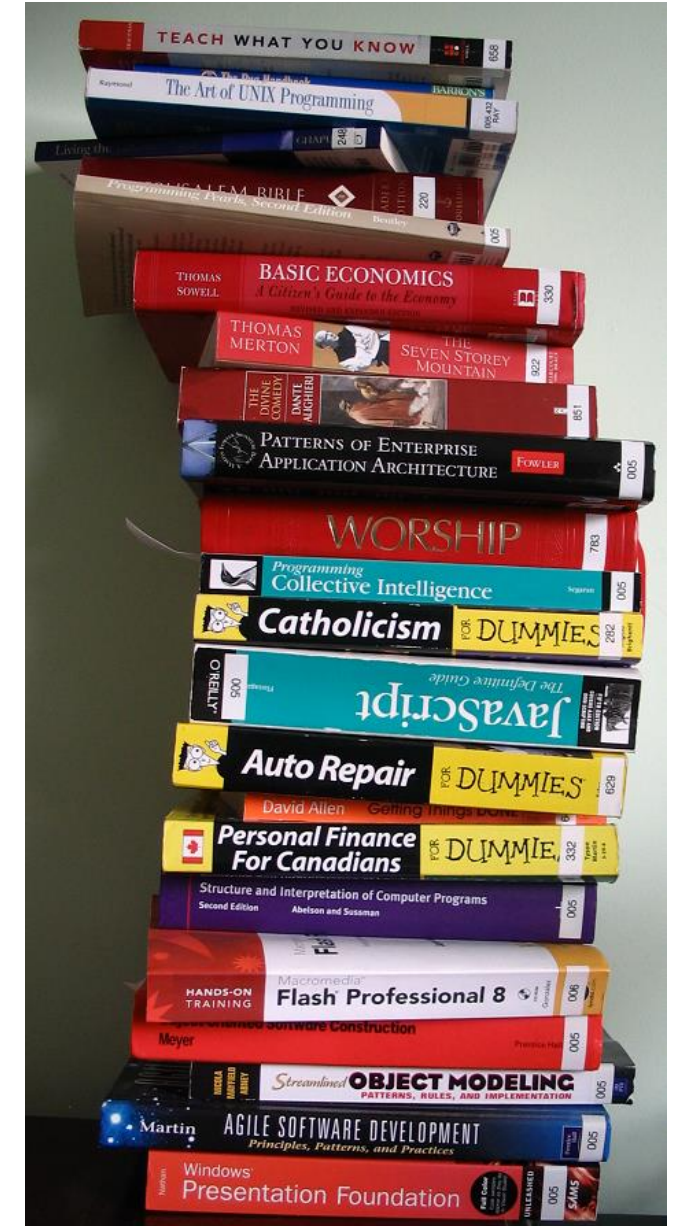
# Devoluções de exemplares de livros

- No momento da devolução de um exemplar na Biblioteca, a funcionária poderá não ter tempo para repor o livro juntamente com os exemplares disponíveis
- O processo normal será que todos os exemplares devolvidos fiquem guardados juntos, empilhados ao lado da receção da biblioteca
- Assim, o processo de devolução tem dois passos:
  - Receber o exemplar, o que implica inserir o exemplar numa pilha de exemplares a repor. A funcionária poderá receber até 10 devoluções antes de fazer, obrigatoriamente as reposições dos exemplares.
  - Mais tarde, com tempo, repor o exemplar no seu lugar. Neste caso, a funcionária irá, naturalmente, retirar o exemplar que está no topo da pilha de reposição, alterar a informação que lhe está associada (tornando-o disponível para empréstimo) e reinseri-lo no conjunto de cópias disponíveis.



# Uma Pilha de Livros

- Para implementar o TAD **DevolucoesExemplar**, vai ser preciso utilizar uma implementação do TAD pilha
- É preciso compreender como o TAD Pilha funciona
- E conhecer pelo menos uma implementação do mesmo TAD



---

## TAD Pilha – Elementos do tipo E

# TAD Pilha de Elementos do Tipo E

```
// Retorna true sse a pilha estiver vazia.  
boolean vazia( );
```

```
// Retorna o elemento do topo da pilha.  
// Pré-condição: a pilha não está vazia.  
E topo( );
```

```
// Coloca o elemento especificado no topo da pilha.  
void empilha( E elemento );
```

```
// Remove e retorna o elemento do topo da pilha.  
// Pré-condição: a pilha não está vazia.  
E desempilha( );
```

# Interface Pilha de Elementos do Tipo E

```
package dataStructures;  
public interface Stack<E>{
```

Vamos usar **Requires** para assinalar pré-condições em métodos

```
    // Returns true iff the stack contains no elements.  
    boolean isEmpty( );  
  
    // Returns the number of elements in the stack.  
    int size( );  
  
    // Returns the element at the top of the stack.  
    // Requires: size() > 0  
    E top( ) throws EmptyStackException;  
  
    // Inserts the specified element onto the top of the stack.  
    void push( E element );  
  
    // Removes and returns the element at the top of the stack.  
    // Requires: size() > 0  
    E pop( ) throws EmptyStackException;  
}
```

# Interface Pilha de Elementos do Tipo E - Javadoc

Todo o código que vos for fornecido será comentado para Javadoc

```
/**
 * Stack Abstract Data Type
 * Includes description of general methods for the
Stack with the LIFO discipline.
 * @author AED Team
 * @version 1.0
 * @param <E> Generic Element
 */
public interface Stack<E>
{

    /**
     * Returns true iff the stack contains no
     * elements.
     * @return true iff the stack contains no
     *         elements, false otherwise
     */
    boolean isEmpty( );

    /**
     * Returns the number of elements in the stack.
     * @return number of elements in the stack
     */
```

Todo o código submetido pelos alunos deve ser comentado para Javadoc

```
/**
 * Returns the element at the top of the stack.
 * Requires
 * @return element at top of stack
 * @throws EmptyStackException when size = 0
 */
E top( ) throws EmptyStackException;

/**
 * Inserts the specified <code>element</code> onto
 * the top of the stack.
 * @param element element to be inserted onto the stack
 */
void push( E element );

/**
 * Removes and returns the element at the top of the
 * stack.
 * @return element removed from top of stack
 * @throws EmptyStackException when size = 0
 */
E pop( ) throws EmptyStackException;
```

# Regras para comentários em AED

- Todo o código que vos for fornecido será comentado para Javadoc
- Devido às necessidades de leitura dos materiais de disciplina, os slides não contêm estes comentários
- **Todo o código submetido pelos alunos deve ser comentado para Javadoc**
  - Comentar os métodos dos TADs (interfaces)
  - Comentar todos os métodos de classes que não façam parte dos TADs
  - Nas classes, só comentar métodos que correspondem ao interface público em casos de implementações específicas que necessitem de esclarecimento (em caso contrário usar a tag `@see` ou `@Override`)

```
/*  
 * @see dataStructures.Stack#top()  
 */  
public E top( ) throws EmptyStackException  
{  
    if ( this.isEmpty() )  
        throw new EmptyStackException("Stack is empty.");  
    return array[top];  
}
```

**Comentar sempre  
que os comentários  
façam falta!!!**

---

## Implementação do Interface Pilha em Vetor

# Pilha em vetor

**capacity** = 4  
**top** = -1





# Pilha em vetor

**capacity** = 4  
**top** = -1

**array**



**push(5);**

# Pilha em vetor

**capacity** = 4  
**top** = -1



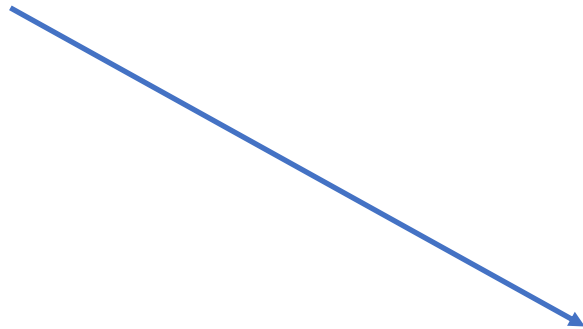
**push(5);**

**top++;**

# Pilha em vetor

**capacity** = 4

**top** = 0



**array**



**push(5);**

**top++;**

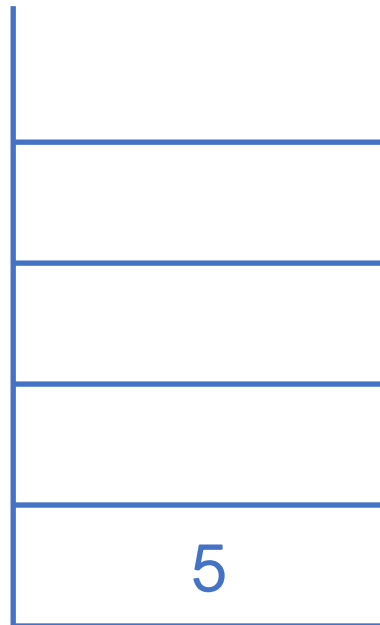
# Pilha em vetor

capacity = 4

top = 0



array



push(5);

array[0] = 5;

# Pilha em vetor

capacity = 4  
top = 0



push(5);  
push(2);

# Pilha em vetor

capacity = 4  
top = 1



push(5);  
push(2);

top++;

# Pilha em vetor

**capacity** = 4  
**top** = 1



**push(5);**  
**push(2);**

**array[top] = 2;**

# Pilha em vetor

capacity = 4  
top = 1



```
push(5);  
push(2);  
top();
```



# Pilha em vetor

capacity = 4  
top = 1



```
push(5);  
push(2);  
top();
```

```
return array[top];
```

# Pilha em vetor

capacity = 4  
top = 1



```
push(5);  
push(2);  
top(); → 2
```

```
return array[top];
```

# Pilha em vetor

capacity = 4  
top = 1

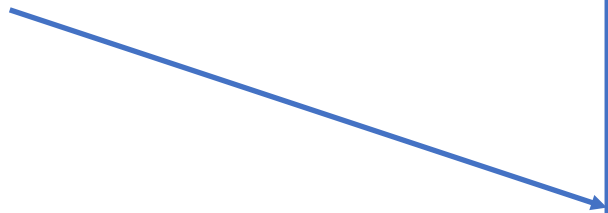


```
push(5);  
push(2);  
top(); → 2  
pop();
```

# Pilha em vetor

capacity = 4

top = 1



array

push(5);

push(2);

top(); → 2

pop();

element = array[top];

# Pilha em vetor

**capacity** = 4

**top** = 1

**element** = 2



**push(5);**

**push(2);**

**top();** → 2

**pop();**

**element = array[top];**

# Pilha em vetor

capacity = 4  
top = 0

element = 2



top--;

```
push(5);  
push(2);  
top(); → 2  
pop();
```

# Pilha em vetor

capacity = 4  
top = 0

element = 2



```
push(5);  
push(2);  
top();  —————> 2  
pop();  —————> 2
```

return element;

# Pilha em vetor

capacity = 4  
top = 0

element = 2



```
push(5);  
push(2);  
top();  —————> 2  
pop();  —————> 2
```

O que falta neste processo ?

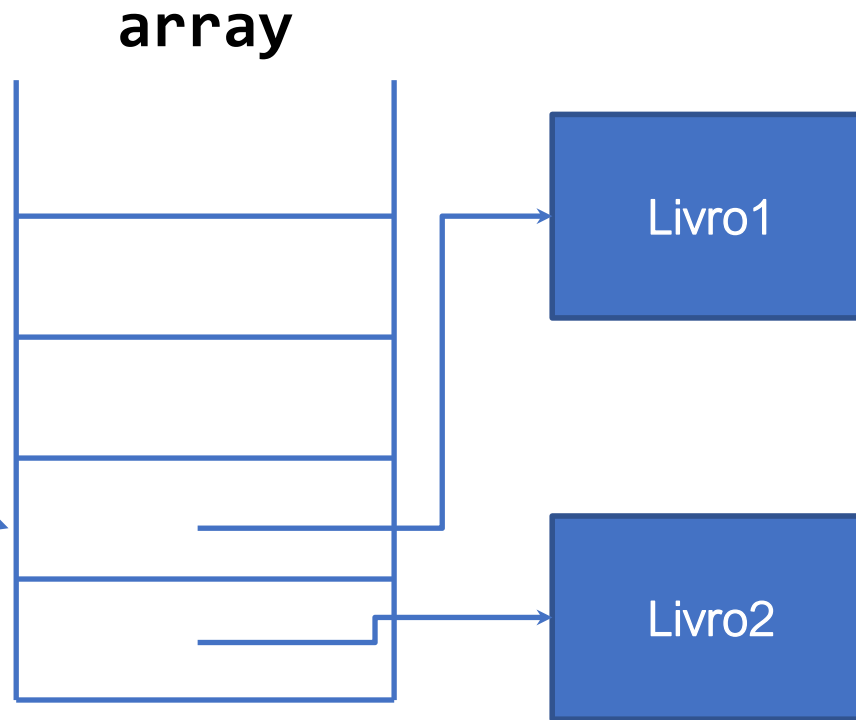
return element;



# Pilha em vetor (de livros)

**capacity** = 4

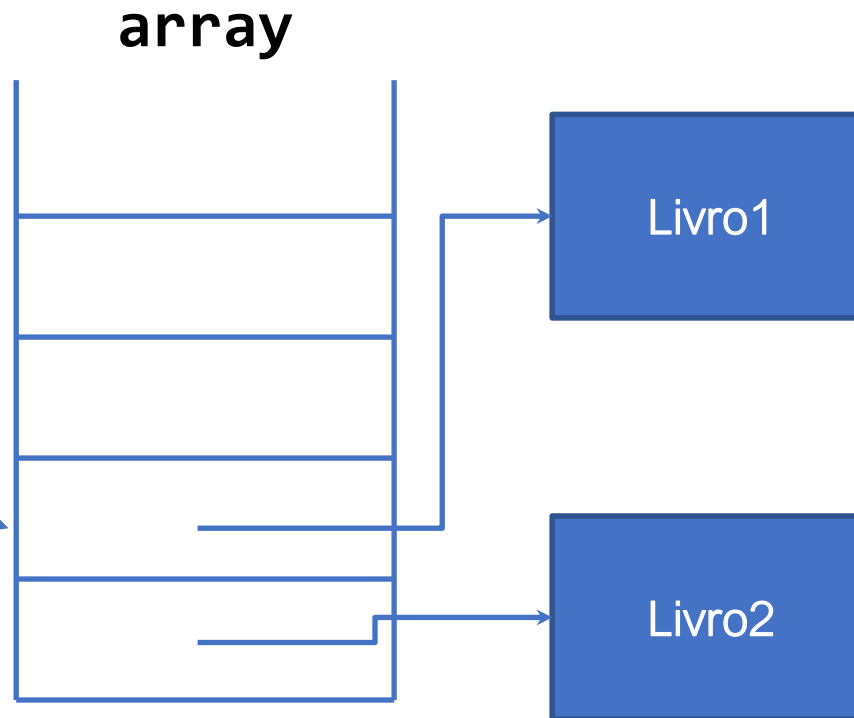
**top** = 1



```
push(Livro2);  
push(Livro1);
```

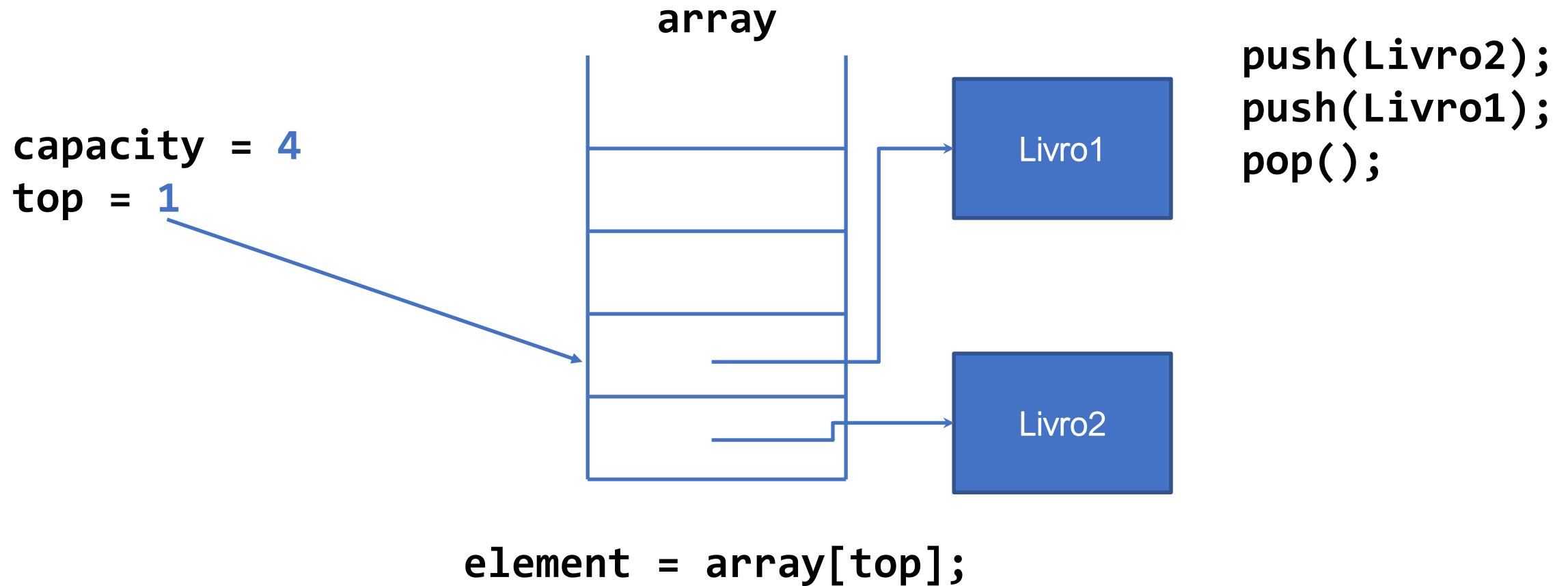
# Pilha em vetor (de livros)

**capacity** = 4  
**top** = 1

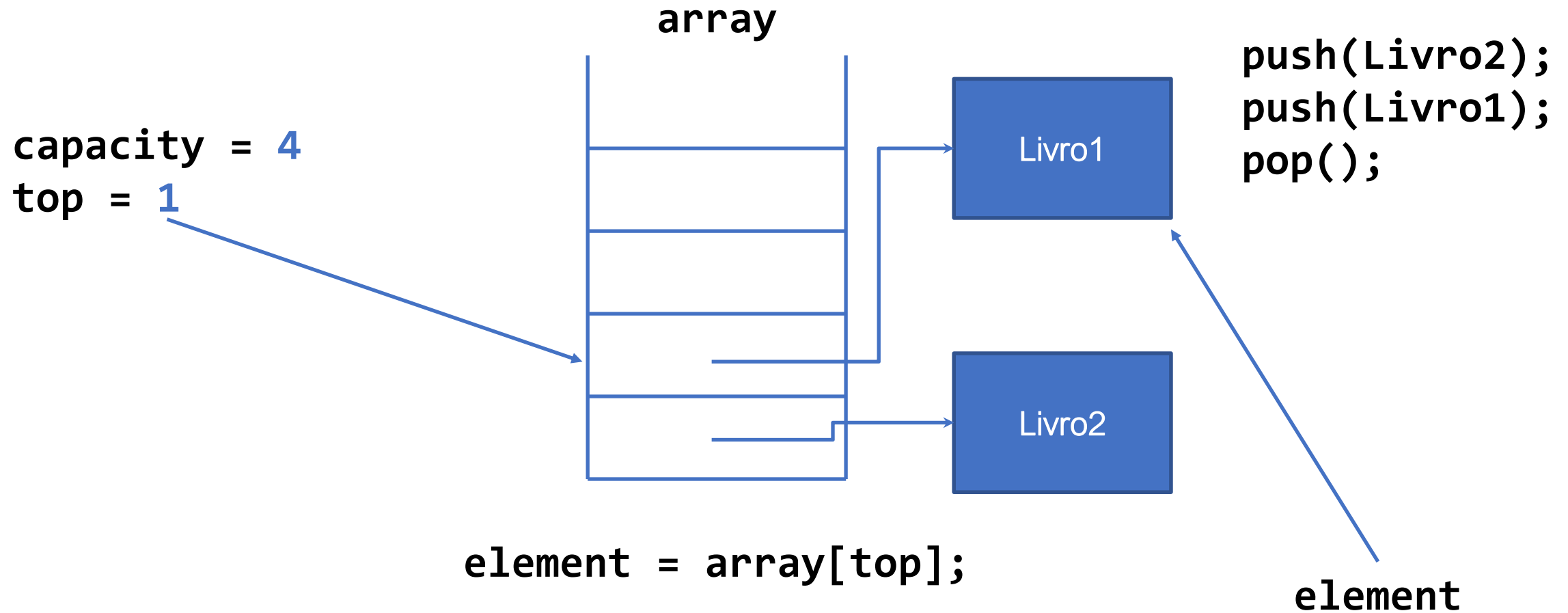


```
push(Livro2);  
push(Livro1);  
pop();
```

# Pilha em vetor (de livros)



# Pilha em vetor (de livros)

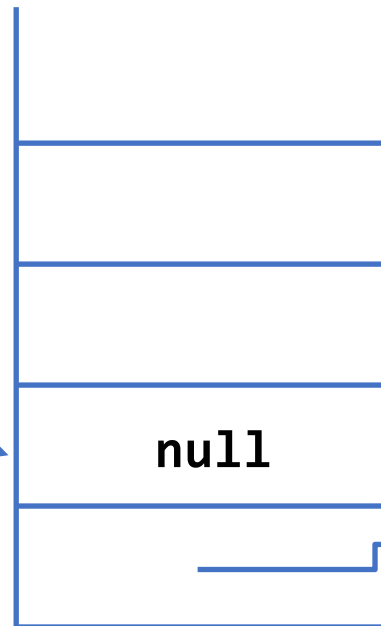


# Pilha em vetor (de livros)

**capacity** = 4  
**top** = 1

É preciso assegurar  
que objeto removido é  
libertado

**array**



**array[top] = null;**

Livro1

Livro2

**push(Livro2);**  
**push(Livro1);**  
**pop();**

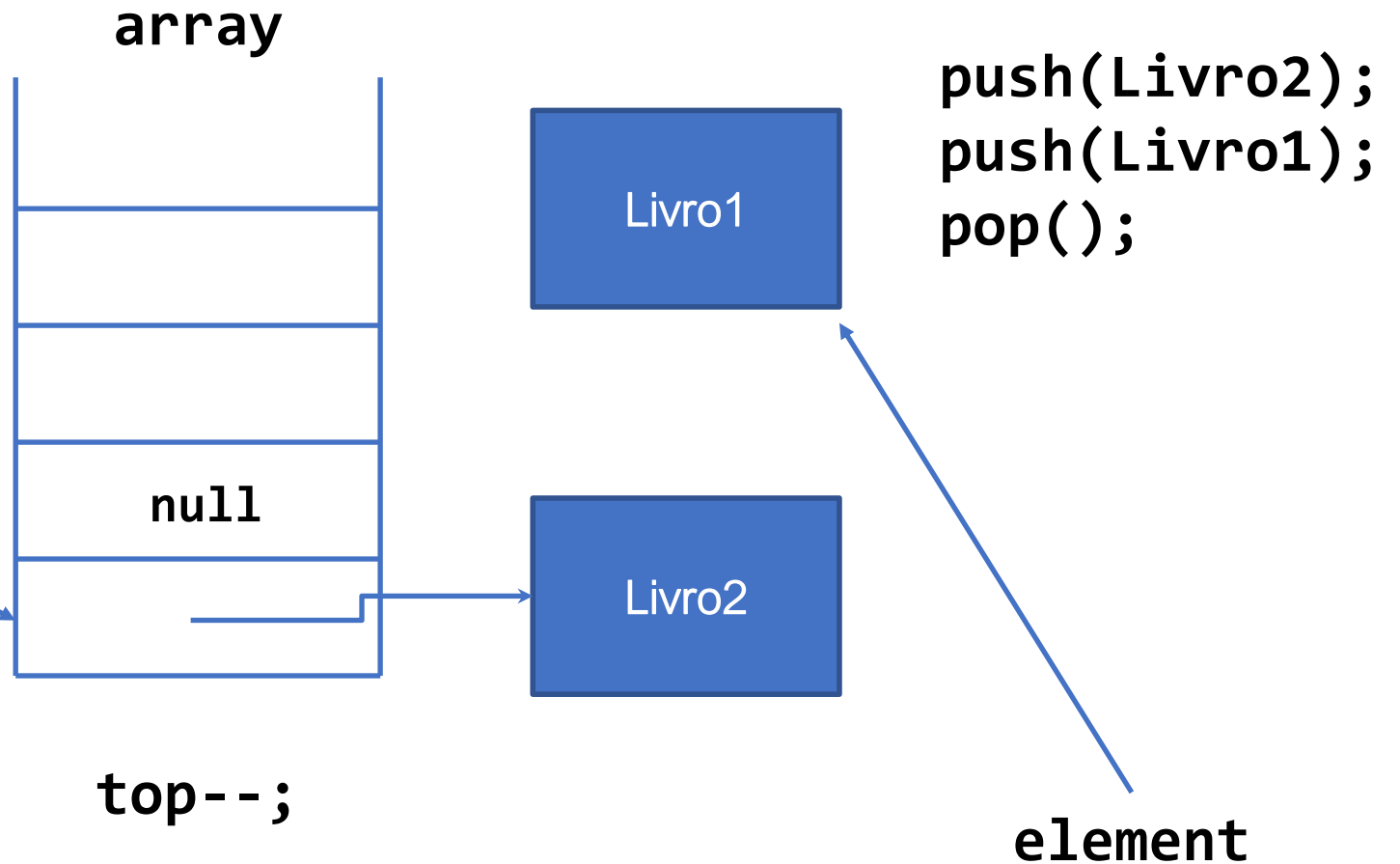
**element**

# Pilha em vetor (de livros)

**capacity** = 4  
**top** = 0

É preciso assegurar que objeto removido é libertado.

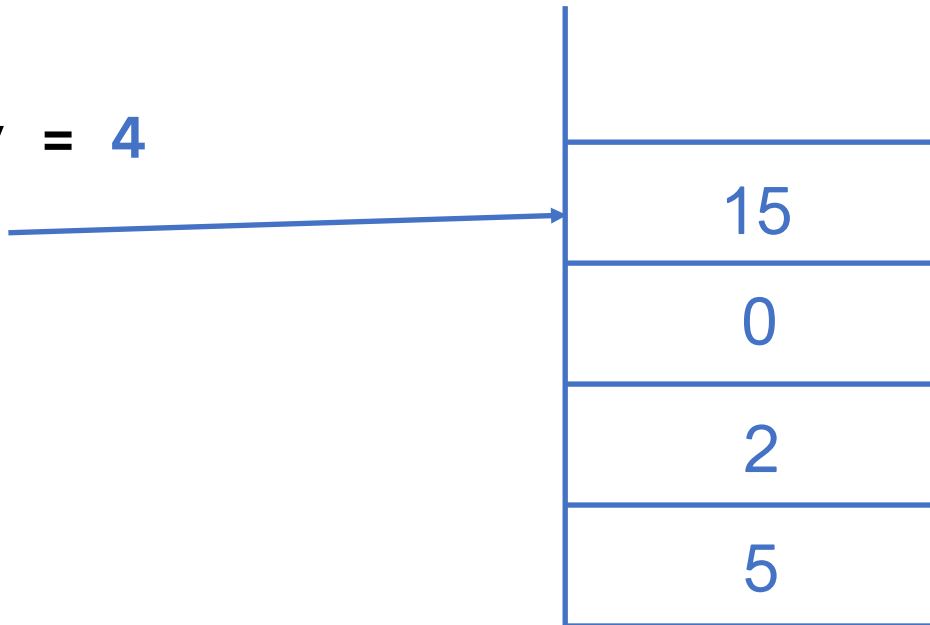
Se não existirem mais referências para Livro1, este será libertado depois de terminada a execução do método



# Pilha em vetor – vetor cheio

**capacity** = 4

**top** = 3

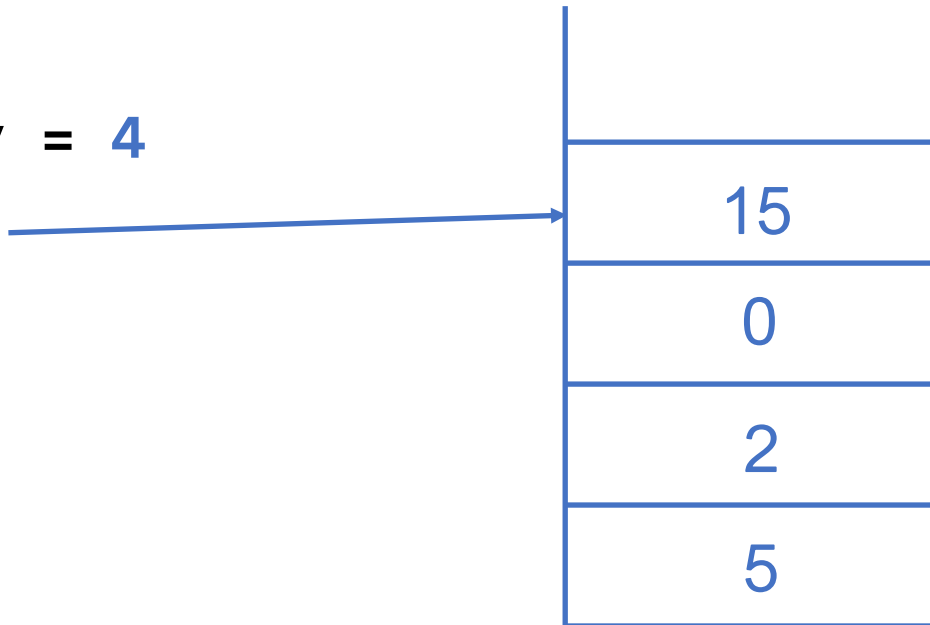


**push(8);**

# Pilha em vetor – vetor cheio

**capacity** = 4

**top** = 3



**push(8);**

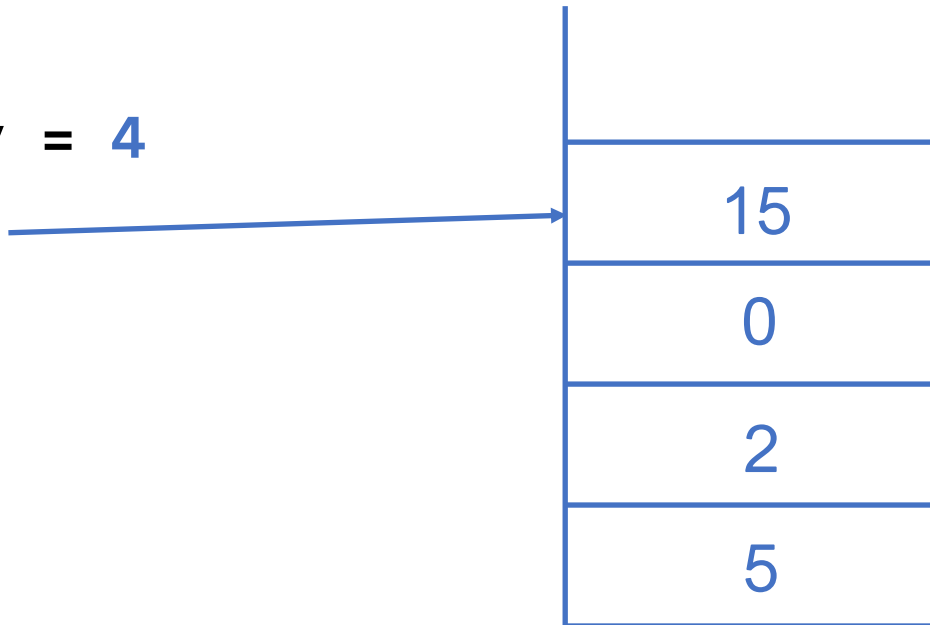
O que acontece  
neste caso ?



# Pilha em vetor – vetor cheio

capacity = 4

top = 3

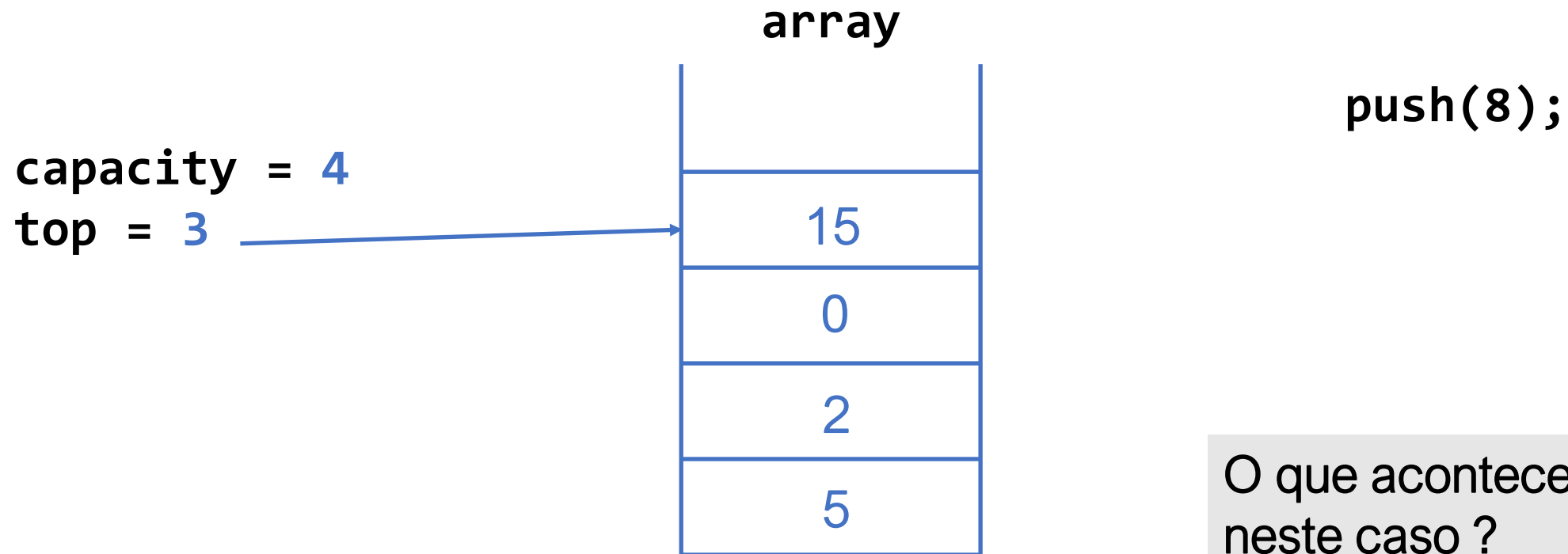


push(8);

O que acontece neste caso ?

**Lançamento de exceção!!**

# Pilha em vetor – vetor cheio



**Lançamento de exceção!!**

Esta exceção não faz parte do TAD Pilha, pois não está associada a uma pré-condição da operação do TAD, mas sim à implementação em vetor.

# Pilha em vetor – vetor vazio

**capacity** = 4  
**top** = -1



# Pilha em vetor – vetor vazio

**capacity** = 4  
**top** = -1



**pop();**

O que acontece  
neste caso ?

# Pilha em vetor – vetor vazio

capacity = 4  
top = -1



pop();

O que acontece  
neste caso ?

**Lançamento de exceção!!**

Esta exceção faz parte do TAD Pilha, e está associada a uma pré-condição da operação do TAD, pelo que deve ser considerada em qualquer implementação do TAD

# Classe Pilha em Vetor (1)

```
package dataStructures;

public class StackInArray<E> implements Stack<E> {

    // Default capacity of the stack.
    public static final int DEFAULT_CAPACITY = 1000;

    // Memory of the stack: an array.
    protected E[] array;

    // Index of the element at the top of the stack.
    protected int top;
```

# Classe Pilha em Vetor (2)

```
@SuppressWarnings("unchecked")  
public StackInArray( int capacity ) {  
    // Compiler gives a warning.  
    array = (E[]) new Object[capacity];  
    top = -1;  
}  
  
public StackInArray( ) {  
    this(DEFAULT_CAPACITY);  
}
```

O aviso vem do facto de não ser possível verificar, em tempo de compilação, os tipos dos dados que vão ser inseridos no vetor.

Para submissão no Mooshak, utilizar `@SuppressWarnings`

# Classe Pilha em Vetor (3)

```
// Returns true iff the stack contains no elements.  
public boolean isEmpty( ){  
    return top == -1;  
}  
  
// Returns true iff the stack cannot contain more elements.  
public boolean isFull( ){  
    return this.size() == array.length;  
}  
  
// Returns the number of elements in the stack.  
public int size( ){  
    return top + 1;  
}
```



# Classe Pilha em Vetor (4)

```
// Returns the element at the top of the stack.  
// Requires: size() > 0  
public E top( ) throws EmptyStackException {  
    if ( this.isEmpty() )  
        throw new EmptyStackException();  
    return array[top];  
}
```

**Novo:** Porque é que testamos as situações de exceção ?

```
// Inserts the specified element onto the top of the stack.  
// Requires: size() < array.length  
public void push( E element ) throws FullStackException {  
    if ( this.isFull() )  
        throw new FullStackException();  
    top++;  
    array[top] = element;  
}
```

# Classe Pilha em Vetor (5)

```
// Removes and returns the element at the top of the stack.  
// Requires: size() > 0  
public E pop( ) throws EmptyStackException {  
    if ( this.isEmpty() )  
        throw new EmptyStackException();  
    E element = array[top];  
    array[top] = null; // For garbage collection.  
    top--;  
    return element;  
}  
} // End of StackInArray.
```

# Classes de Exceções da Pilha

```
package dataStructures;
```

```
public class EmptyStackException extends RuntimeException{  
}
```

```
public class FullStackException extends RuntimeException{  
}
```

**RuntimeException**: superclasse das exceções que podem acontecer durante a operação normal da máquina virtual Java. São exceções de aplicação e não são graves.

Estas exceções são ***unchecked***. Não é necessário declará-las na cláusula **throws** dos métodos, mesmo que elas possam acontecer durante a execução dos mesmos. Podem propagar-se para fora do método, sendo capturadas no local certo onde isso deve acontecer.

---

## Interface para TAD DevolucoesExemplar

# Relembremos: Devoluções de exemplares de livros

- No momento da devolução de um exemplar na Biblioteca, a funcionária poderá não ter tempo para repor o livro juntamente com os exemplares disponíveis
- O processo normal será que todos os exemplares devolvidos fiquem guardados juntos, empilhados ao lado da receção da biblioteca
- Assim, o processo de devolução tem dois passos:
  - Receber o exemplar, o que implica inserir o exemplar numa pilha de exemplares a repor. A funcionária poderá receber até 10 devoluções antes de fazer, obrigatoriamente as reposições dos exemplares.
  - Mais tarde, com tempo, repor o exemplar no seu lugar. Neste caso, a funcionária irá, naturalmente, retirar o exemplar que está no topo da pilha de reposição, alterar a informação que lhe está associada (tornando-o disponível para empréstimo) e reinseri-lo no conjunto de cópias disponíveis.

# Interface CopyReturn

```
package library;

public interface CopyReturn {

    int size();

    boolean isEmpty();

    //checks if the maximum number of returns has been reached
    boolean maxReturns();

    //copy is added do stack of books to return
    //Requires: !maxReturns()
    void receiveCopy(Copy copy) throws MaxReturnsException;

    //most recent copy to be returned is made available
    //Requires: !isEmpty()
    Copy returnCopy() throws NoReturnsException;

}
```

Tipo associado a  
Biblioteca (Library)

MaxReturnsException:  
Exceção do domínio

# Implementação com Stack - CopyReturnClass (1)

```
package library;
import dataStructures.*;

public class CopyReturnsClass implements CopyReturn {
    public static final int MAX_RETURNS=10;
    protected Stack<Copy> returns;

    public CopyReturnsClass(){
        returns=new StackInArray<>(MAX_RETURNS);
    }

    public int size(){
        return returns.size();
    }

    public boolean isEmpty(){
        return returns.isEmpty();
    }

    public boolean maxReturns(){
        return this.size()== MAX_RETURNS;
    }
}
```

# Implementação com Stack - CopyReturnClass (1)

```
package library;
import dataStructures.*;

public class CopyReturnsClass implements CopyReturn {
    public static final int MAX_RETURNS=10;
    protected Stack<Copy> returns;

    public CopyReturnsClass(){
        returns=new StackInArray<>(MAX_RETURNS);
    }

    public int size(){
        return returns.size();
    }

    public boolean isEmpty(){
        return returns.isEmpty();
    }

    public boolean maxReturns(){
        return this.size()== MAX_RETURNS;
    }
}
```

Uso do operador  
diamante.

Para que serve ?



# Implementação com Stack - CopyReturnClass (1)

```
package library;
import dataStructures.*;

public class CopyReturnsClass implements CopyReturn {
    public static final int MAX_RETURNS=10;
    protected Stack<Copy> returns;

    public CopyReturnsClass(){
        returns=new StackInArray<>(MAX_RETURNS);
    }

    public int size(){
        return returns.size();
    }

    public boolean isEmpty(){
        return returns.isEmpty();
    }

    public boolean maxReturns(){
        return this.size()== MAX_RETURNS;
    }
}
```

Uso do operador diamante.

Este operador permite inferir qual o tipo que será contido na criação do objeto StackInArray, a partir do tipo associado à variável returns.

## Implementação com Stack - CopyReturnClass (2)

```
public boolean isFull() {  
    return ((StackInArray<Copy>)returns).isFull();  
}
```

Não é necessária  
na classe

```
//Requires: !this.maxReturns()  
public void receiveCopy(Copy copy) throws  
    MaxReturnsException {  
    if (this.maxReturns())  
        throw new MaxReturnsException();  
    returns.push(copy);  
}
```

Devia ser sempre  
possível ?

```
//Requires: !this.isEmpty()  
public Copy returnCopy() throws NoReturnsException{  
    if (this.isEmpty())  
        throw new NoReturnsException();  
    Copy c=returns.pop();  
    ((LibrarianCopy)c).setSituation(true);  
    return c;  
}
```

returnCopy  
actualiza o  
estado de c e  
devolve-o

# Reservas de livros

- O processo de requisição de uma cópia de um livro implica que um leitor da biblioteca reserve o mesmo.
- As reservas de um livro são organizadas como uma fila de espera, o primeiro leitor a reservar terá acesso ao primeiro exemplar do livro que ficar disponível para requisição
- Necessária a implementação do TAD Leitor, que será o tipo a guardar na fila de espera de reservas de um livro.



---

## TAD Fila– Elementos do tipo E

# TAD Fila de Elementos do Tipo E

```
// Retorna true sse a fila estiver vazia.
```

```
boolean vazia( );
```

```
// Coloca o elemento especificado na retaguarda da fila.
```

```
void insere( E elemento );
```

```
// Retira e retorna o elemento da frente da fila.
```

```
// Pré-condição: a fila não está vazia.
```

```
E remove( );
```

# Interface Fila de Elementos do Tipo E

```
package dataStructures;

public interface Queue<E> {

    // Returns true iff the queue contains no elements.
    boolean isEmpty( );

    // Returns the number of elements in the queue.
    int size( );

    // Inserts the specified element at the rear of the queue.
    void enqueue( E element );

    // Removes and returns the element at the front of the queue.
    E dequeue( ) throws EmptyQueueException;
}
```

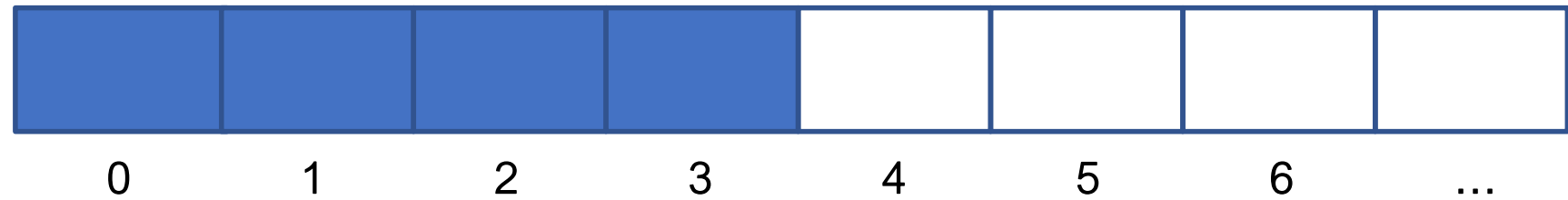
# Classes de Exceções da Fila

```
package dataStructures;
```

```
public class EmptyQueueException extends RuntimeException{  
}
```

```
public class FullQueueException extends RuntimeException{  
}
```

# Fila em Vetor



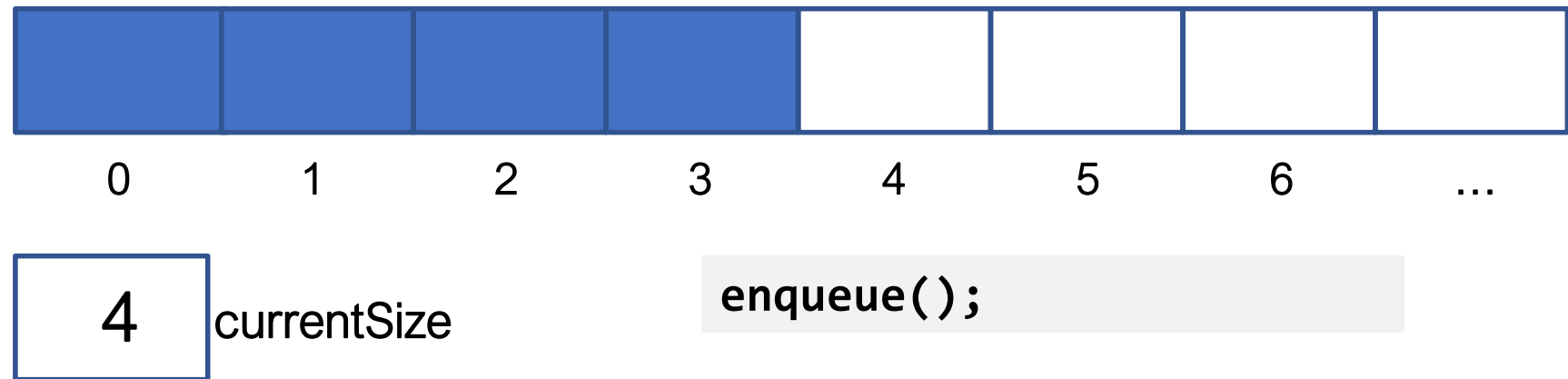
4 currentSize

Dúvidas:

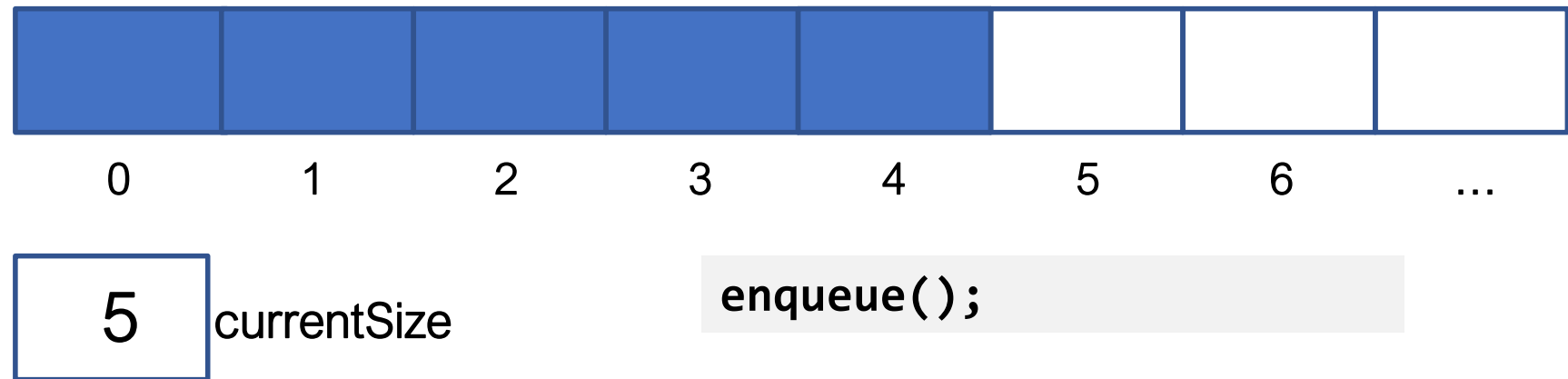
1. Como se implementa enqueue?
2. Como se implementa dequeue?
3. Como melhorar isto ?



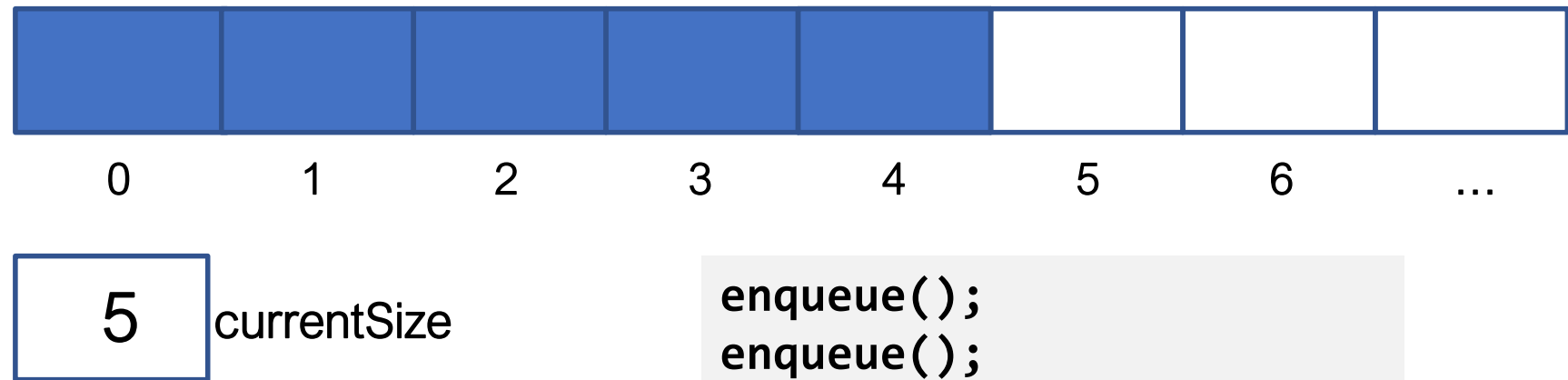
# Fila em Vetor



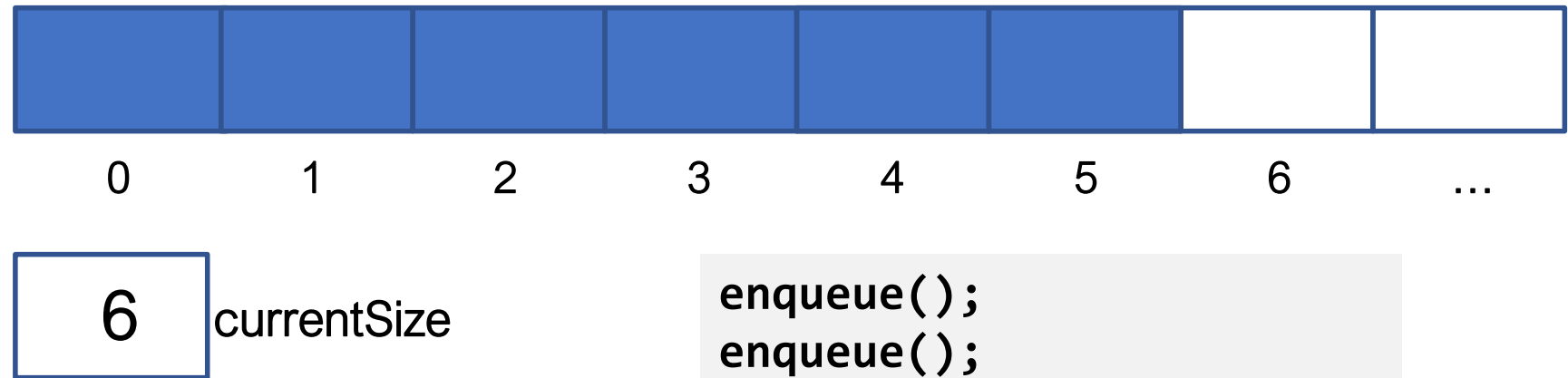
# Fila em Vetor



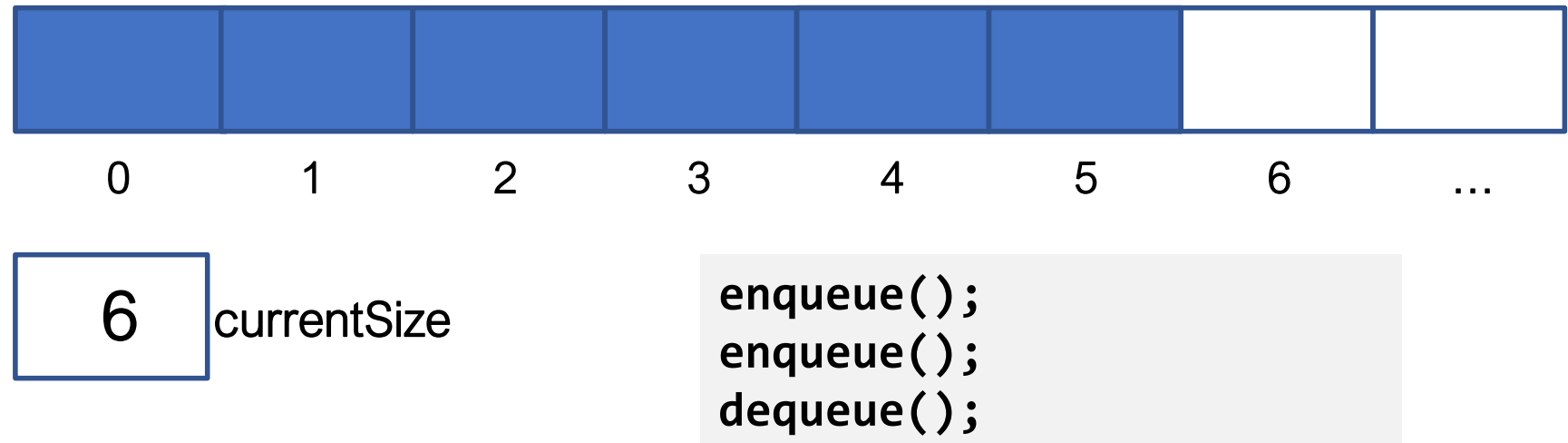
# Fila em Vetor



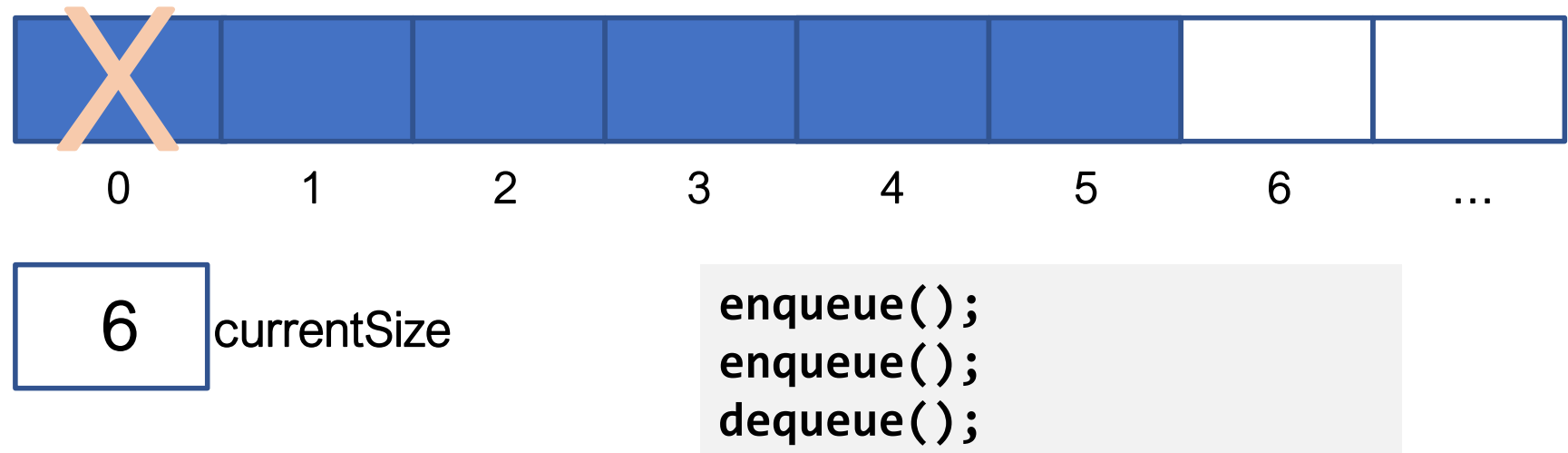
# Fila em Vetor



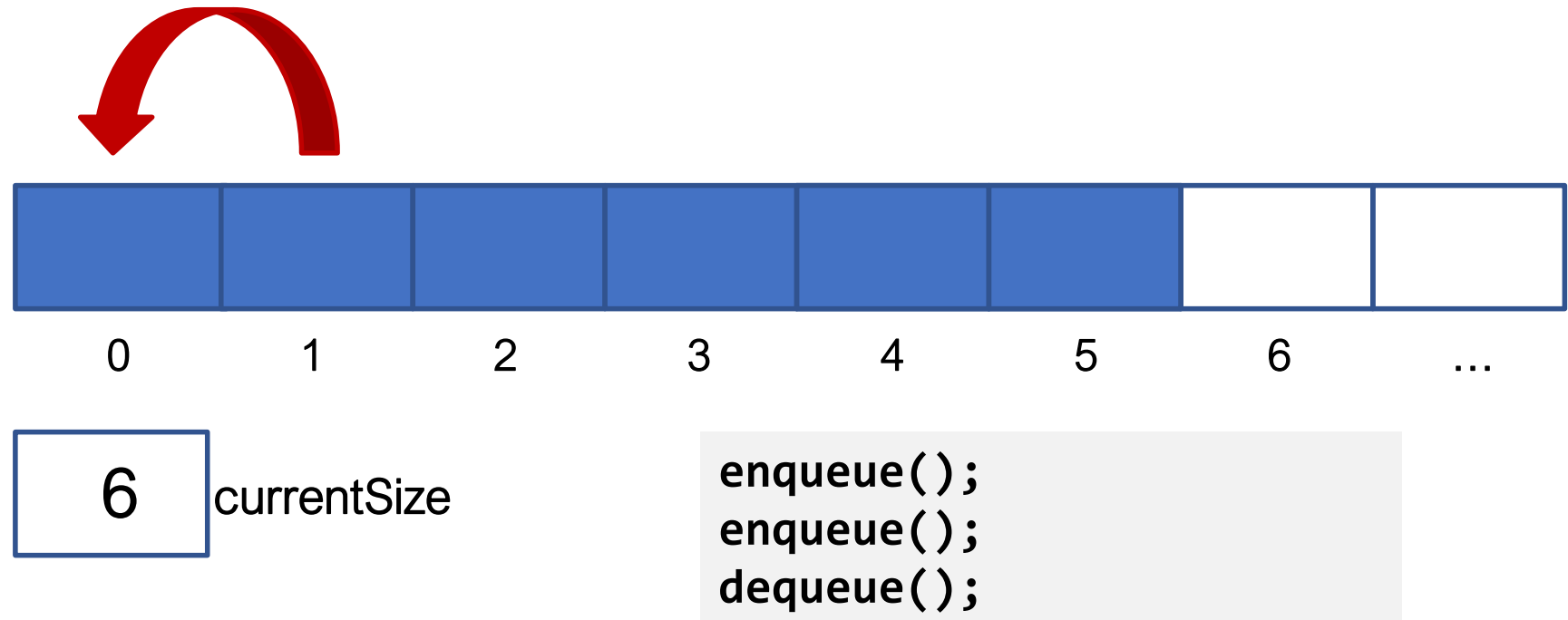
# Fila em Vetor



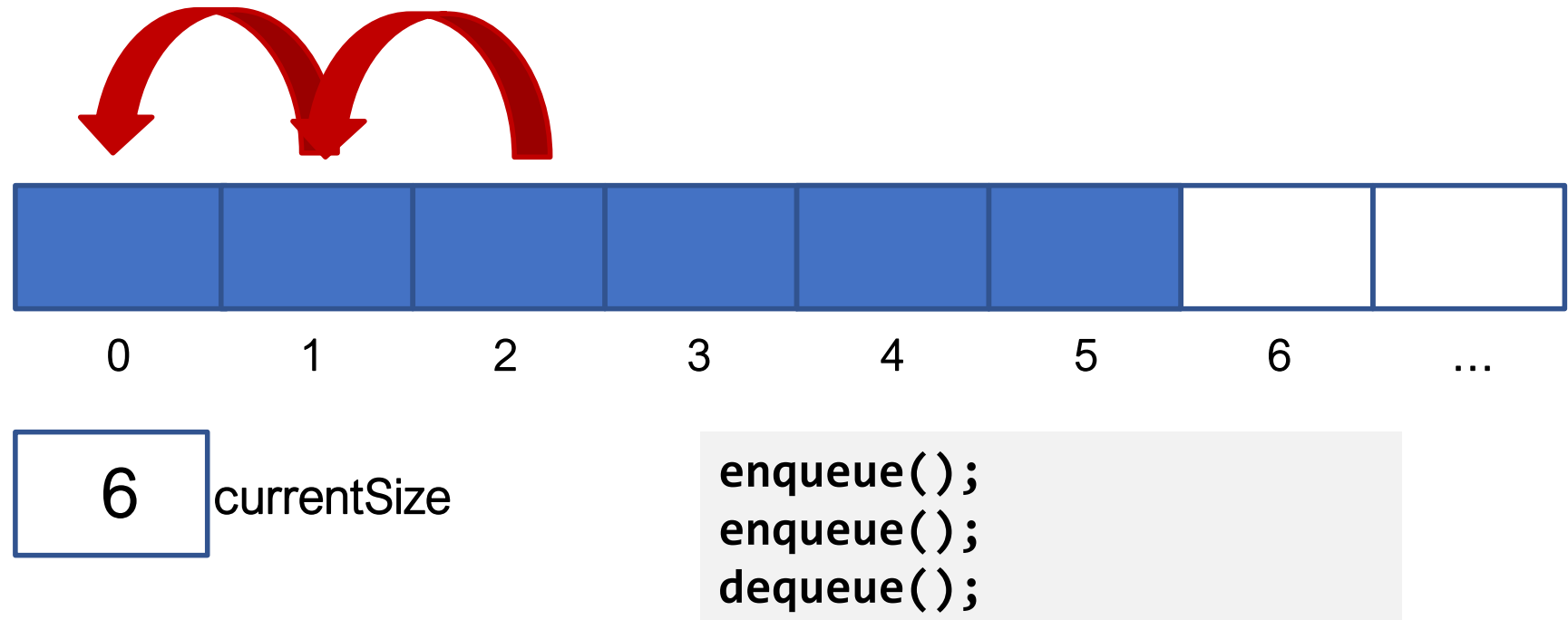
# Fila em Vetor



# Fila em Vetor

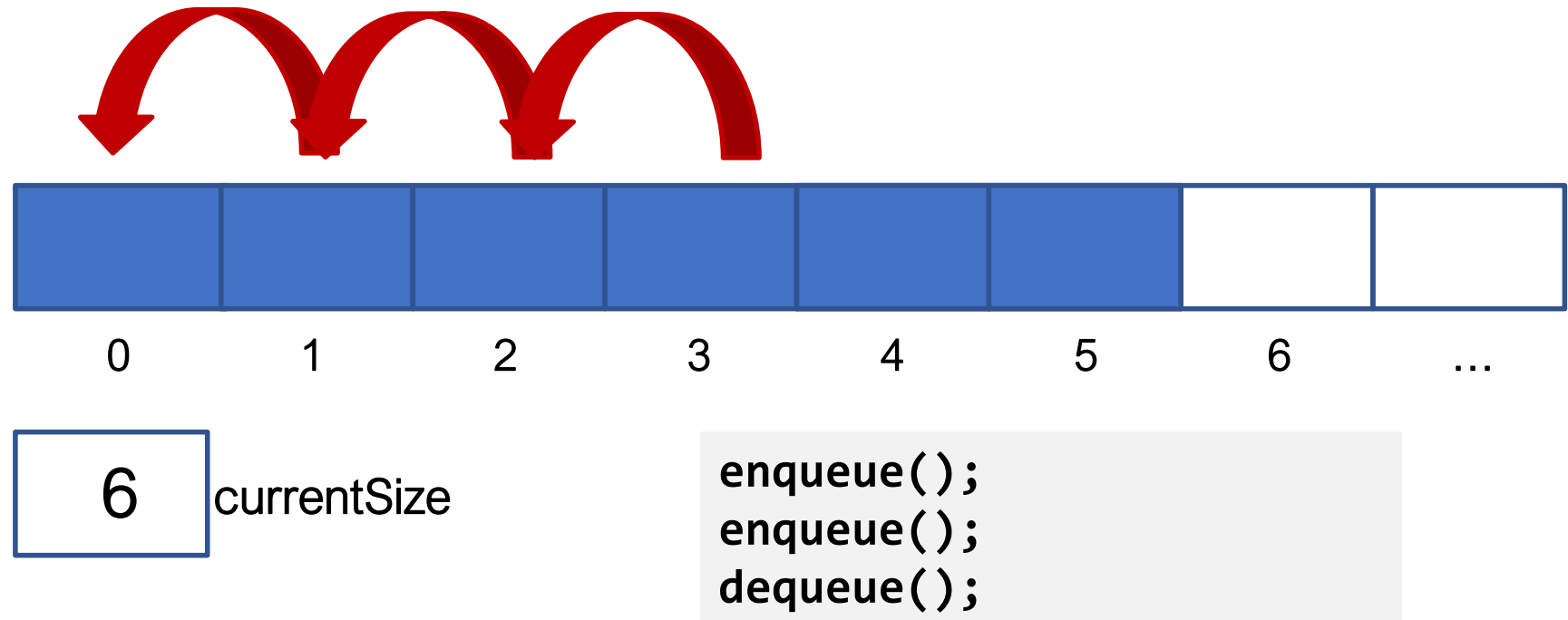


# Fila em Vetor

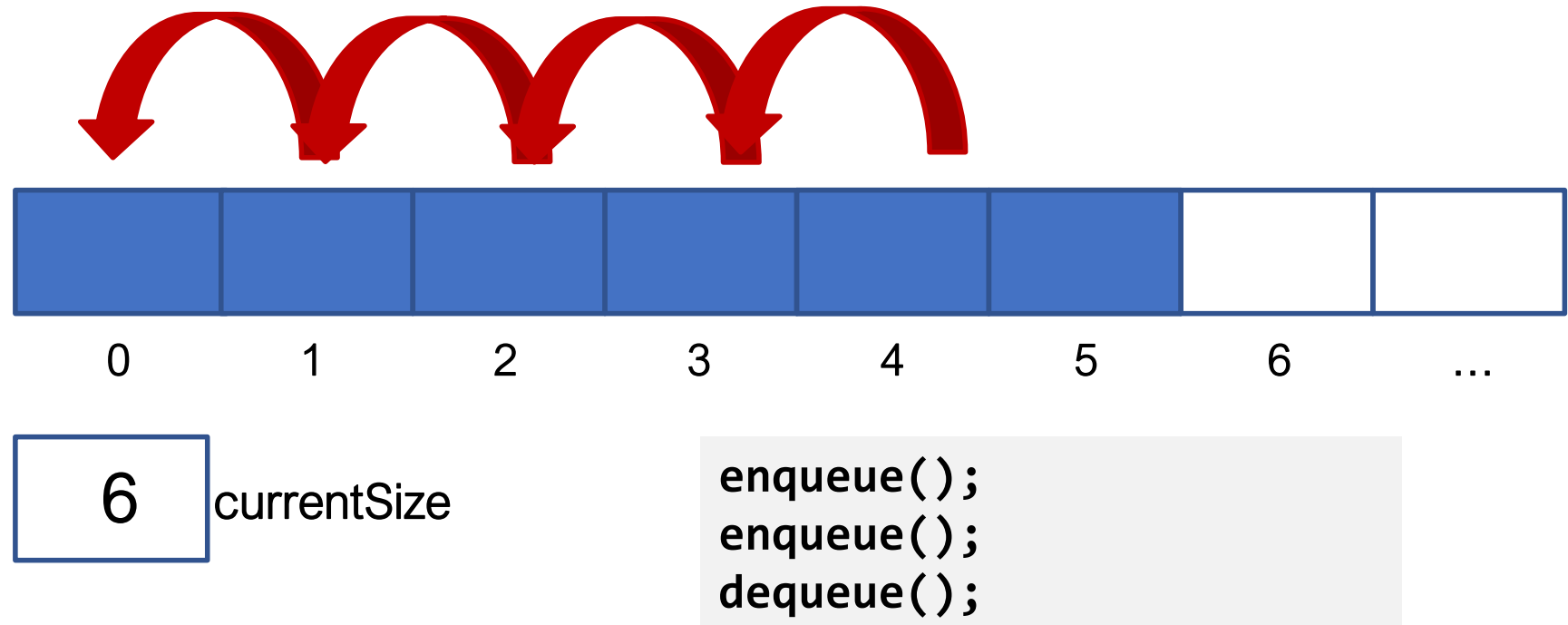




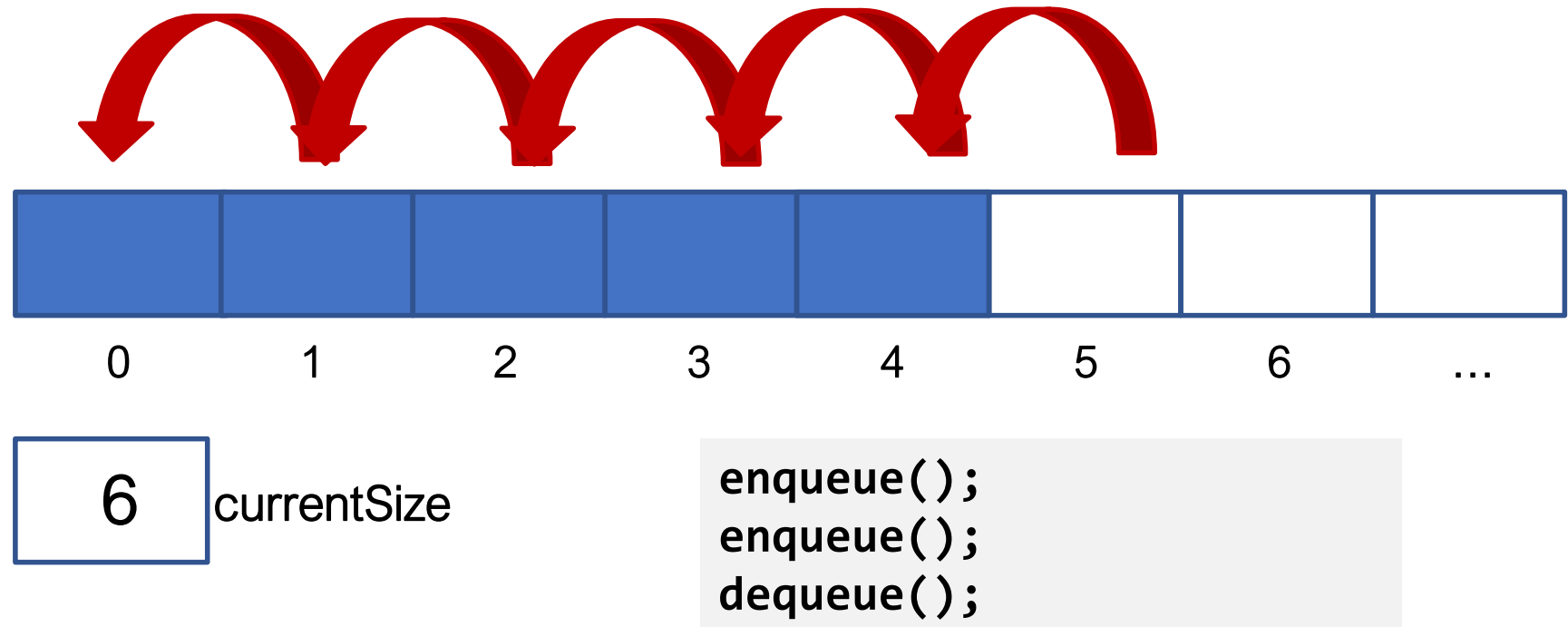
# Fila em Vetor



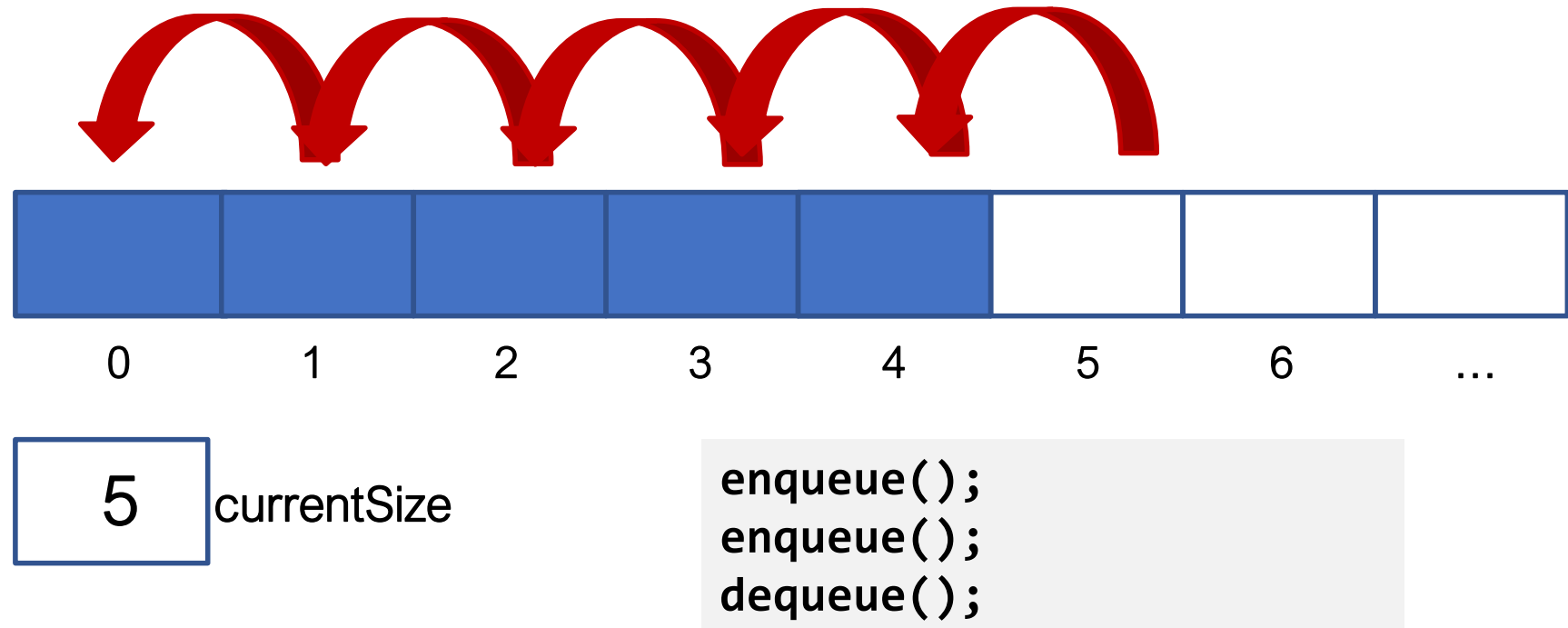
# Fila em Vetor



# Fila em Vetor

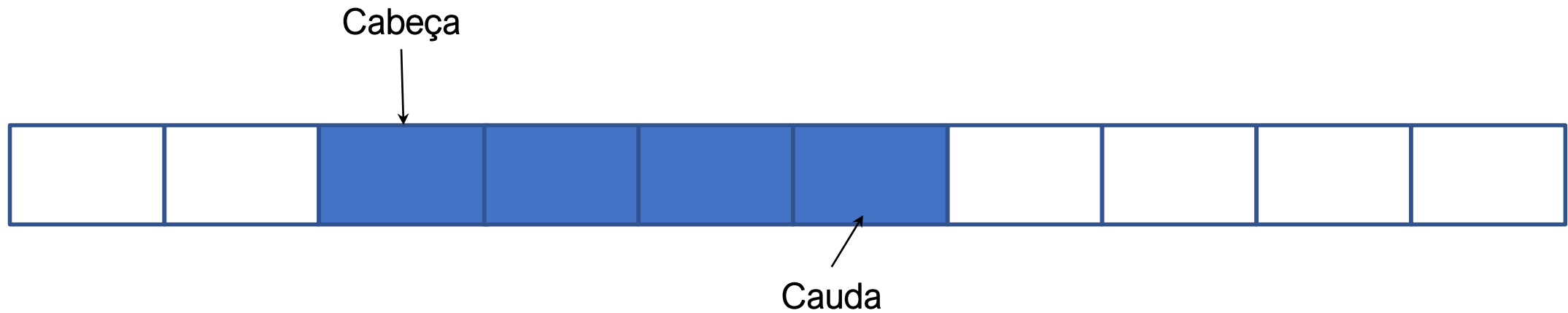


# Fila em Vetor

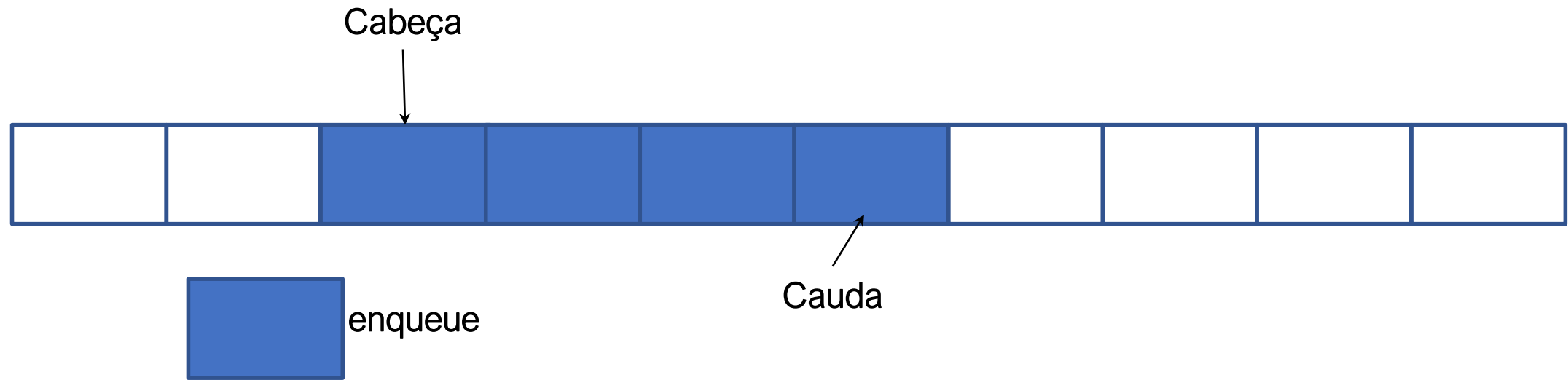


Qual o problema desta implementação ?

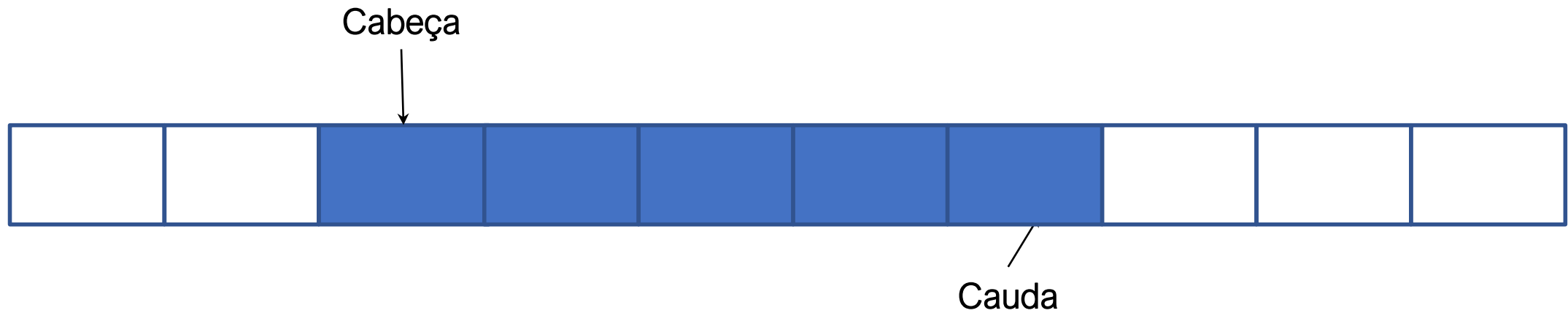
# Fila em Vetor circular



# Fila em Vetor circular

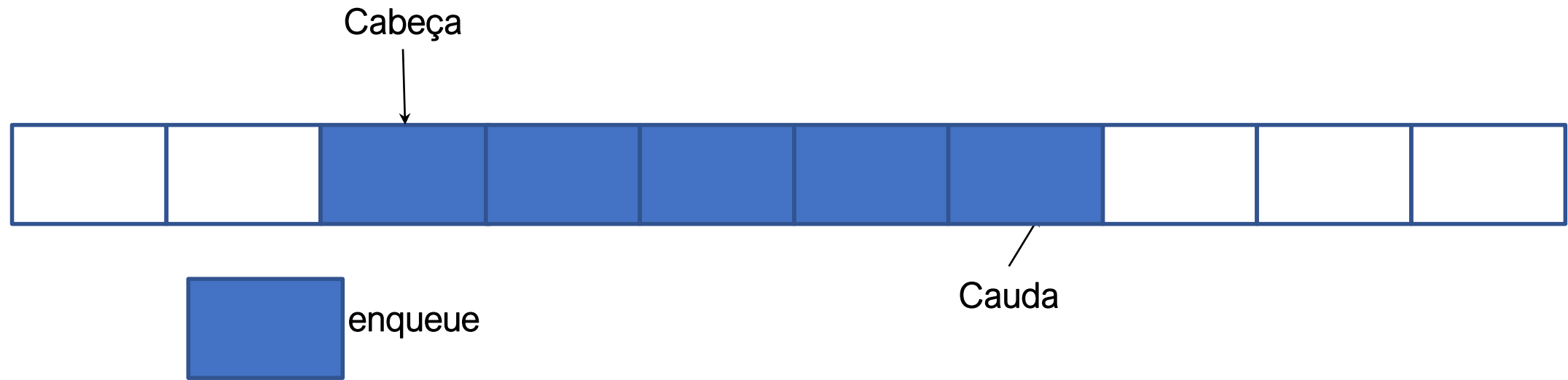


# Fila em Vetor circular



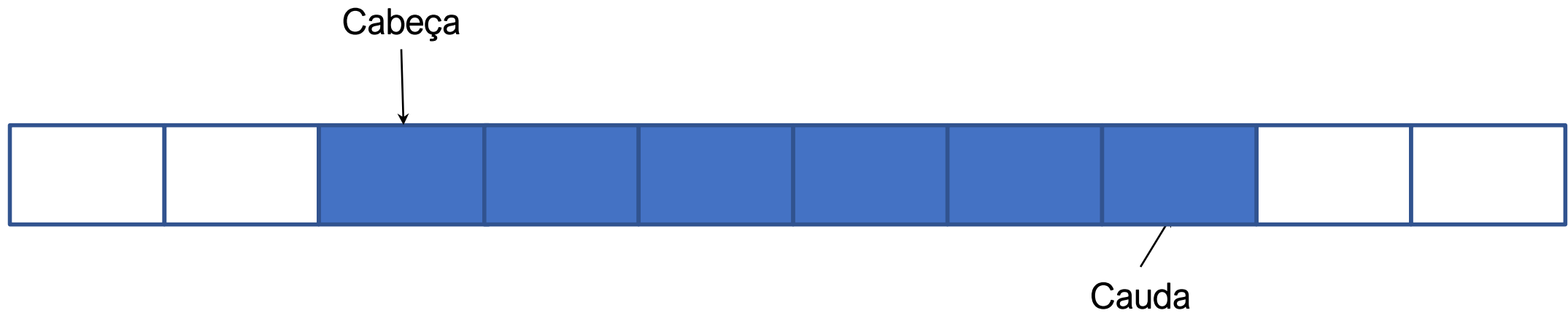
**`cauda = cauda + 1;`**

# Fila em Vetor circular





# Fila em Vetor circular

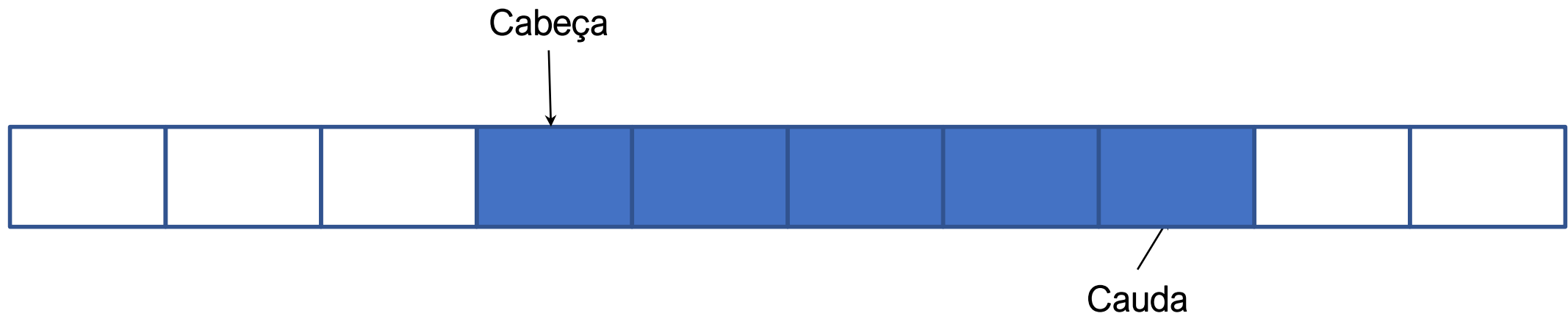


**`cauda = cauda + 1;`**

# Fila em Vetor circular

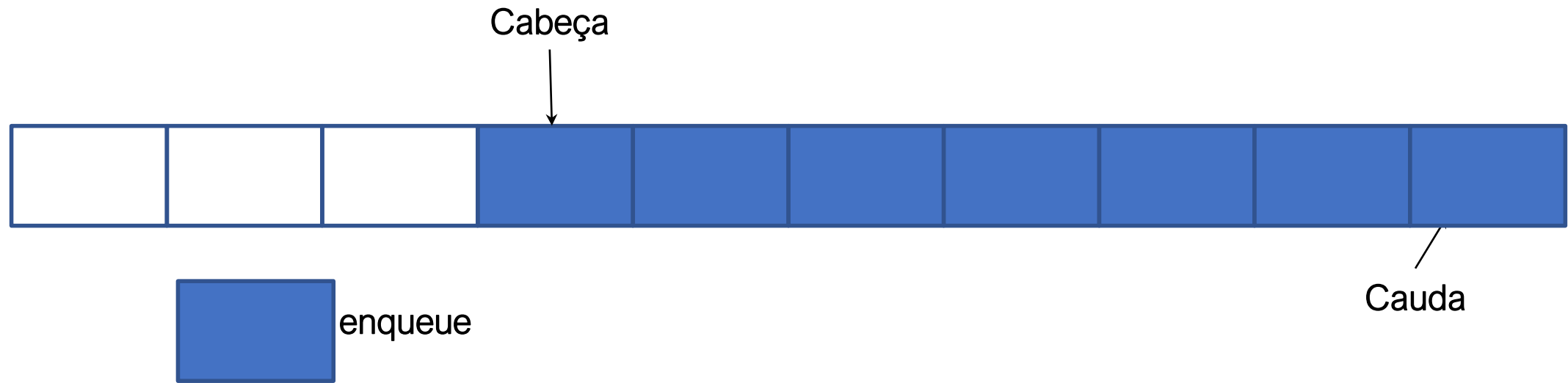


# Fila em Vetor circular



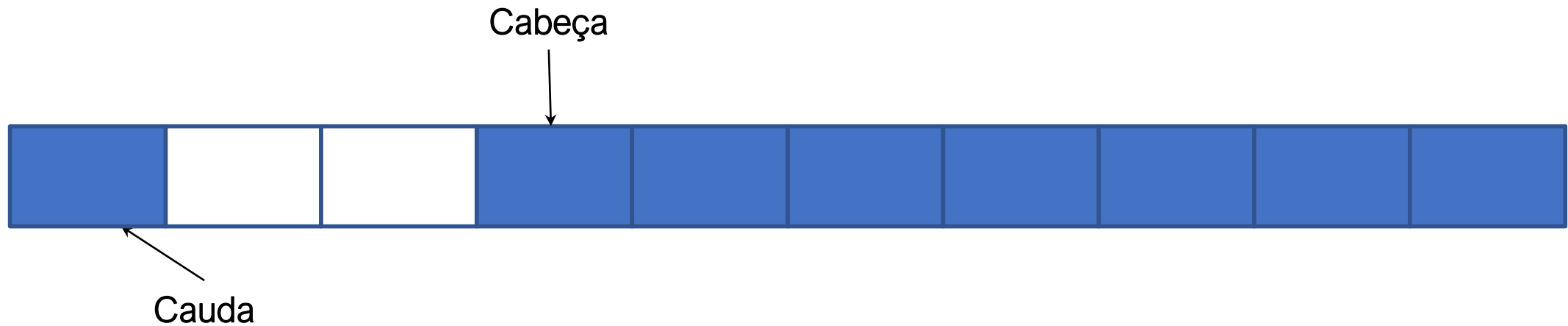
**`cabeca = cabeca + 1;`**

# Fila em Vetor circular



Como se calcula a próxima posição da cauda ?

# Fila em Vetor circular

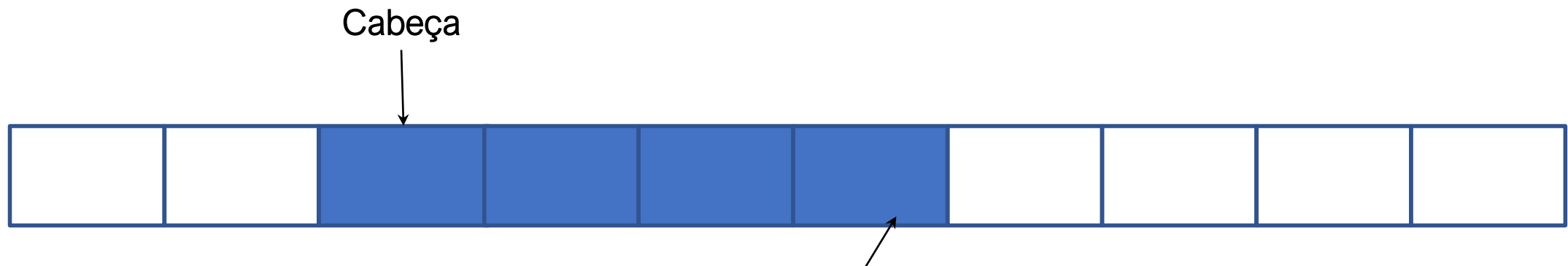
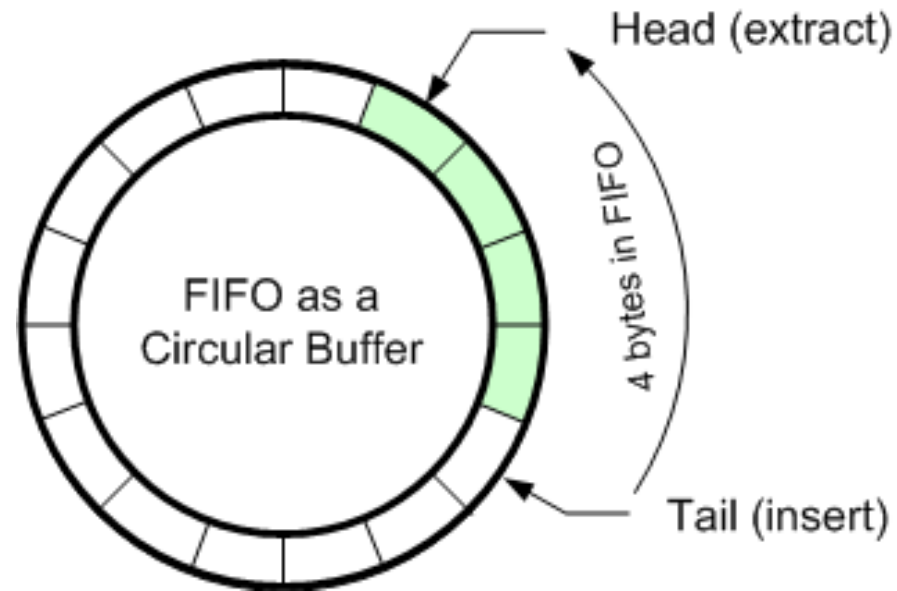


Como se calcula a próxima posição da cauda ?

Temos de ter em conta a dimensão total do vetor  
(neste caso **length**)

# Vetor circular

Qual a vantagem do vetor circular ?



# Classe Fila em Vetor (1)

```
package dataStructures;

public class QueueInArray<E> implements Queue<E> {

    // Default capacity of the queue.
    public static final int DEFAULT_CAPACITY = 1000;

    // Memory of the queue: a circular array.
    protected E[] array;

    // Index of the element at the front of the queue.
    protected int front;

    // Index of the element at the rear of the queue.
    protected int rear;

    // Number of elements in the queue.
    protected int currentSize;
```

# Classe Fila em Vetor (2)

```
@SuppressWarnings("unchecked")
public QueueInArray( int capacity ) {
    // Compiler gives a warning.
    array = (E[]) new Object[capacity];
    front = 0;
    rear = capacity - 1;
    currentSize = 0;
}
```

Porque precisamos disto ?

```
public QueueInArray( ) {
    this(DEFAULT_CAPACITY);
}
```



# Classe Fila em Vetor (3)

```
// Returns true if the queue contains no elements.  
public boolean isEmpty( ){  
    return currentSize == 0;  
}  
  
// Returns true if the queue cannot contain more elements.  
public boolean isFull( ){  
    return currentSize == array.length;  
}  
  
// Returns the number of elements in the queue.  
public int size( ) {  
    return currentSize;  
}
```

# Classe Fila em Vetor (4)

```
// Increments with “wrap around”.
protected int nextIndex( int index ){
    return ( index + 1 ) % array.length;
}

// Inserts the specified element at the rear of the queue.
public void enqueue( E element ) throws FullQueueException{
    if ( this.isFull() )
        throw new FullQueueException();
    rear = this.nextIndex(rear);
    array[rear] = element;
    currentSize++;
}
```

# Classe Fila em Vetor (5)

```
// Removes and returns the element at the front of the queue.  
public E dequeue( ) throws EmptyQueueException{  
    if ( this.isEmpty() )  
        throw new EmptyQueueException();  
    E element = array[front];  
    array[front] = null; // For garbage collection.  
    front = this.nextIndex(front);  
    currentSize--;  
    return element;  
}  
} // End of QueueInArray.
```

---

Interface para TAD LivroBiblioteca (extensão não pública de Livro contendo fila de espera de Reservas)

# LibrarianBook

```
package library;

interface LibrarianBook extends Book {

    //returns the number of reservations of
    //the book - could be in Book
    public int numberReservations();

    //true if the book has no reservations - could be in Book
    public boolean noReservations();

    //removes reader from reservation queue
    //Requires: !noReservations()
    Reader oldestReservation() throws NoReservationsException;

    //adds reader to reservation queue
    void addReservation(Reader reader);
    ...
    //Other methods in the ADT are ommitted...
}
```

Implementação de Reader deve conter (e.g.)

- Nome
- Número Leitor
- Categoria
- Data de inscrição
- Validade

Interface não público – altera a fila de reservas apenas dentro do pacote

# LibrarianBookClass (1) – não completa


```
package library;
import dataStructures.*;

class LibrarianBookClass implements LibrarianBook {
    public static final int MAX_RESERVATIONS=5;
    protected String author;
    ...
    protected Queue<Reader> reservations;

    public LibrarianBookClass(String author, long ISBN,
                               String title, String subject, String code,
                               String publisher) {
        this.author=author;
        ...
        reservations=new QueueInArray<>(MAX_RESERVATIONS);
    }
}
```

## LibrarianBookClass (2) – não completa

Podem estar no Tipo Book  
A implementação está bem aqui.



```
//true if the book has no reservations
public boolean noReservations() {
    return this.numberReservations()==0;
}

//returns the number of reservations of the book
public int numberReservations() {
    return this.reservations.size();
}

//true if the maximum number of reservations has been reached.
public boolean maxReservations() {
    return this.size() == MAX_RESERVATIONS;
}
```

## LibrarianBookClass (3) – não completa

```
//removes reader from reservation queue
//Requires: !noReservations()
public Reader oldestReservation()
    throws NoReservationsException{
    if (this.noReservations())
        throw new NoReservationsException();
    Reader reader = reservations.dequeue();
    return reader;
}
```

```
//adds reader to reservation queue
//Requires: !maxReservations()
public void addReservation(Reader reader)
    throws MaxReservationsException{
    if (this.maxReservations())
        throw new MaxReservationsException();
    reservations.enqueue(reader);
}
```





---

TAD Lista (Sequência de elementos com acesso por posição) –  
Elementos do tipo E

# TAD Lista de Elementos do Tipo E (1)

```
// Retorna o número de elementos na lista.
```

```
int dimensao( );
```

```
// Retorna o elemento que está na posição especificada.
```

```
// Pré-condição: a posição tem de ser válida.
```

```
E acede( int posição );
```

```
// Coloca o elemento especificado na posição especificada.
```

```
// Pré-condição: a posição tem de ser válida.
```

```
void insere( int posição, E elemento );
```

```
// Remove e retorna o elemento que está na posição especificada.
```

```
// Pré-condição: a posição tem de ser válida.
```

```
E remove( int posição );
```

# TAD Lista de Elementos do Tipo E (2)

```
// Retorna a posição na lista da primeira ocorrência  
// do elemento especificado, se a lista contiver o elemento.  
// No caso contrário, retorna -1.  
int pesquisa( E elemento );
```

```
// Remove da lista a primeira ocorrência do elemento  
// especificado e retorna true, se a lista contiver o elemento.  
// No caso contrário, retorna false.  
boolean remove( E elemento );
```

# Interface Lista de Elementos do Tipo E (1)

```
package dataStructures;  
public interface List<E>  
{  
    // Returns true iff the list contains no elements.  
    boolean isEmpty( );  
  
    // Returns the number of elements in the list.  
    int size( );  
  
    // Returns an iterator of the elements in the list  
    // (in proper sequence).  
    Iterator<E> iterator( );  
}
```

# Interface Lista de Elementos do Tipo E (2)

```
// Returns the first element of the list.  
E getFirst( ) throws EmptyListException;
```

```
// Returns the last element of the list.  
E getLast( ) throws EmptyListException;
```

```
// Returns the element at the specified position in the list.  
// Range of valid positions: 0, ..., size()-1.  
// If the specified position is 0, get corresponds to getFirst.  
// If the specified position is size()-1, get corresponds to getLast.  
E get( int position ) throws InvalidPositionException;
```

# Interface Lista de Elementos do Tipo E (3)

```
// Inserts the specified element at the first position in the list.  
void addFirst( E element );
```

```
// Inserts the specified element at the last position in the list.  
void addLast( E element );
```

```
// Inserts the specified element at the specified position in the list.  
// Range of valid positions: 0, ..., size().  
// If the specified position is 0, add corresponds to addFirst.  
// If the specified position is size(), add corresponds to addLast.  
void add( int position, E element ) throws InvalidPositionException;
```

# Interface Lista de Elementos do Tipo E (4)

```
// Removes and returns the element at the first position  
// in the list.
```

```
E removeFirst( ) throws EmptyListException;
```

```
// Removes and returns the element at the last position  
// in the list.
```

```
E removeLast( ) throws EmptyListException;
```

```
// Removes and returns the element at the specified position  
// in the list.
```

```
// Range of valid positions: 0, ..., size()-1.
```

```
// If the specified position is 0, remove corresponds to removeFirst.
```

```
// If the specified position is size()-1, remove corresponds to  
// removeLast.
```

```
E remove( int position ) throws InvalidPositionException;
```



# Interface Lista de Elementos do Tipo E (5)

```
// Returns the position of the first occurrence of the specified
// element in the list, if the list contains the element.
// Otherwise, returns -1.
int find( E element );

// Removes the first occurrence of the specified element from the
// list and returns true, if the list contains the element.
// Otherwise, returns false.
boolean remove( E element );

} // End of List.
```

# Classes de Exceções da Lista

```
package dataStructures;
```

```
public class EmptyListException extends RuntimeException{  
}
```

```
public class InvalidPositionException extends RuntimeException{  
}
```

---

## TAD Iterador (Percurso Sequencial) – Elementos do tipo E

# Interface Iterador de Elementos do Tipo E

```
package dataStructures;
public interface Iterator<E>{

    // Returns true iff the iteration has more elements.
    // In other words, returns true if a call to next()
    // would return an element instead of throwing an exception.
    boolean hasNext( );

    // Returns the next element in the iteration.
    E next( ) throws NoSuchElementException;

    // Restarts the iteration.
    // After rewind, if the iteration is not empty,
    // next() will return the first element in the iteration.
    void rewind( );
}
```

# Classes de Exceções do Iterador

```
package dataStructures;
```

```
public class NoSuchElementException extends RuntimeException{  
}
```

---

## TAD Iterador Bidirecional (Percurso Sequencial) - Elementos do tipo E

# Interface Iterador Bidireccional – Elementos E

```
package dataStructures;
public interface TwoWayIterator<E> extends Iterator<E>{

    // Returns true iff the iteration has more elements
    // in the reverse direction. In other words, returns true
    // if a call to previous() would return an element
    // instead of throwing an exception.
    boolean hasPrevious( );

    // Returns the previous element in the iteration.
    E previous( ) throws NoSuchElementException;

    // Restarts the iteration in the reverse direction.
    // After fullForward, if the iteration is not empty,
    // previous() will return the last element in the iteration.
    void fullForward( );
}
```

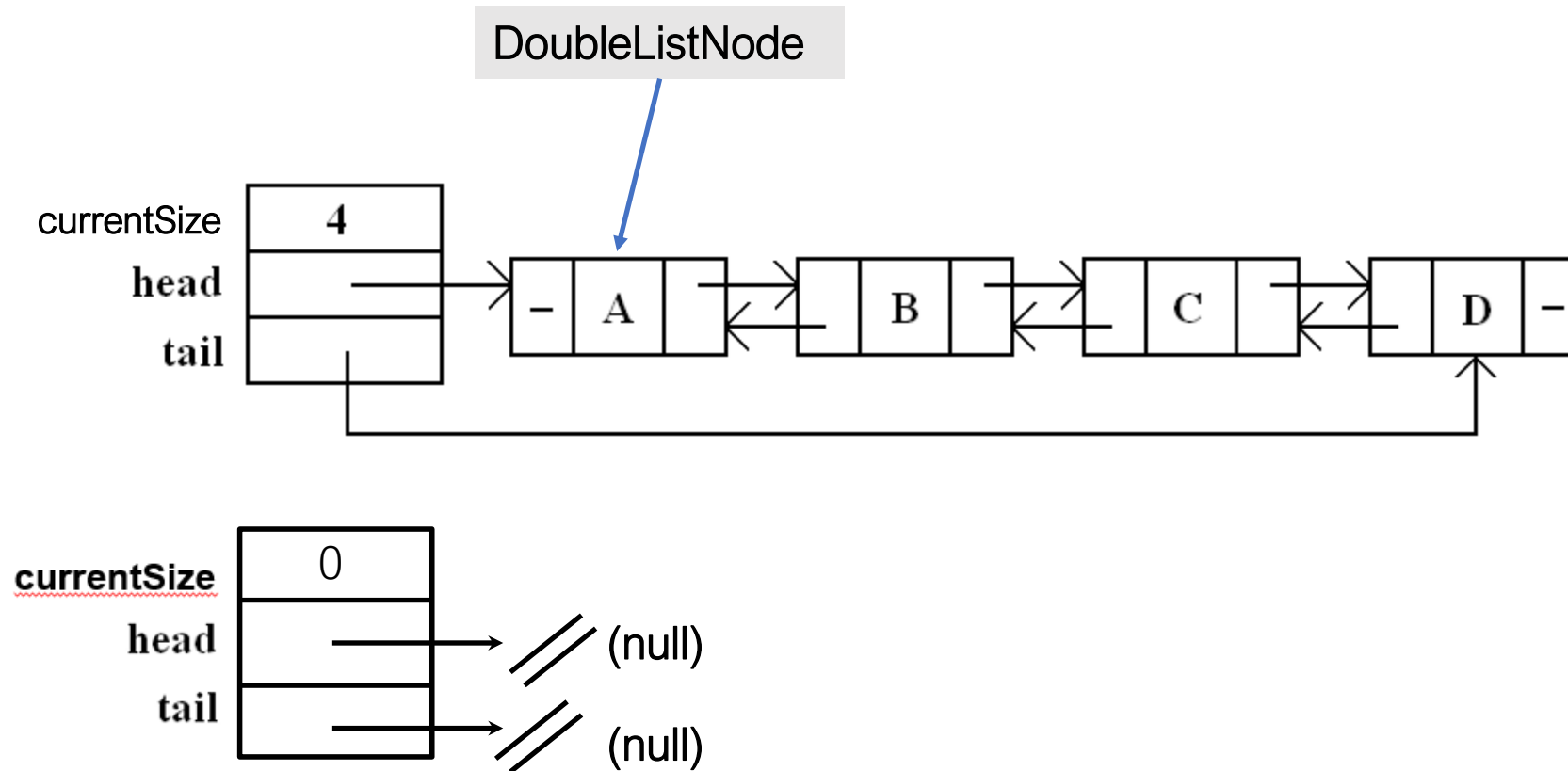
---

## Implementação de TAD Lista – elementos tipo E

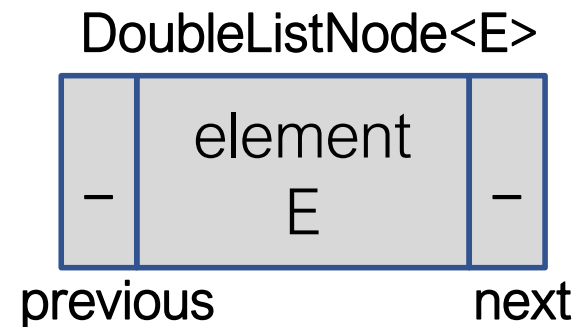
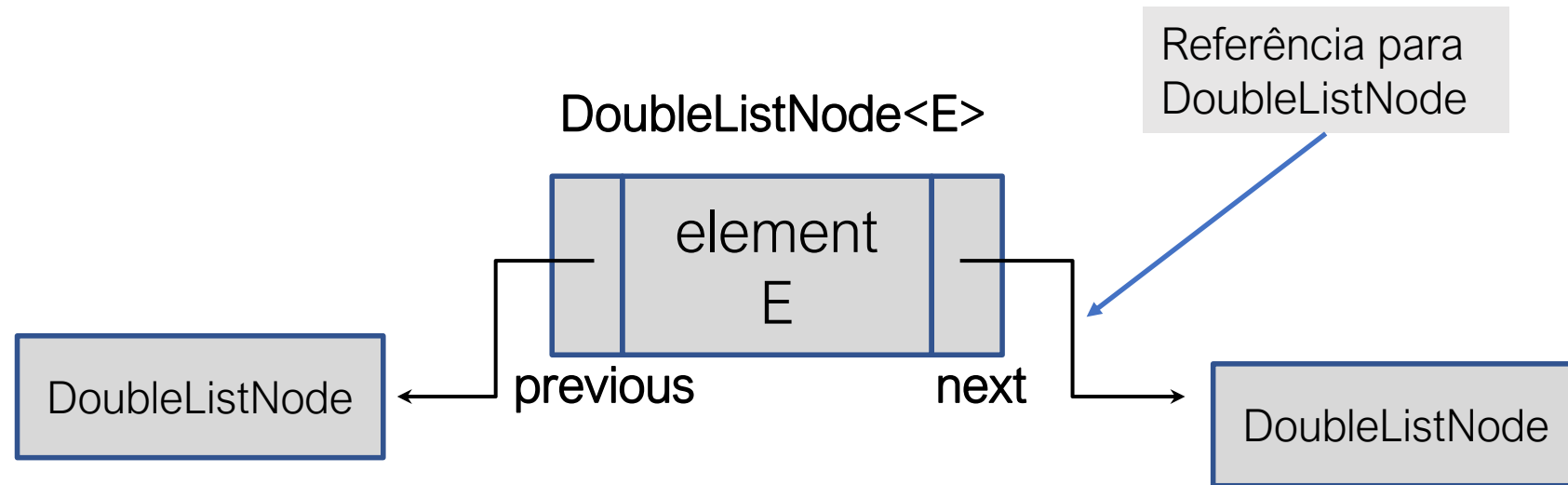
### Estrutura de Dados: Lista Duplamente Ligada (DLL)



# Lista Duplamente Ligada

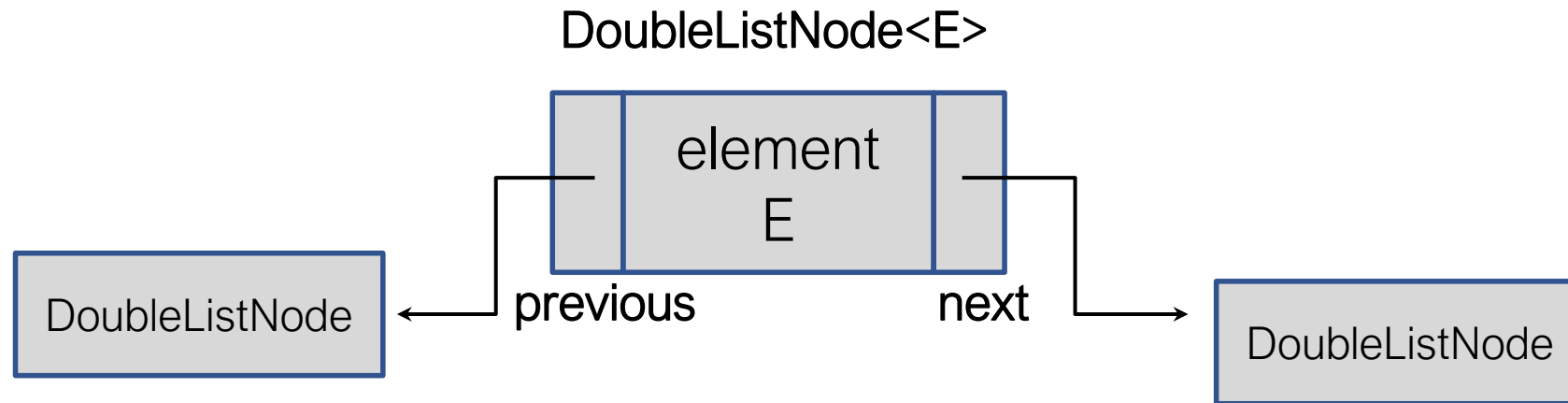


# Célula Dupla



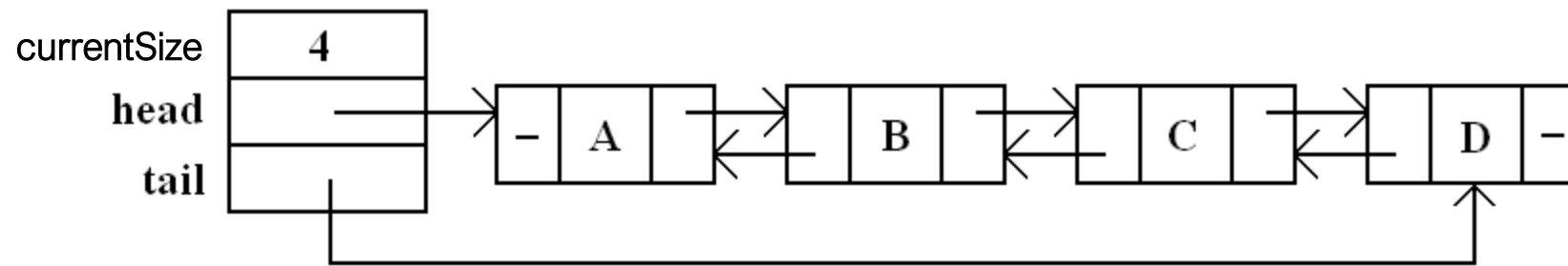
Célula sem ligações:  
next é NULL e previous é NULL

# Célula Dupla (Nó com conteúdo Tipo E)

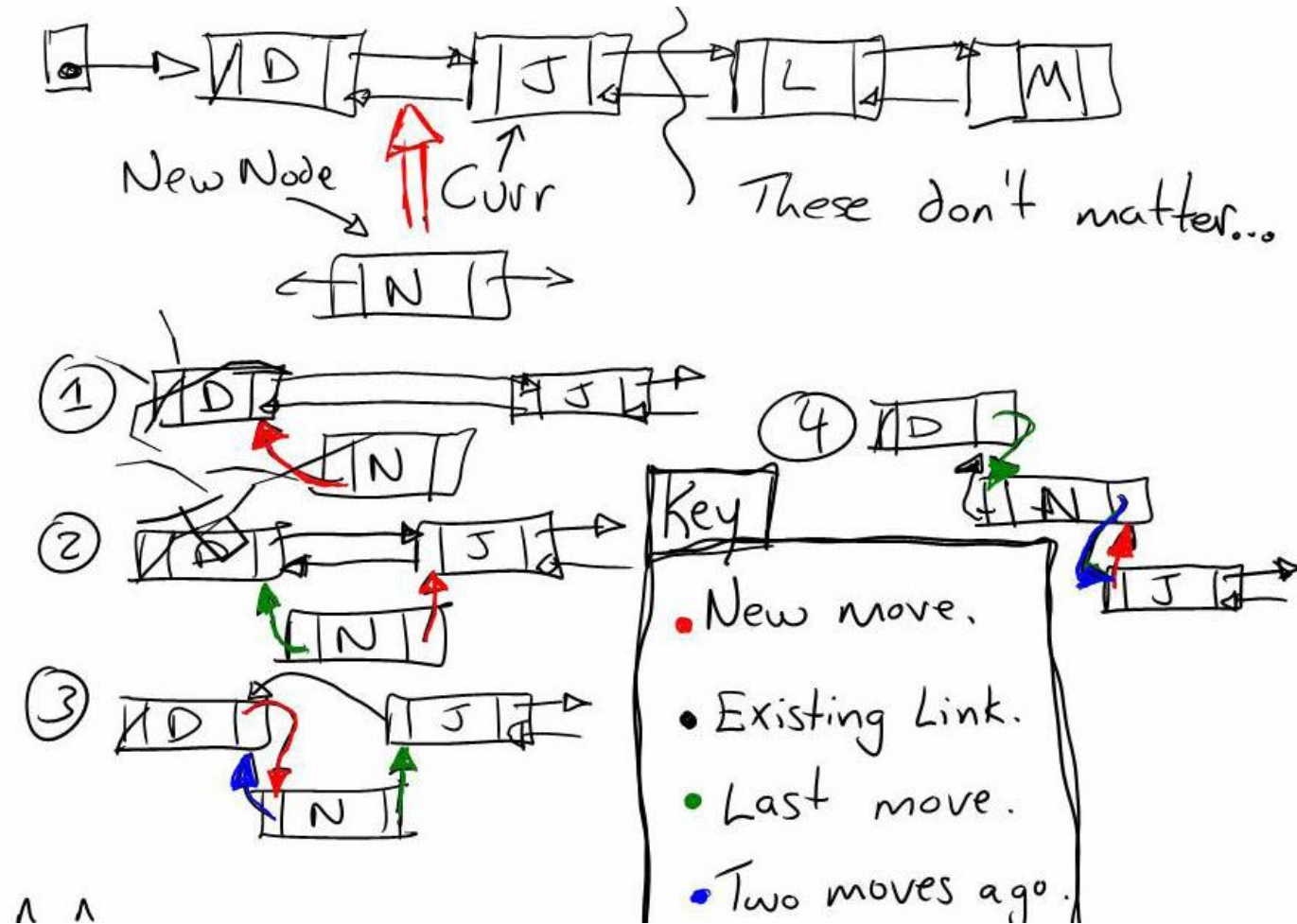


- Métodos:
  - `getNext()` – devolve o nó seguinte
  - `setNext()` – altera o nó seguinte (altera ligações)
  - `getPrevious()` – devolve o nó anterior
  - `setPrevious()` – altera o nó anterior (altera ligações)
  - `getElement()` – devolve o elemento contido no nó
  - `setElement()` – altera o elemento contido no nó

# Lista Duplamente Ligada



# DLL: O Imbróglio...



# Classe Lista Duplamente Ligada e Célula

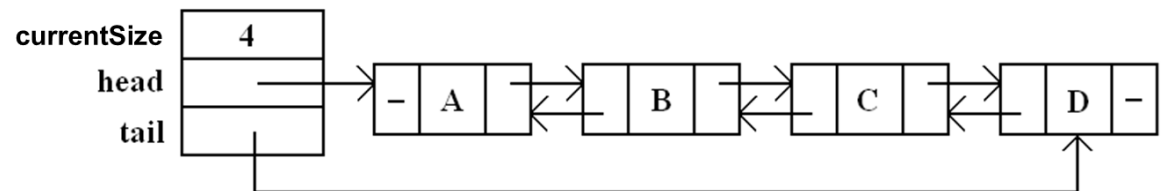
```
package dataStructures;
public class DoubleList<E> implements List<E>{
```

```
    static class DoubleListNode<E> ...{
        ...
    }
```

nested Class estática

```
    // Node at the head of the list.
    protected DoubleListNode<E> head;
    // Node at the tail of the list.
    protected DoubleListNode<E> tail;
    // Number of elements in the list.
    protected int currentSize;
```

```
public DoubleList( ){
    head = null;
    tail = null;
    currentSize = 0;
```



# Classe Célula Dupla de Elementos do Tipo E (1)

```
package dataStructures;  
public class DoubleList<E> implements List<E>{
```

...

```
    static class DoubleListNode<E>{  
  
        // Element stored in the node.  
        private E element;  
  
        // (Pointer to) the previous node.  
        private DoubleListNode<E> previous;  
  
        // (Pointer to) the next node.  
        private DoubleListNode<E> next;  
    }
```

...

Nested Class estática

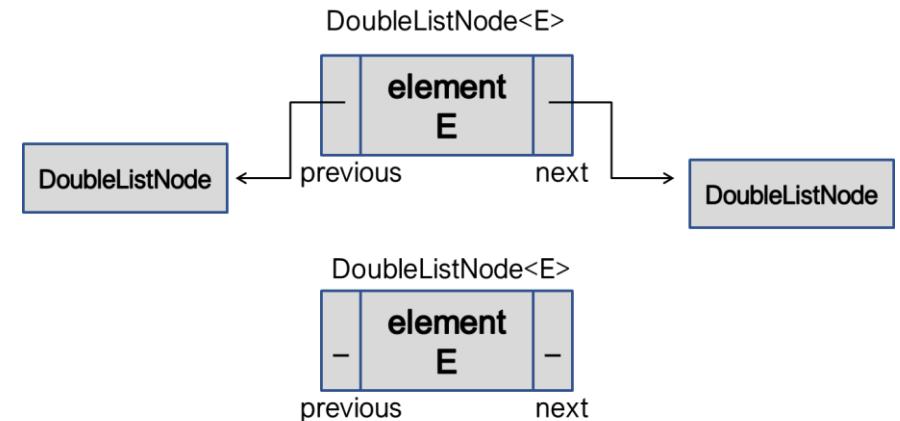


# Classe Célula Dupla (2)

```
public DoubleListNode( E theElement, DoubleListNode<E> thePrevious,  
                      DoubleListNode<E> theNext ){
```

```
    element = theElement;  
    previous = thePrevious;  
    next = theNext;  
}
```

```
public DoubleListNode( E theElement ){  
    this(theElement, null, null);  
}
```





# Classe Célula Dupla (3)

```
public E getElement( ){  
    return element;  
}
```

```
public DoubleListNode<E> getPrevious( ){  
    return previous;  
}
```

```
public DoubleListNode<E> getNext( ){  
    return next;  
}
```

# Classe Célula Dupla (4)

```
public void setElement( E newElement ){
    element = newElement;
}

public void setPrevious( DoubleListNode<E> newPrevious ){
    previous = newPrevious;
}

public void setNext( DoubleListNode<E> newNext ){
    next = newNext;
}

} // End of DoubleListNode.
```

# Classe Lista Duplamente Ligada e Célula

```
package dataStructures;
public class DoubleList<E> implements List<E>{

    static class DoubleListNode<E> ...{
        ...
    }

    // Node at the head of the list.
    protected DoubleListNode<E> head;
    // Node at the tail of the list.
    protected DoubleListNode<E> tail;
    // Number of elements in the list.
    protected int currentSize;

    public DoubleList( ){
        head = null;
        tail = null;
        currentSize = 0;
    }
}
```

# Classe Lista Duplamente Ligada (2)

```
// Returns true if the list contains no elements.  
public boolean isEmpty( ){  
    return currentSize == 0;  
}
```

```
// Returns the number of elements in the list.  
public int size( ){  
    return currentSize;  
}
```

# Classe Lista Duplamente Ligada (3)

```
// Returns the first element of the list.  
public E getFirst( ) throws EmptyListException{  
    if ( this.isEmpty() )  
        throw new EmptyListException();  
    return head.getElement();  
}
```

```
// Returns the last element of the list.  
public E getLast( ) throws EmptyListException{  
    //TODO: Left as an exercise.  
  
}
```

# Classe Lista Duplamente Ligada (6)

```
// Returns the element at the specified position in the list.  
// Range of valid positions: 0, ..., size() - 1.  
// If the specified position is 0, get corresponds to getFirst.  
// If the specified position is size()-1, get corresponds to  
//getLast.  
public E get( int position ) throws InvalidPositionException{  
    if ( position < 0 || position >= currentSize )  
        throw new InvalidPositionException();  
    // acesso ao elemento da lista em position.  
}
```

Como fazer ?

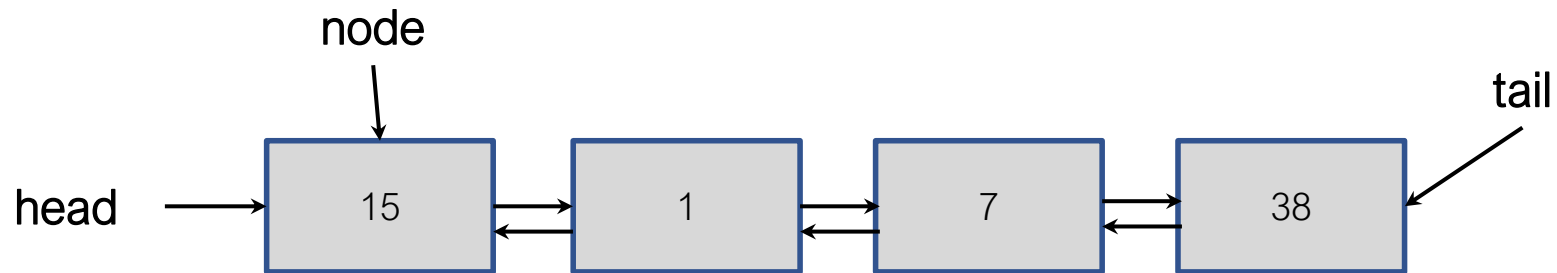
# Classe Lista Duplamente Ligada (4): getNode: Primeira Versão

```
// Returns the node at the specified position in the list.  
// Requires: position ranges from 0 to currentSize - 1.  
protected DoubleListNode<E> getNode( int position ){  
  
    DoubleListNode<E> node = head;  
  
    for ( int i = 0; i < position; i++ )  
        node = node.getNext();  
    return node;  
}
```

# getNode(): Primeira versão

position = 2

i = 0



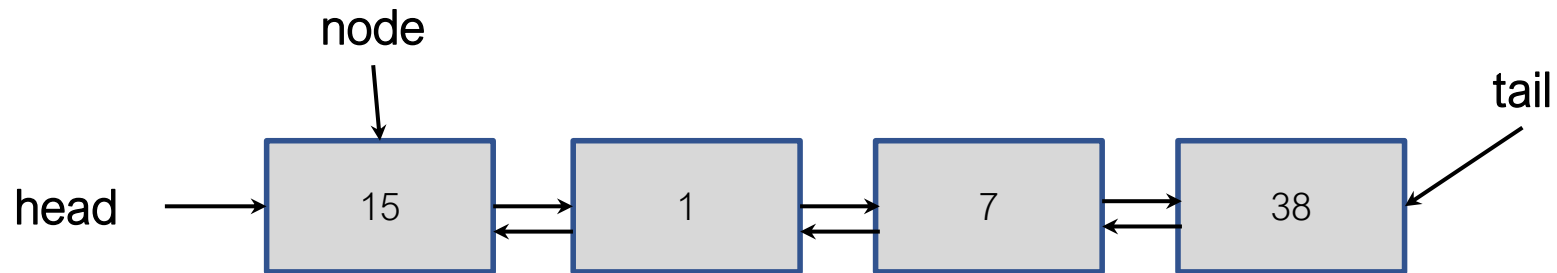
```
DoubleListNode<E> node = head;
```



# getNode(): Primeira versão

position = 2

i = 0

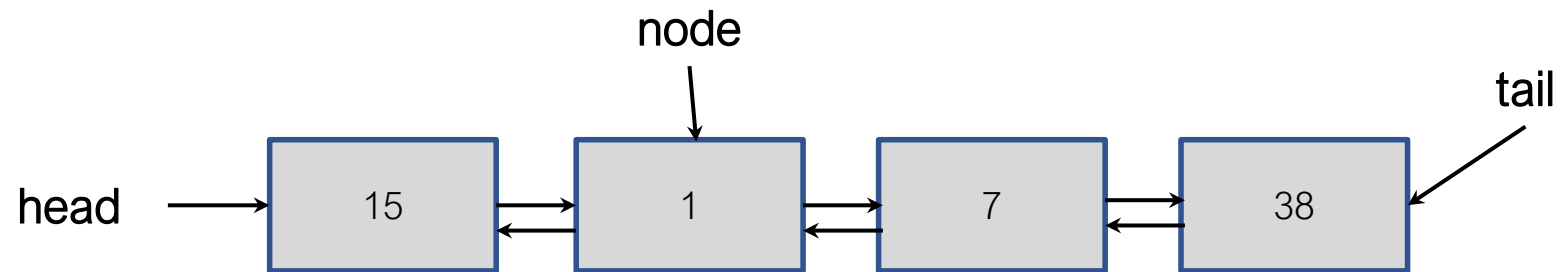


```
node = node.getNext();
```

# getNode(): Primeira versão

position = 2

i = 0

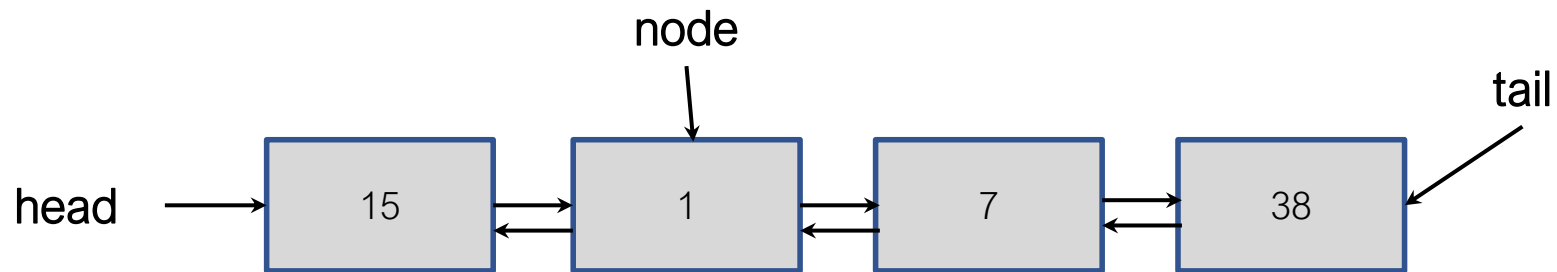


```
node = node.getNext();
```

# getNode(): Primeira versão

position = 2

i = 1

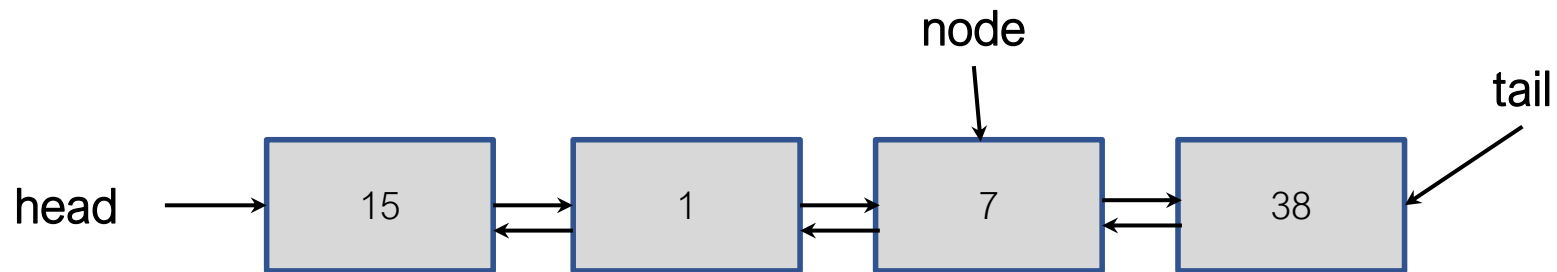


```
node = node.getNext();
```

# getNode(): Primeira versão

position = 2

i = 1

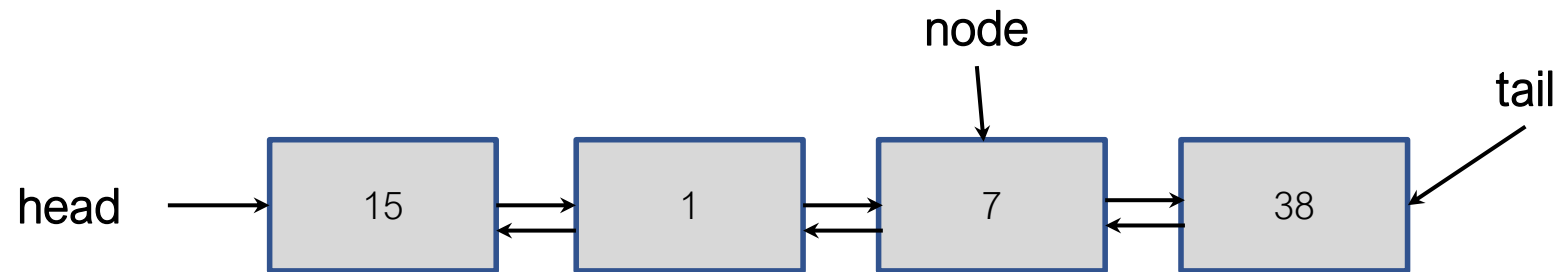


```
node = node.getNext();
```

# getNode(): Primeira versão

position = 2

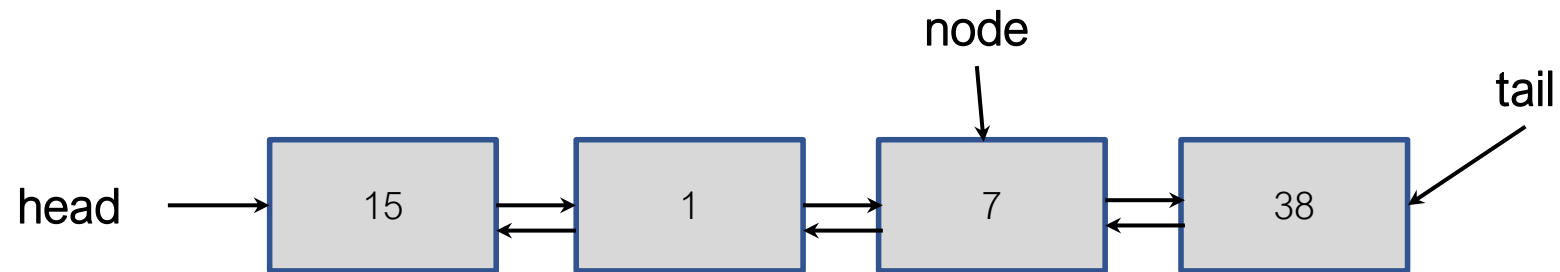
i = 2



# getNode(): Primeira versão

position = 2

i = 2



Fim do ciclo!

# Classe Lista Duplamente Ligada (4): getNode: Primeira Versão

```
// Returns the node at the specified position in the list.  
// Requires: position ranges from 0 to currentSize - 1.  
protected DoubleListNode<E> getNode( int position ){  
  
    DoubleListNode<E> node = head;  
  
    for ( int i = 0; i < position; i++ )  
        node = node.getNext();  
    return node;  
}
```

# Classe Lista Duplamente Ligada (4): getNode: Primeira Versão

```
// Returns the node at the specified position in the list.  
// Requires: position ranges from 0 to currentSize - 1.  
protected DoubleListNode<E> getNode( int position ){  
  
    DoubleListNode<E> node = head;  
  
    for ( int i = 0; i < position; i++ )  
        node = node.getNext();  
    return node;  
}
```

O que podemos fazer para melhorar esta implementação ?



# Classe Lista Duplamente Ligada (5)

```
// Returns the node at the specified position in the list.  
// Requires: position ranges from 0 to currentSize - 1.  
protected DoubleListNode<E> getNode( int position ){
```

```
    DoubleListNode<E> node = head;
```

```
    if ( position <= ( currentSize - 1 ) / 2 ){  
        node = head;  
        for ( int i = 0; i < position; i++ )  
            node = node.getNext();  
    }
```

```
    else{  
        node = tail;  
        for ( int i = currentSize - 1; i > position; i-- )  
            node = node.getPrevious();  
    }
```

```
    return node;
```

# Classe Lista Duplamente Ligada (6)

```
// Returns the element at the specified position in the list.  
// Range of valid positions: 0, ..., size() - 1.  
// If the specified position is 0, get corresponds to getFirst.  
// If the specified position is size()-1, get corresponds to  
//getLast.  
public E get( int position ) throws InvalidPositionException{  
    if ( position < 0 || position >= currentSize )  
        throw new InvalidPositionException();  
    // acesso ao elemento da lista em position.  
}
```

Como fazer ?

# Classe Lista Duplamente Ligada (7)

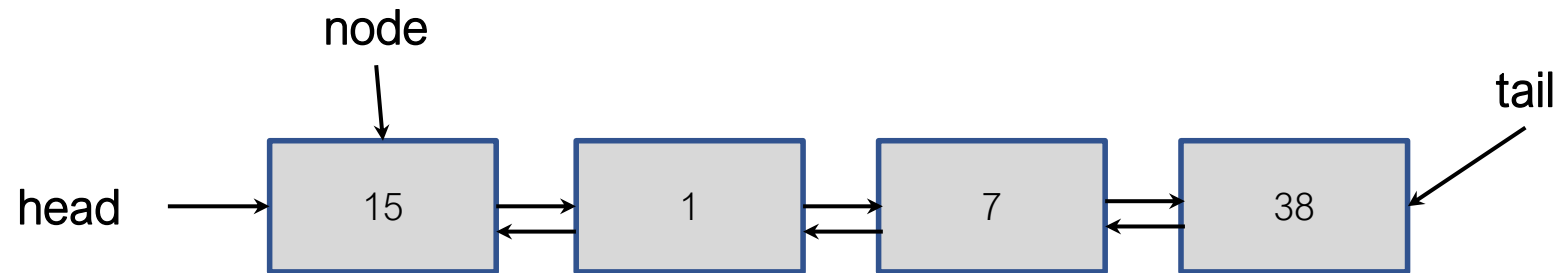
```
// Returns the position of the first occurrence of the specified
// element in the list, if the list contains the element.
// Otherwise, returns -1.
public int find( E element ){
    DoubleListNode<E> node = head;
    int position = 0;
    while ( node != null && !node.getElement().equals(element) ){
        node = node.getNext();
        position++;
    }
    if ( node == null )
        return -1;
    else
        return position;
}
```

↳ obriga a percorrer a lista toda para concluir que não há o node

# find

```
element = 38    node !=null
```

```
position = 0    !node.getElement().equals(element)
```

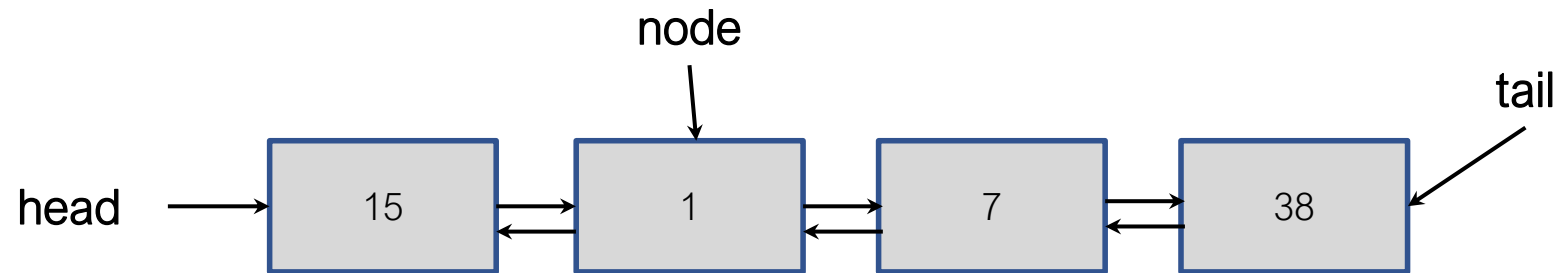


```
node = node.getNext();  
position++;
```

# find

```
element = 38    node !=null
```

```
position = 1    !node.getElement().equals(element)
```

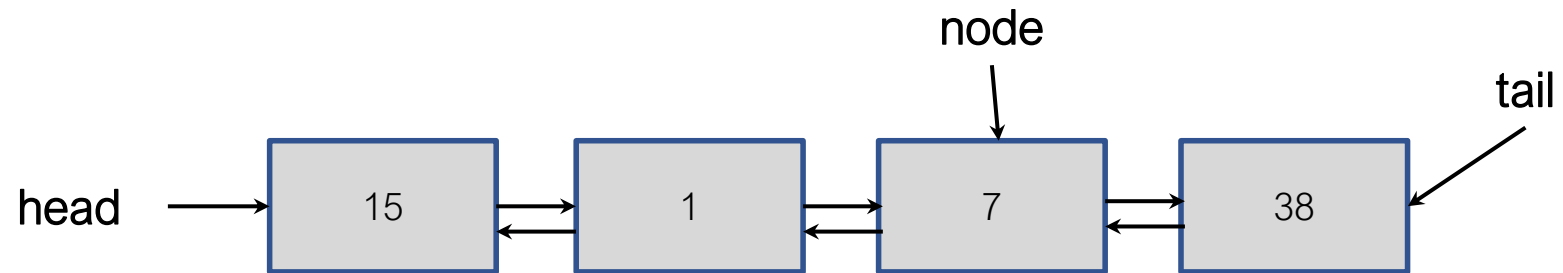


```
node = node.getNext();  
position++;
```

# find

```
element = 38    node !=null
```

```
position = 2    !node.getElement().equals(element)
```

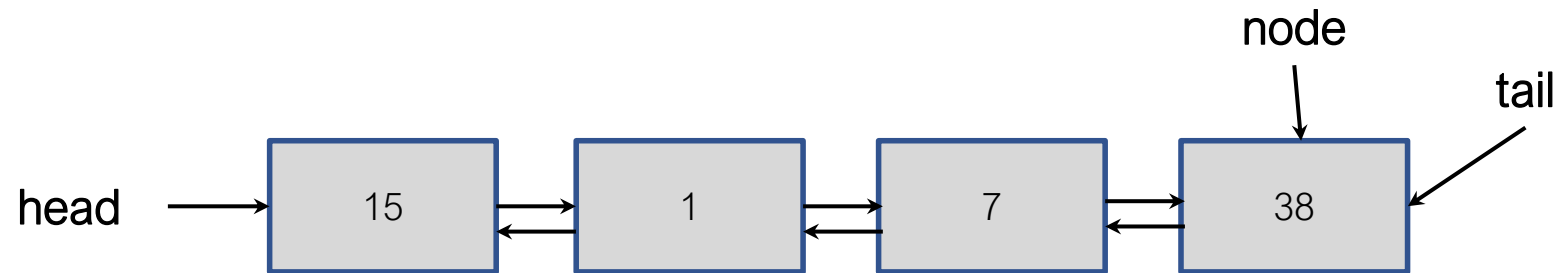


```
node = node.getNext();  
position++;
```

# find

```
element = 38    node !=null
```

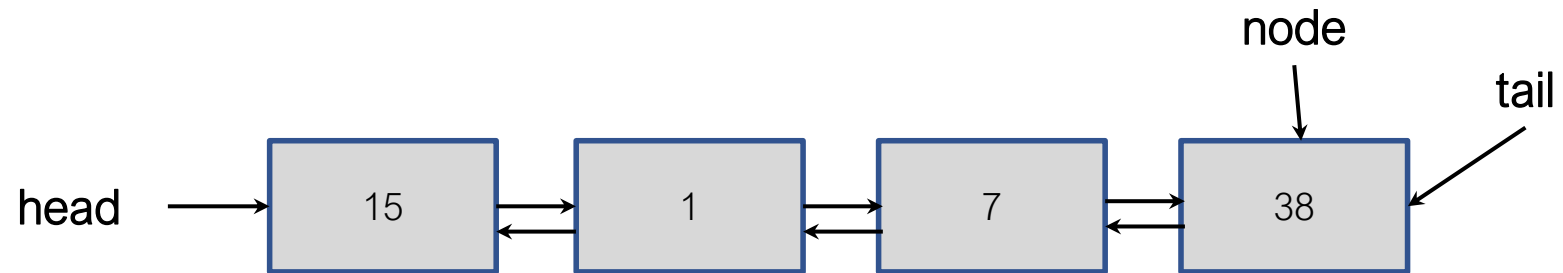
```
position = 3    !node.getElement().equals(element)
```



# find

element = 38      node != null

position = 3      !node.getElement().equals(element)



**Fim do ciclo!**

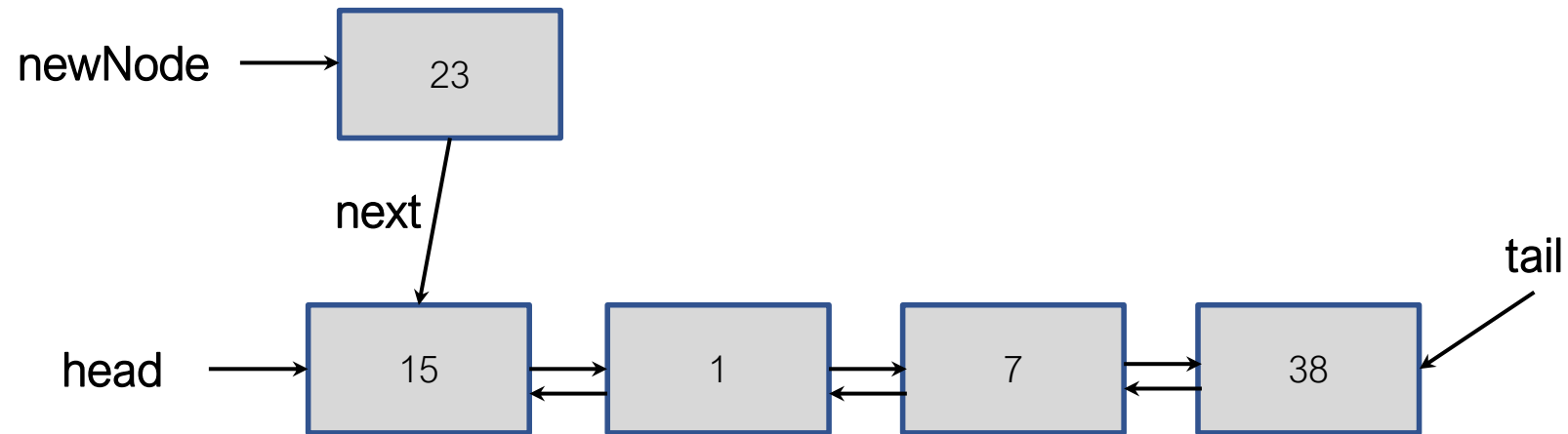


# Classe Lista Duplamente Ligada (8)

```
// Inserts the specified element at the first position in the
// list.
public void addFirst( E element ){
    DoubleListNode<E> newNode = new DoubleListNode<E>(element, null, head);
    if ( this.isEmpty() )
        tail = newNode;
    else
        head.setPrevious(newNode);
    head = newNode;
    currentSize++;
}
```

# addFirst

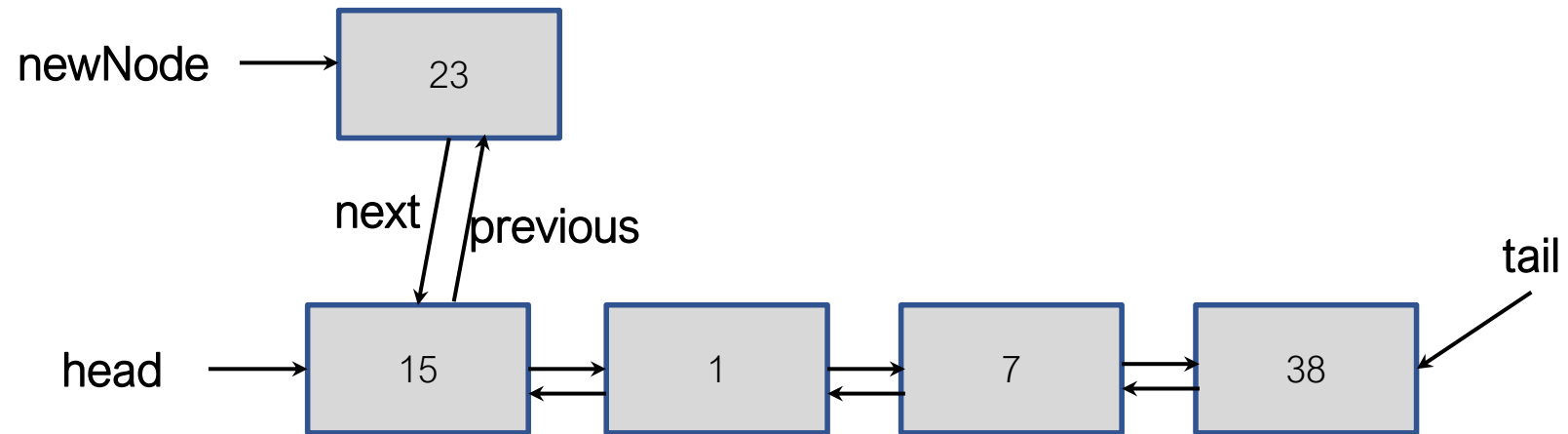
currentSize = 4



```
DoubleListNode<E> newNode = new DoubleListNode<E>(element, null,  
head);
```

# addFirst

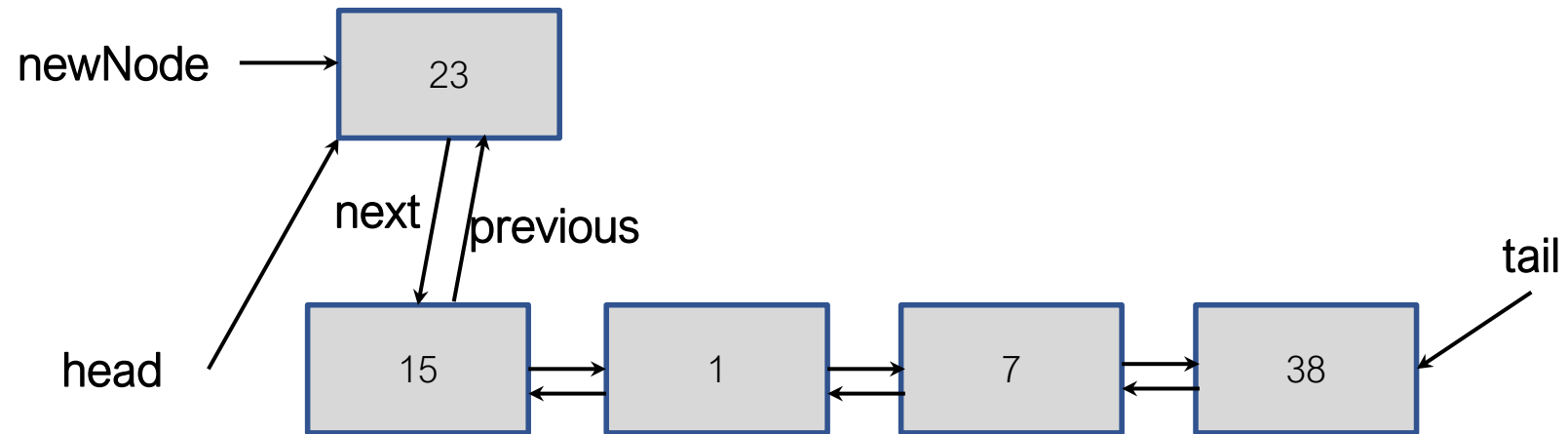
currentSize = 4



```
head.setPrevious(newNode);
```

# addFirst

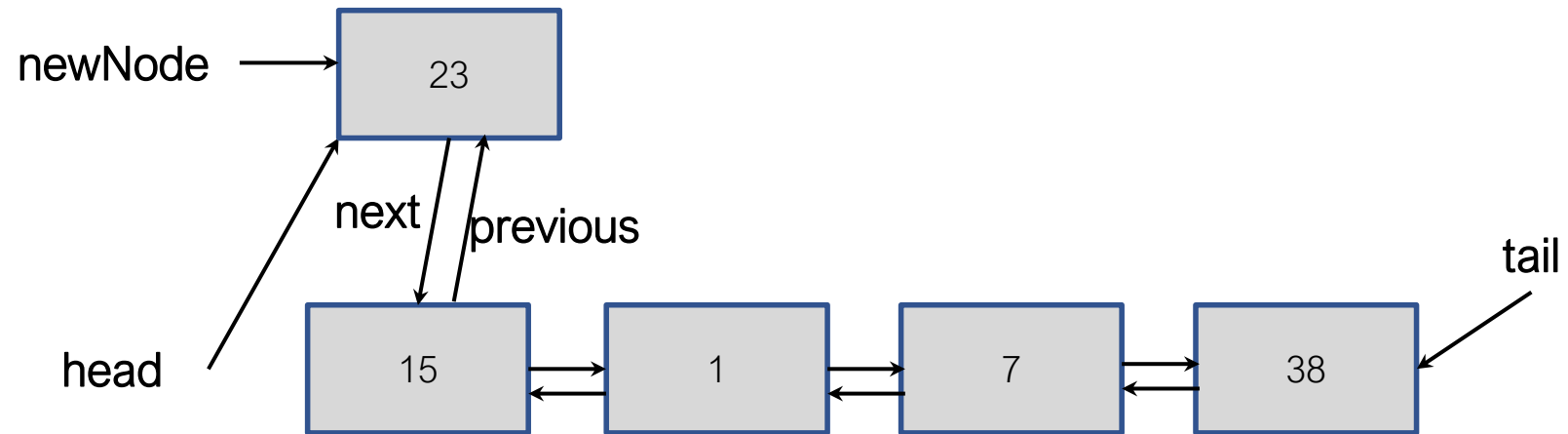
currentSize = 4



head = newNode;

# addFirst

currentSize = 5



currentSize++;

# Classe Lista Duplamente Ligada (9)

```
// Inserts the specified element at the last position in the
// list.
public void addLast( E element ){
    //TODO: Left as an exercise.
    new Node = (element)
    if (this.isEmpty())
        head = new Node,
    else
        tail.setNext(new Node),
    tail = new Node,
    currentSize++,
}
```

# Classe Lista Duplamente Ligada (10)

```
// Inserts the specified element at the specified position in
// the list.
// Range of valid positions: 0, ..., size().
// If the specified position is 0, add corresponds to addFirst.
// If the specified position is size(), add corresponds to
// addLast.
public void add( int position, E element )
                throws InvalidPositionException{
    if ( position < 0 || position > currentSize )
        throw new InvalidPositionException();
    if ( position == 0 )
        this.addFirst(element);
    else if ( position == currentSize )
        this.addLast(element);
    else
        this.addMiddle(position, element);
}
```

# Classe Lista Duplamente Ligada (10)

```
// Inserts the specified element at the specified position in
// the list.
// Range of valid positions: 0, ..., size().
// If the specified position is 0, add corresponds to addFirst.
// If the specified position is size(), add corresponds to
// addLast.
public void add( int position, E element )
                    throws InvalidPositionException{
    if ( position < 0 || position > currentSize )
        throw new InvalidPositionException();
    if ( position == 0 )
        this.addFirst(element);
    else if ( position == currentSize )
        this.addLast(element);
    else
        this.addMiddle(position, element);
}
```



# Classe Lista Duplamente Ligada (11)

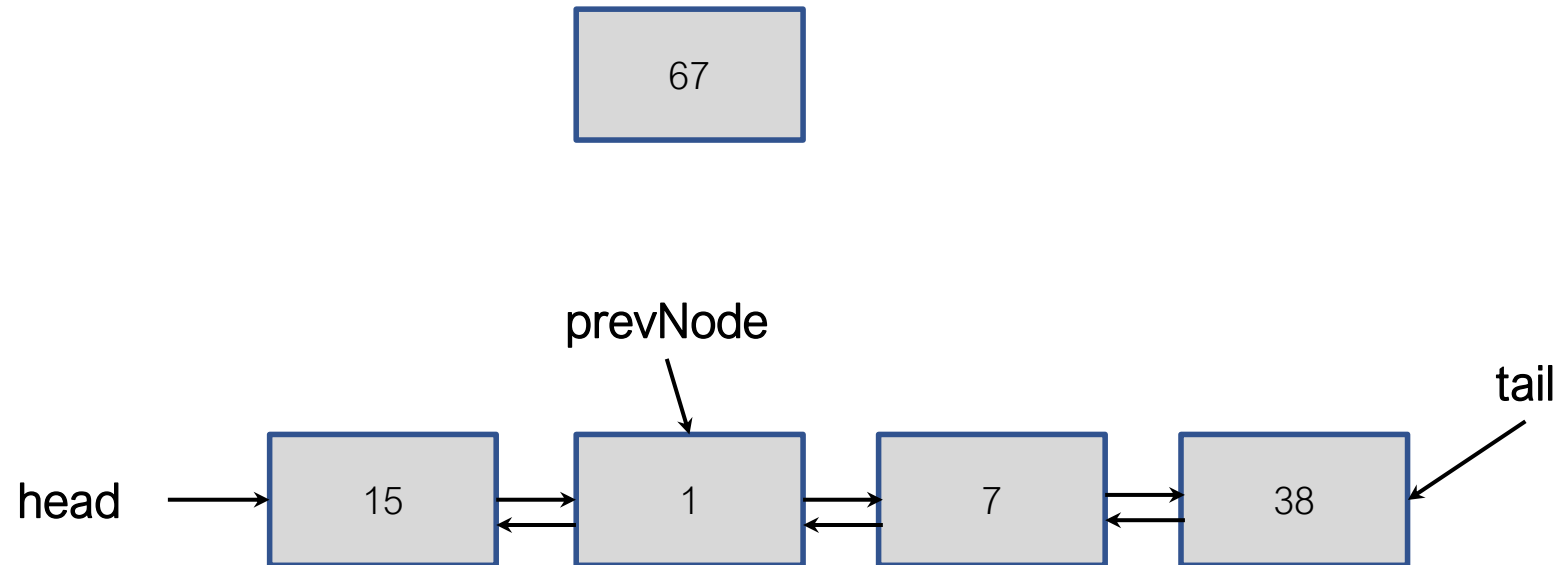
```
// Inserts the specified element at the specified position in
// the list.
// Requires: position ranges from 1 to currentSize - 1.
protected void addMiddle( int position, E element ){
    DoubleListNode<E> prevNode = this.getNode(position - 1);
    DoubleListNode<E> nextNode = prevNode.getNext();
    DoubleListNode<E> newNode =
        new DoubleListNode<E>(element, prevNode, nextNode);
    //TODO: Left as an exercise.
}
    prevNode.setNext(newNode),
    nextNode.setPrevious(newNode),
    currentSize++,
```

# addMiddle

element = 67

position = 2

currentSize = 4



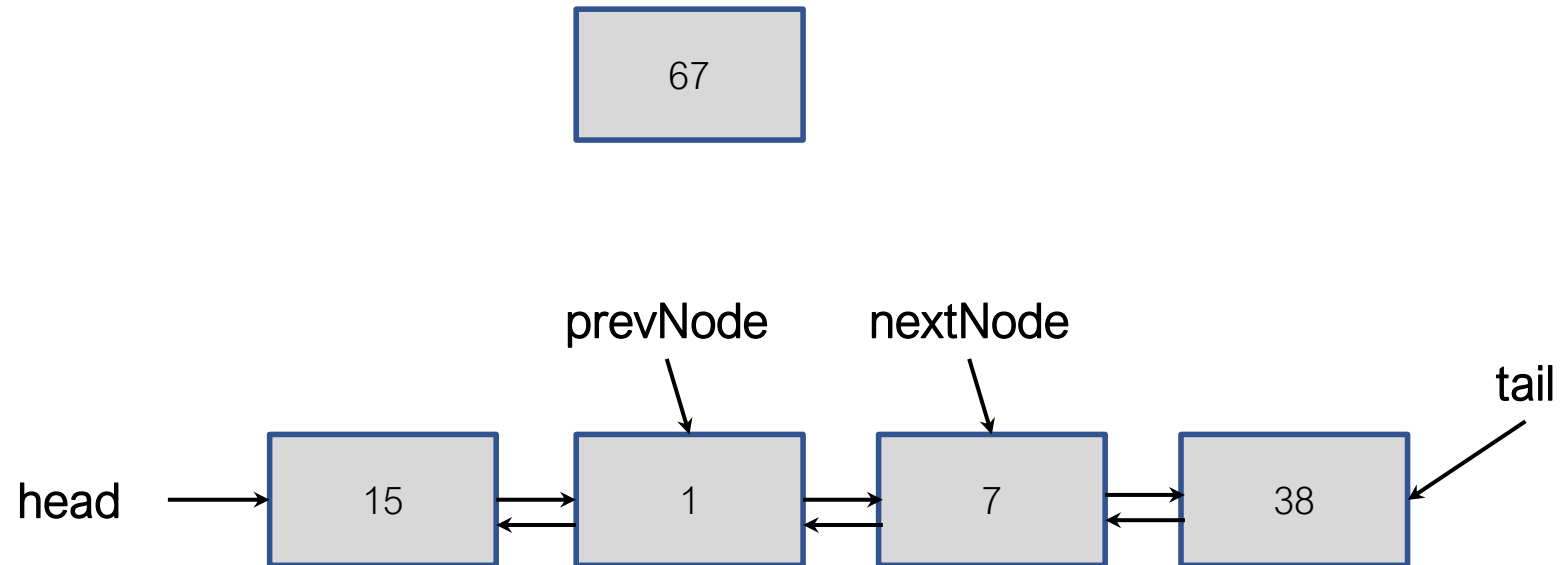
```
prevNode = this.getNode(position - 1);
```

# addMiddle

element = 67

position = 2

currentSize = 4



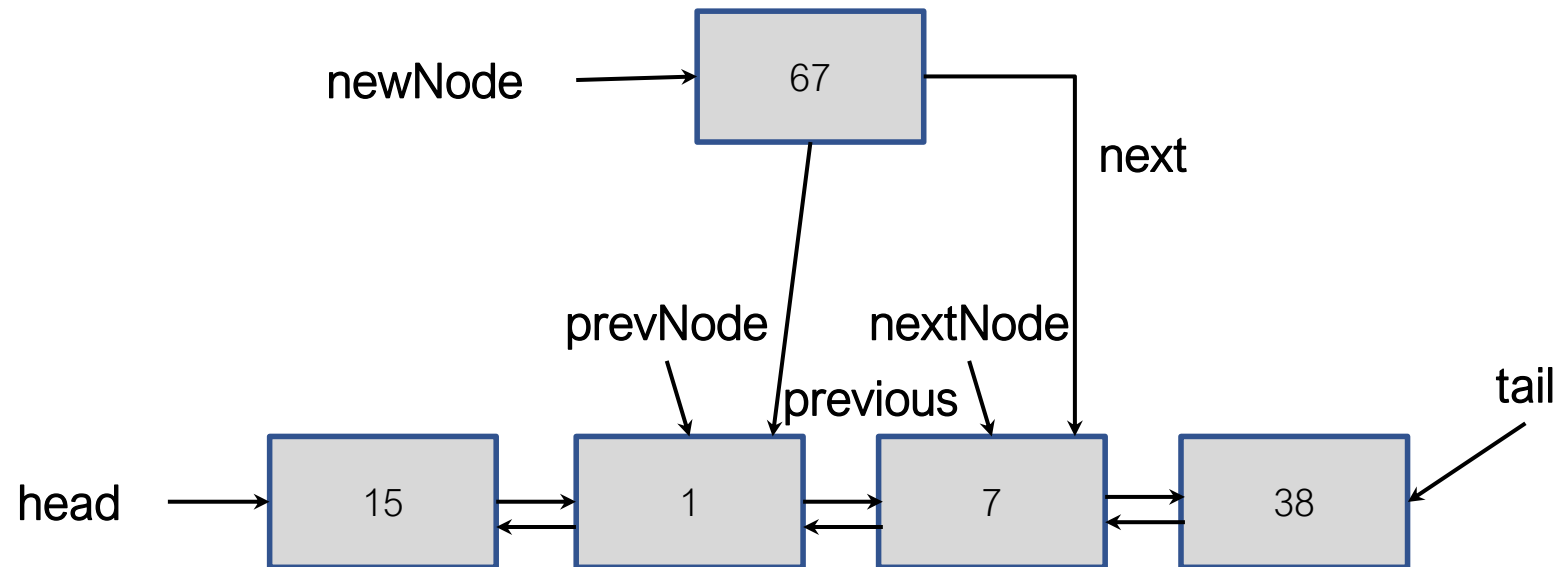
```
nextNode = prevNode.getNext();
```

# addMiddle

element = 67

position = 2

currentSize = 4



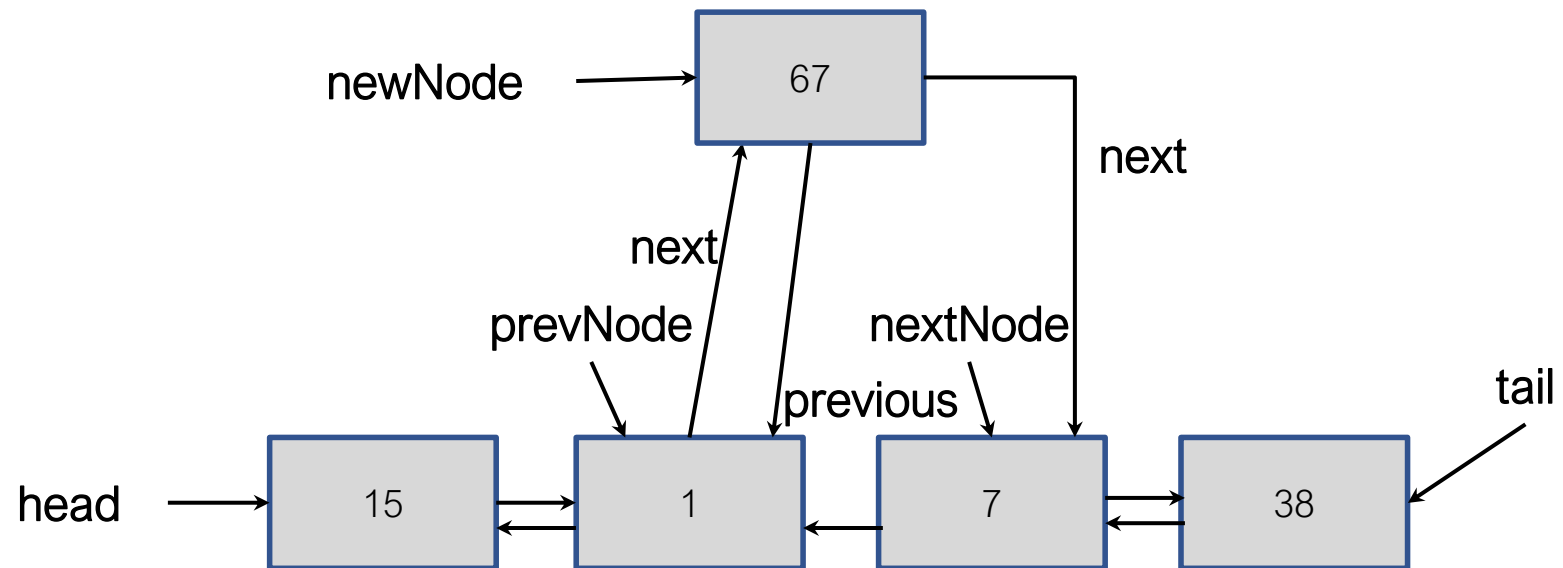
```
newNode = new DoubleListNode<E>(element, prevNode, nextNode);
```

# addMiddle

element = 67

position = 2

currentSize = 4

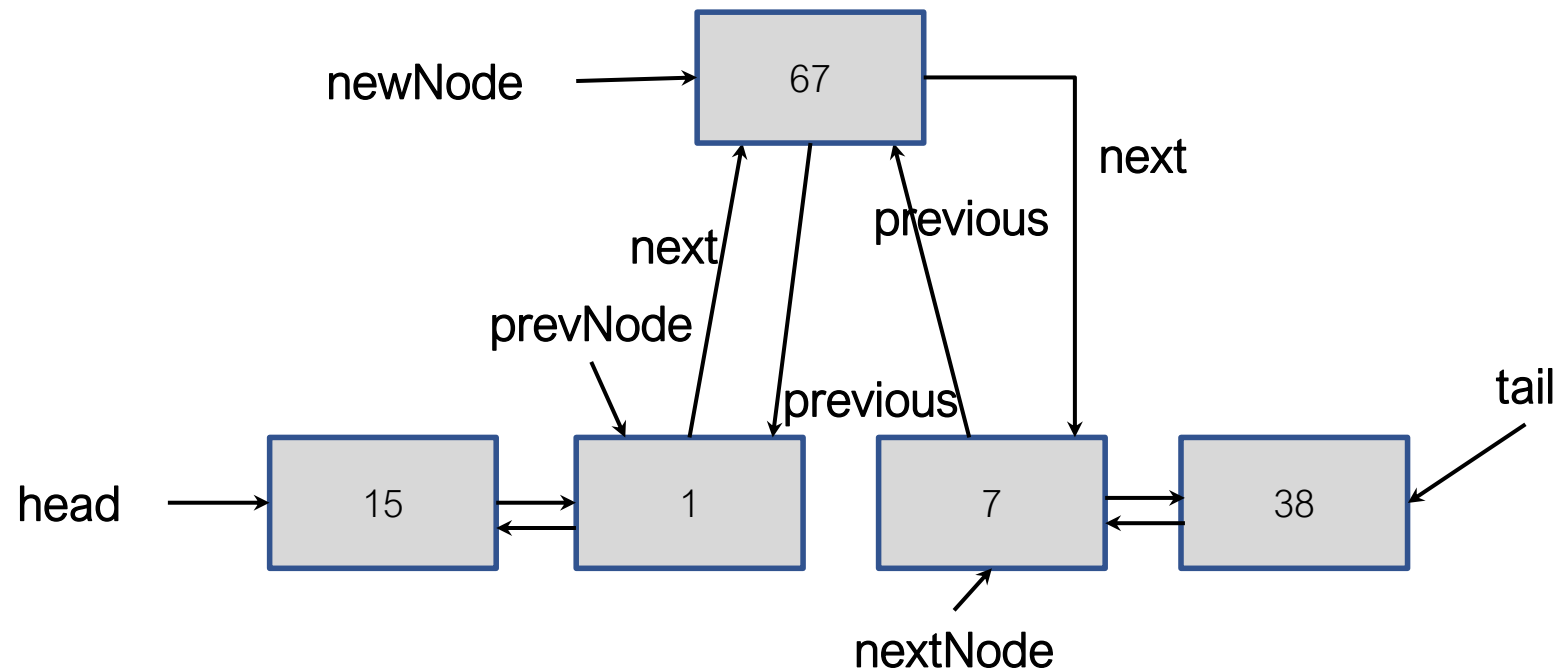


# addMiddle

element = 67

position = 2

currentSize = 4

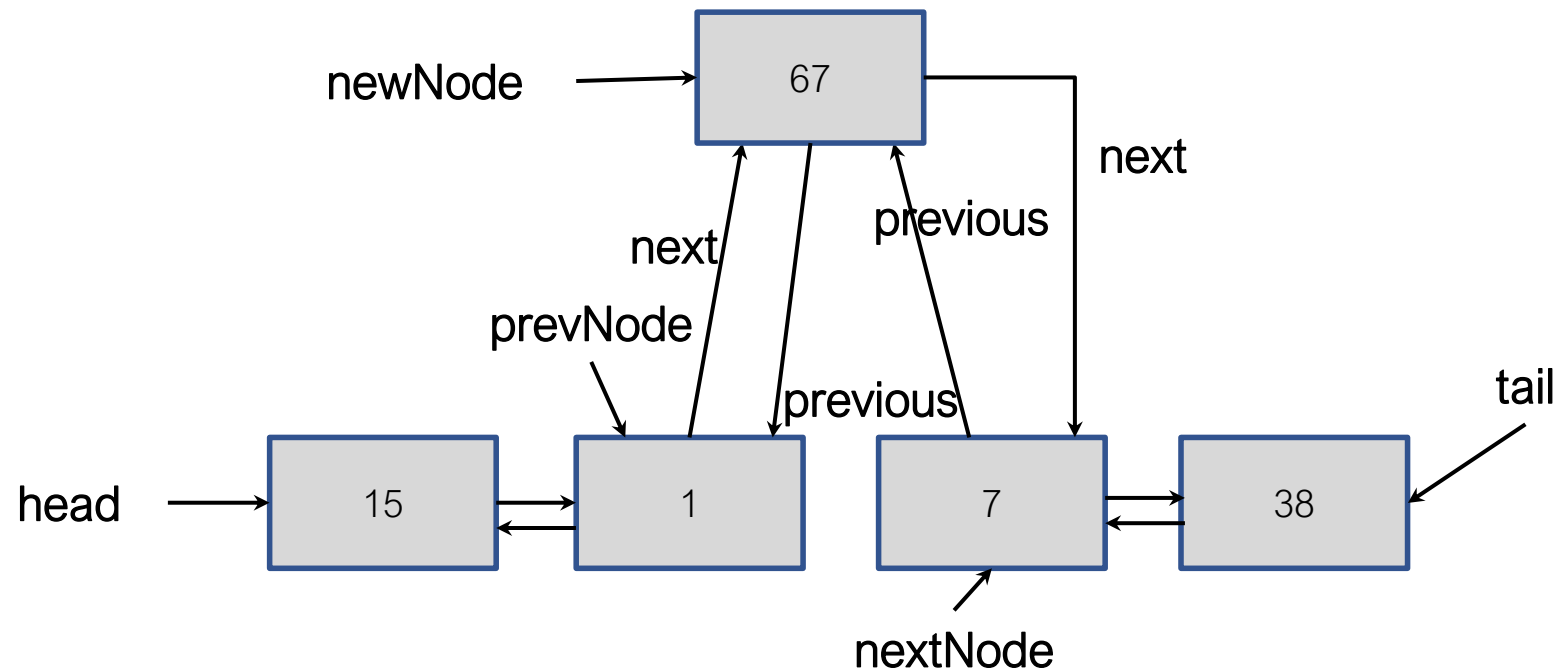


# addMiddle

element = 67

position = 2

currentSize = 5



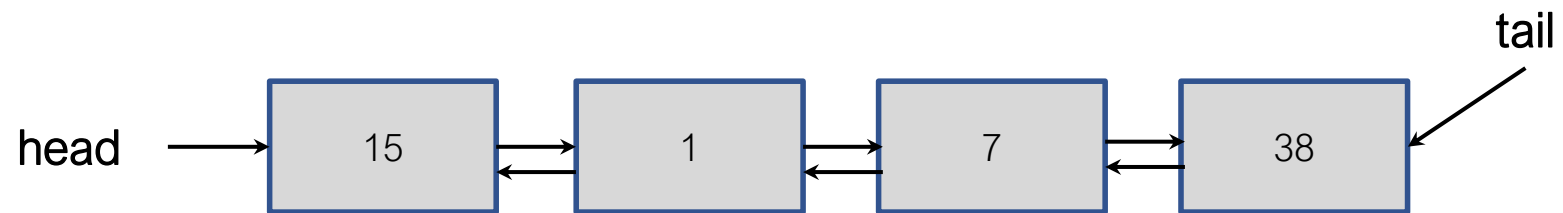
# Classe Lista Duplamente Ligada (12)

```
// Removes the first node in the list.  
// Requires: the list is not empty.  
protected void removeFirstNode( ){  
    head = head.getNext();  
    if ( head == null )  
        tail = null;  
    else  
        head.setPrevious(null);  
        currentSize--;  
}
```



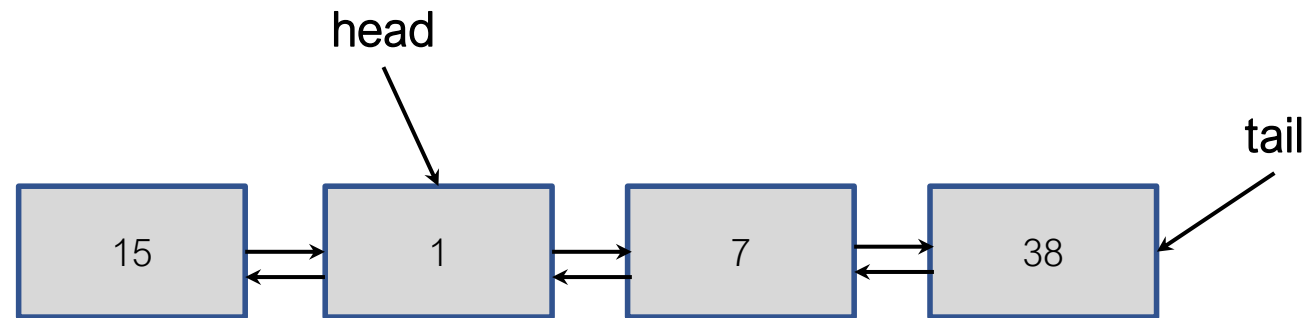
# removeFirstNode

currentSize = 4



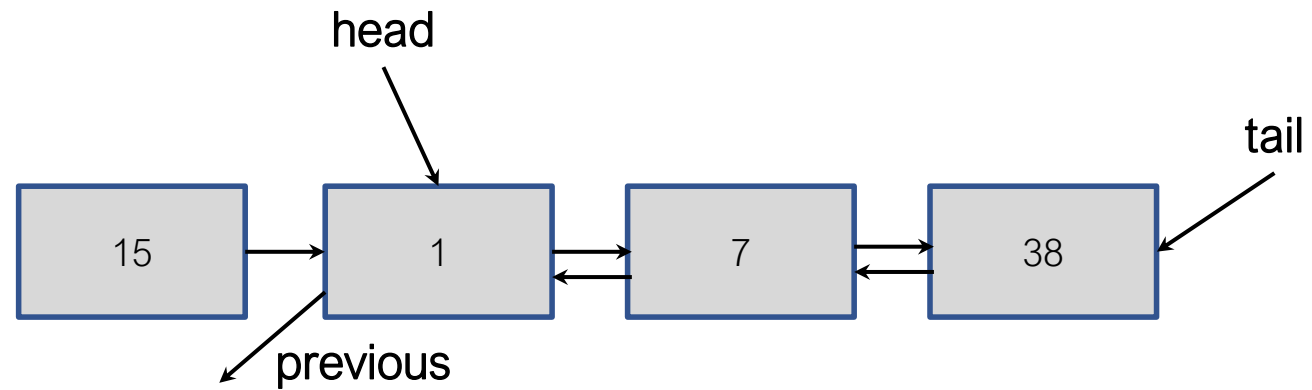
# removeFirstNode

currentSize = 4



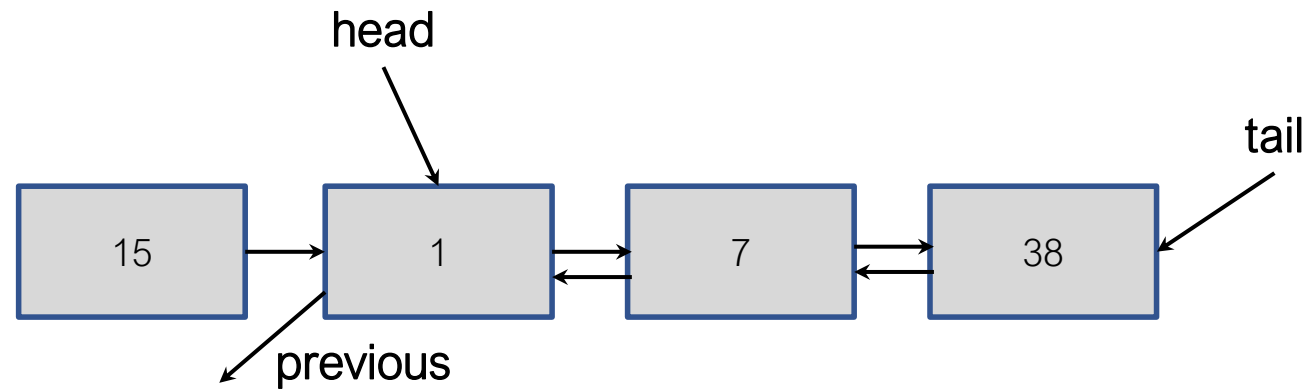
# removeFirstNode

currentSize = 4



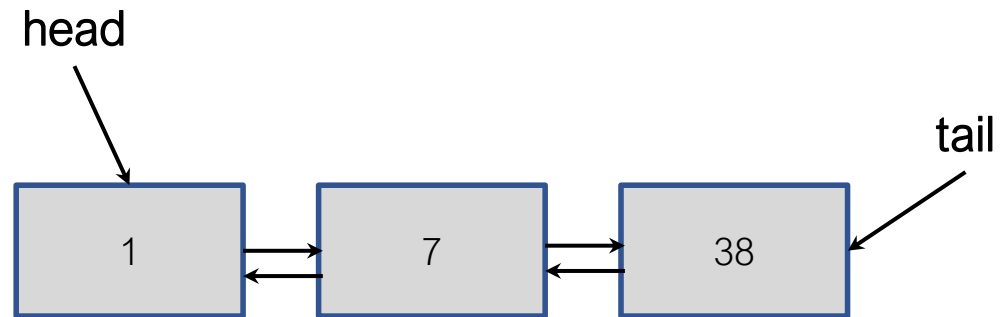
# removeFirstNode

currentSize = 3



# removeFirstNode

currentSize = 3



## Classe Lista Duplamente Ligada (13)

```
// Removes and returns the element at the first position
// in the list.
public E removeFirst( ) throws EmptyListException{
//TODO: Left as an exercise.
}
    if (this.isEmpty())
        throw new EmptyListException();

    E element = head.getElement();
    this.removeFirstElement();
    return element;
```

# Classe Lista Duplamente Ligada (14)

```
// Removes the last node in the list.  
// Requires: the list is not empty.  
protected void removeLastNode( ){  
    //TODO: Left as an exercise.  
}  
    tail = tail.getPrevious()  
    if (tail == null)  
        head = null,  
    else  
        tail.setNext(),  
    currentSize--;
```

## Classe Lista Duplamente Ligada (15)

```
// Removes and returns the element at the last position
// in the list.
public E removeLast( ) throws EmptyListException{
    if ( this.isEmpty() )
        throw new EmptyListException();
    E element = tail.getElement();
    this.removeLastNode();
    return element;
}
```



# Classe Lista Duplamente Ligada (16)

```
// Removes the specified node from the list.
// Requires: the node is neither the head nor the tail of
// the list.
protected void removeMiddleNode( DoubleListNode<E> node ){
```

```
    //TODO: Left as an exercise.
```

```
}
```

```
    E prev = node getPrevious(),
    E next = node getNext(),
```

```
    prev setNext(next),
    next setPrevious(prev),
```

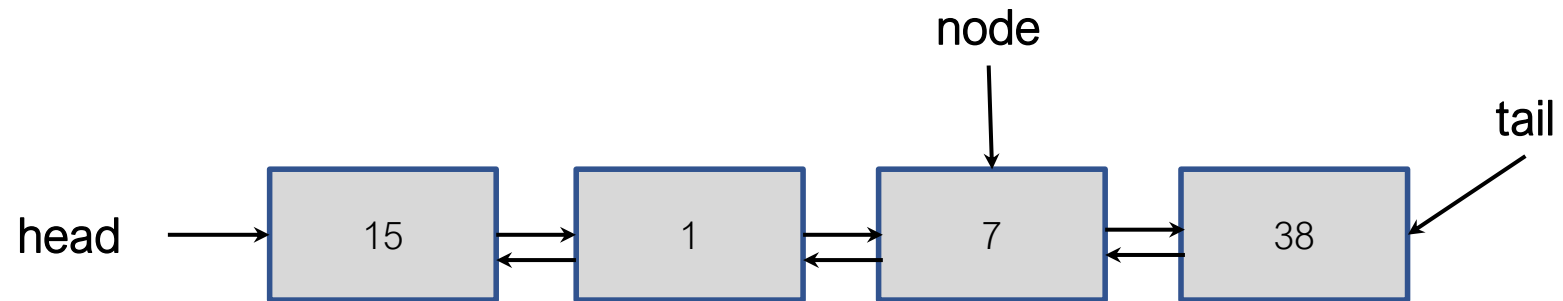
// dúvida se é necessário alterar o previous e o next no elemento removido

```
    currentSize--;
```

↓  
acho que não  
pois não será  
possível acessar-lhe

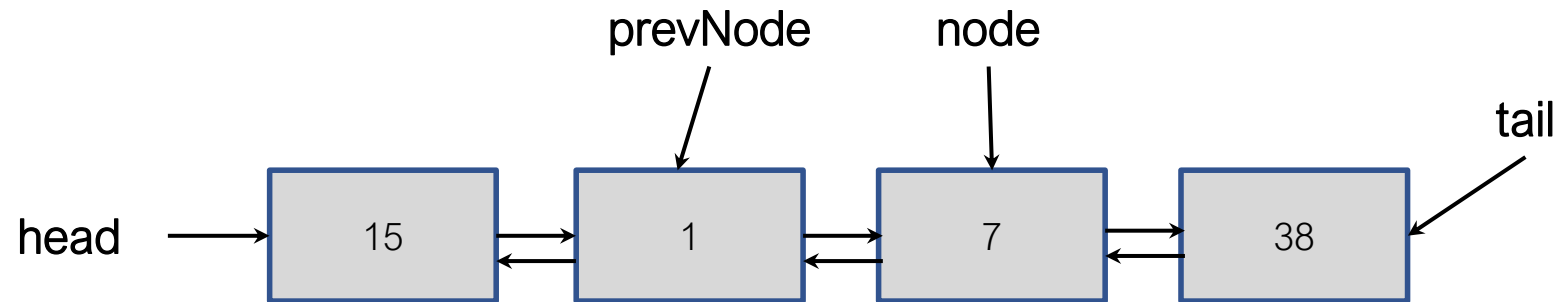
# removeMiddleNode

currentSize = 4



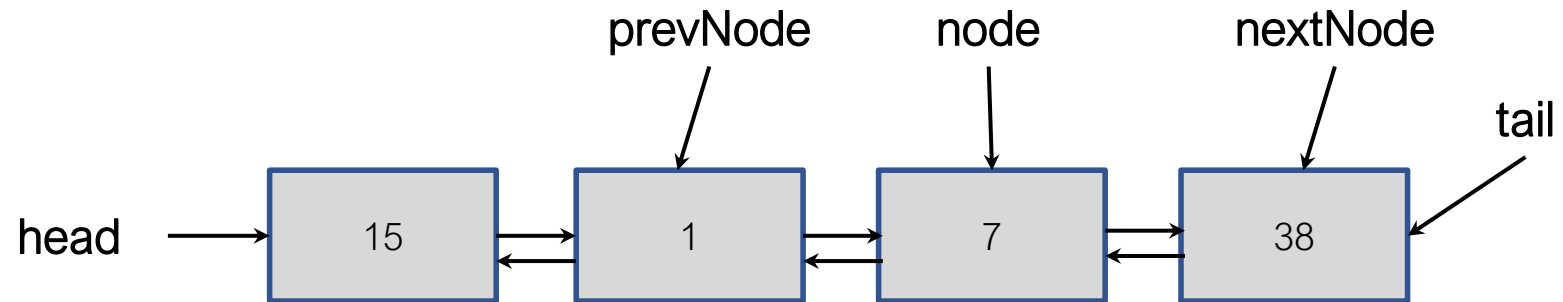
# removeMiddleNode

currentSize = 4



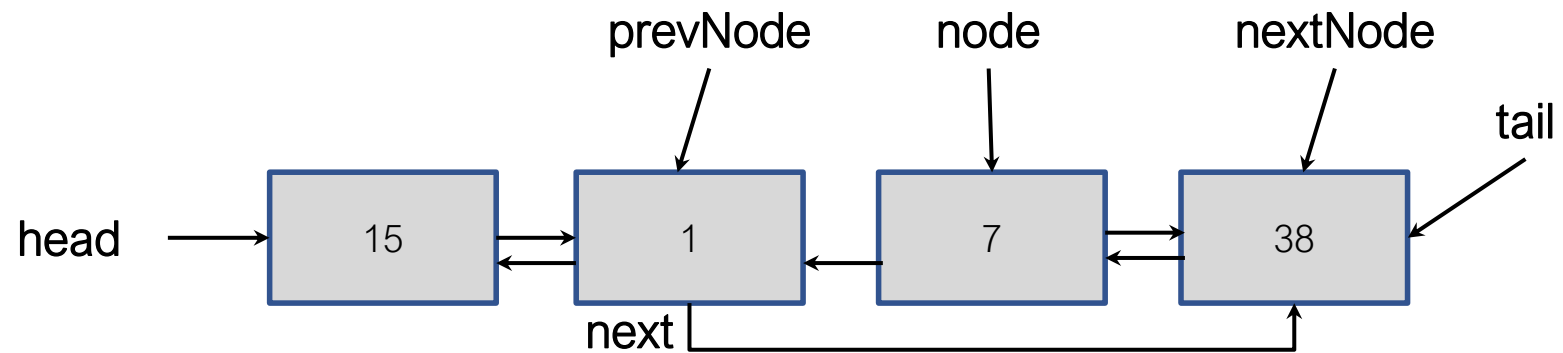
# removeMiddleNode

currentSize = 4



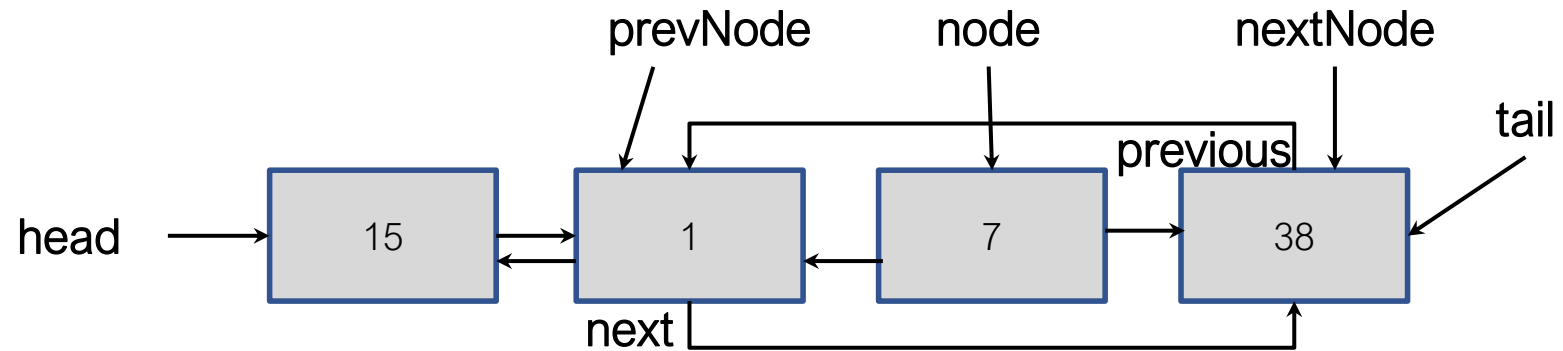
# removeMiddleNode

currentSize = 4



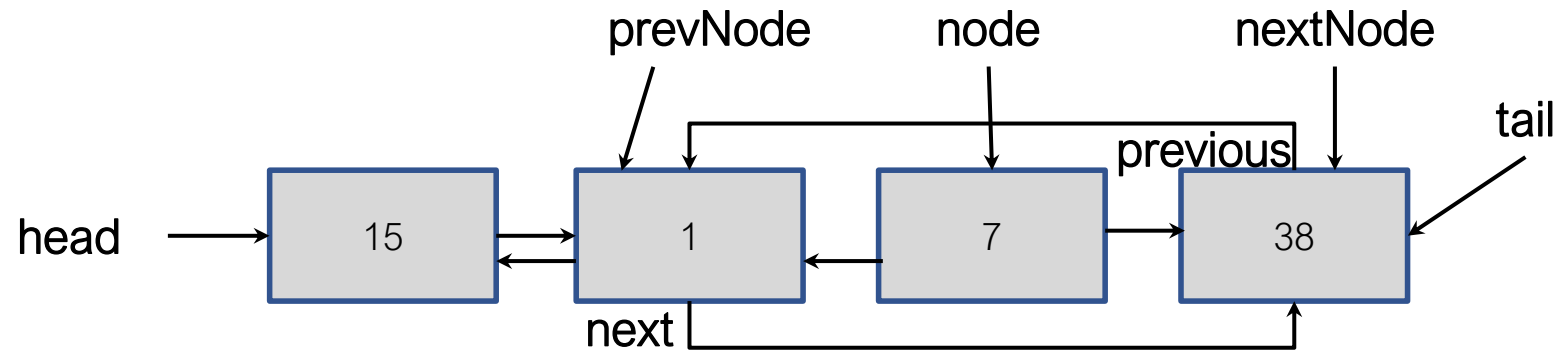
# removeMiddleNode

currentSize = 4



# removeMiddleNode

currentSize = 3



# Classe Lista Duplamente Ligada (17)

```
// Removes and returns the element at the specified position in
// the list.
// Range of valid positions: 0, ..., size()-1.
// If the specified position is 0, remove corresponds to
// removeFirst.
// If the specified position is size()-1, remove corresponds to
// removeLast.
public E remove( int position ) throws InvalidPositionException{
    if ( position < 0 || position >= currentSize )
        throw new InvalidPositionException();
    if ( position == 0 )
        return this.removeFirst();
    else if ( position == currentSize - 1 )
        return this.removeLast();
    else {
        //TODO: Left as an exercise.
    }
}
```



# Classe Lista Duplamente Ligada (18)

```
// Returns the node with the first occurrence of the specified  
// element in the list, if the list contains the element.  
// Otherwise, returns null.  
protected DoubleListNode<E> findNode( E element ){  
  
    //TODO: Left as an exercise.  
}
```

Parecido com o find...

# Pode vir a ser preciso ...

```
// finds the first element in the list equal to the one
// in the parameter.
// Otherwise, returns null.
public E findEquals( E element ){

    // Left as an exercise ?

    //If needed, the methods should be added to a new interface
    //of a searchable List which extends the current one
}
```

Pensem nisto...

# Classe Lista Duplamente Ligada (19)

```
// Removes the first occurrence of the specified element from the
// list and returns true, if the list contains the element.
// Otherwise, returns false.
public boolean remove( E element ){
    DoubleListNode<E> node = this.findNode(element);
    if ( node == null )
        return false;
    else {
        if ( node == head )
            this.removeFirstNode();
        else if ( node == tail )
            this.removeLastNode();
        else
            this.removeMiddleNode(node);
        return true;
    }
}
```

## Classe Lista Duplamente Ligada (20)

```
// Returns an iterator of the elements in the list
// (in proper sequence).
public Iterator<E> iterator( ){
    return new DoubleListIterator<E>(head, tail);
}

// Removes all of the elements from the specified list and
// inserts them at the end of the list (in proper sequence).
public void append( DoubleList<E> list ){
    //TODO: Left as an exercise.
}
} // End of DoubleList.
```

# Exemplo de iteração



rewind()	
next()	
next()	
previous()	
fullForward()	
previous()	
next()	

Como evolui o iterador ?

# Exemplo de iteração



rewind()	
next()	9
next()	2
previous()	9
fullForward()	
previous()	7
next()	exception

Como evolui o iterador ?