

# Aula 07

## Estruturas de dados recursivas

### *Listas ligadas*

*Programação II, 2019-2020*

*v1.4, 29-03-2020*

Lista Ligada

Polimorfismo  
Paramétrico

Processamento  
recursivo de listas

**1** Lista Ligada

**2** Polimorfismo Paramétrico

**3** Processamento recursivo de listas

Lista Ligada

Polimorfismo  
Paramétrico

Processamento  
recursivo de listas

**1** Lista Ligada

**2** Polimorfismo Paramétrico

**3** Processamento recursivo de listas

# Como guardar colecções de dados?

- Temos utilizado vectores (*arrays*).
- Permitem guardar elementos preservando a sua ordem.
- Permitem **acesso aleatório**, i.e., acesso direto rápido a qualquer elemento, por qualquer ordem.
- No entanto, os **vectores têm limitações**:
  - A sua capacidade tem de ser fixada quando são criados, isto obriga a realocar/eliminar um vector quando o número de elementos não é conhecido à partida.
  - Ou então, realocar/eliminar o vector quando chegam novos elementos, com custos em tempo de processamento.
  - Iniciar ou remover elementos numa posição intermédia pode demandar bastante tempo se for necessário deslocar muitos elementos.

- Temos utilizado vectores (**arrays**).
- Permitem guardar elementos preservando a sua ordem.
- Permitem **acesso aleatório**, i.e., acesso direto rápido a qualquer elemento, por qualquer ordem.
- No entanto, os **vectores têm limitações**:
  - A sua capacidade tem de ser fixada quando são criados.
  - Isto obriga a sobredimensionar um vector quando o número de elementos não é conhecido à partida.
  - Ou então, redimensionar o vector quando chegam novos elementos, com custos em tempo de processamento.
  - Inserir ou remover elementos numa posição intermédia pode demorar bastante tempo se for necessário deslocar muitos elementos.

- Temos utilizado vectores (**arrays**).
- Permitem guardar elementos preservando a sua ordem.
- Permitem **acesso aleatório**, i.e., acesso direto rápido a qualquer elemento, por qualquer ordem.
- No entanto, os **vectores têm limitações**:
  - A sua capacidade tem de ser fixada quando são criados.
  - Isto obriga a sobredimensionar um vector quando o número de elementos não é conhecido à partida.
  - Ou então, redimensionar o vector quando chegam novos elementos, com custos em tempo de processamento.
  - Inserir ou remover elementos numa posição intermédia pode demorar bastante tempo se for necessário deslocar muitos elementos.

- Temos utilizado vectores (**arrays**).
- Permitem guardar elementos preservando a sua ordem.
- Permitem **acesso aleatório**, i.e., acesso direto rápido a qualquer elemento, por qualquer ordem.
- No entanto, os **vectores têm limitações**:
  - A sua capacidade tem de ser fixada quando são criados.
  - Isto obriga a sobredimensionar um vector quando o número de elementos não é conhecido à partida.
  - Ou então, redimensionar o vector quando chegam novos elementos, com custos em tempo de processamento.
  - Inserir ou remover elementos numa posição intermédia pode demorar bastante tempo se for necessário deslocar muitos elementos.

- Temos utilizado vectores (**arrays**).
- Permitem guardar elementos preservando a sua ordem.
- Permitem **acesso aleatório**, i.e., acesso direto rápido a qualquer elemento, por qualquer ordem.
- No entanto, os **vectores têm limitações**:
  - A sua capacidade tem de ser fixada quando são criados.
  - Isto obriga a sobredimensionar um vector quando o número de elementos não é conhecido à partida.
  - Ou então, redimensionar o vector quando chegam novos elementos, com custos em tempo de processamento.
  - Inserir ou remover elementos numa posição intermédia pode demorar bastante tempo se for necessário deslocar muitos elementos.



- Temos utilizado vectores (***arrays***).
- Permitem guardar elementos preservando a sua ordem.
- Permitem **acesso aleatório**, i.e., acesso direto rápido a qualquer elemento, por qualquer ordem.
- No entanto, os **vectores têm limitações**:
  - A sua capacidade tem de ser fixada quando são criados.
  - Isto obriga a sobredimensionar um vector quando o número de elementos não é conhecido à partida.
  - Ou então, redimensionar o vector quando chegam novos elementos, com custos em tempo de processamento.
  - Inserir ou remover elementos numa posição intermédia pode demorar bastante tempo se for necessário deslocar muitos elementos.

- Temos utilizado vectores (***arrays***).
- Permitem guardar elementos preservando a sua ordem.
- Permitem **acesso aleatório**, i.e., acesso direto rápido a qualquer elemento, por qualquer ordem.
- No entanto, os **vectores têm limitações**:
  - A sua capacidade tem de ser fixada quando são criados.
  - Isto obriga a sobredimensionar um vector quando o número de elementos não é conhecido à partida.
  - Ou então, redimensionar o vector quando chegam novos elementos, com custos em tempo de processamento.
  - Inserir ou remover elementos numa posição intermédia pode demorar bastante tempo se for necessário deslocar muitos elementos.

- Temos utilizado vectores (***arrays***).
- Permitem guardar elementos preservando a sua ordem.
- Permitem **acesso aleatório**, i.e., acesso direto rápido a qualquer elemento, por qualquer ordem.
- No entanto, os **vectores têm limitações**:
  - A sua capacidade tem de ser fixada quando são criados.
  - Isto obriga a sobredimensionar um vector quando o número de elementos não é conhecido à partida.
  - Ou então, redimensionar o vector quando chegam novos elementos, com custos em tempo de processamento.
  - Inserir ou remover elementos numa posição intermédia pode demorar bastante tempo se for necessário deslocar muitos elementos.

- Temos utilizado vectores (***arrays***).
- Permitem guardar elementos preservando a sua ordem.
- Permitem **acesso aleatório**, i.e., acesso direto rápido a qualquer elemento, por qualquer ordem.
- No entanto, os **vectores têm limitações**:
  - A sua capacidade tem de ser fixada quando são criados.
  - Isto obriga a sobredimensionar um vector quando o número de elementos não é conhecido à partida.
  - Ou então, redimensionar o vector quando chegam novos elementos, com custos em tempo de processamento.
  - Inserir ou remover elementos numa posição intermédia pode demorar bastante tempo se for necessário deslocar muitos elementos.

- Estrutura de dados sequencial em que cada elemento da lista contém uma referência para o próximo elemento.
  - No último elemento, a referência é `null`.
- Ao contrário do vector, é completamente **dinâmica**.
- No entanto, obriga a um **acesso sequencial**.
- Recorre a uma estrutura auxiliar (um *nó*) para armazenar cada elemento.
- O nó é uma estrutura de dados **recursiva**, dado que a sua definição contém uma referência para si própria.

- Estrutura de dados sequencial em que cada elemento da lista contém uma referência para o próximo elemento.
  - No último elemento, a referência é `null`.
- Ao contrário do vector, é completamente **dinâmica**.
- No entanto, obriga a um **acesso sequencial**.
- Recorre a uma estrutura auxiliar (um *nó*) para armazenar cada elemento.
- O nó é uma estrutura de dados **recursiva**, dado que a sua definição contém uma referência para si própria.

- Estrutura de dados sequencial em que cada elemento da lista contém uma referência para o próximo elemento.
  - No último elemento, a referência é `null`.
- Ao contrário do vector, é completamente **dinâmica**.
- No entanto, obriga a um **acesso sequencial**.
- Recorre a uma estrutura auxiliar (um *nó*) para armazenar cada elemento.
- O nó é uma estrutura de dados **recursiva**, dado que a sua definição contém uma referência para si própria.

- Estrutura de dados sequencial em que cada elemento da lista contém uma referência para o próximo elemento.
  - No último elemento, a referência é `null`.
- Ao contrário do vector, é completamente **dinâmica**.
- No entanto, obriga a um **acesso sequencial**.
- Recorre a uma estrutura auxiliar (um *nó*) para armazenar cada elemento.
- O nó é uma estrutura de dados **recursiva**, dado que a sua definição contém uma referência para si própria.



- Estrutura de dados sequencial em que cada elemento da lista contém uma referência para o próximo elemento.
  - No último elemento, a referência é `null`.
- Ao contrário do vector, é completamente **dinâmica**.
- No entanto, obriga a um **acesso sequencial**.
- Recorre a uma estrutura auxiliar (um *nó*) para armazenar cada elemento.
- O nó é uma estrutura de dados **recursiva**, dado que a sua definição contém uma referência para si própria.

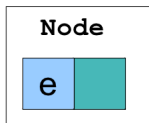
- Estrutura de dados sequencial em que cada elemento da lista contém uma referência para o próximo elemento.
  - No último elemento, a referência é `null`.
- Ao contrário do vector, é completamente **dinâmica**.
- No entanto, obriga a um **acesso sequencial**.
- Recorre a uma estrutura auxiliar (um *nó*) para armazenar cada elemento.
- O nó é uma estrutura de dados **recursiva**, dado que a sua definição contém uma referência para si própria.

- Estrutura de dados sequencial em que cada elemento da lista contém uma referência para o próximo elemento.
  - No último elemento, a referência é `null`.
- Ao contrário do vector, é completamente **dinâmica**.
- No entanto, obriga a um **acesso sequencial**.
- Recorre a uma estrutura auxiliar (um *nó*) para armazenar cada elemento.
- O nó é uma estrutura de dados **recursiva**, dado que a sua definição contém uma referência para si própria.

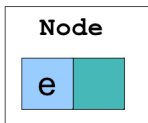
# Lista ligada simples: exemplo



# Lista ligada simples: exemplo



# Lista ligada simples: exemplo

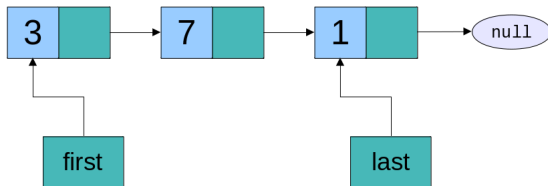


```
class Node
{
    int e;
    Node next;
}
```



- Exemplo: lista com os elementos 3, 7 e 1.
- A lista possui acesso direto ao primeiro e último elementos.
- É fácil acrescentar elementos no início e no fim da lista.
- É fácil remover elementos do início da lista.

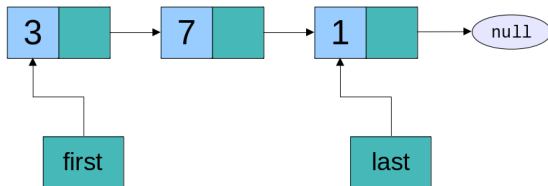
- Exemplo: lista com os elementos 3, 7 e 1.



- A lista possui acesso direto ao primeiro e último elementos.
- É fácil acrescentar elementos no início e no fim da lista.
- É fácil remover elementos do início da lista.

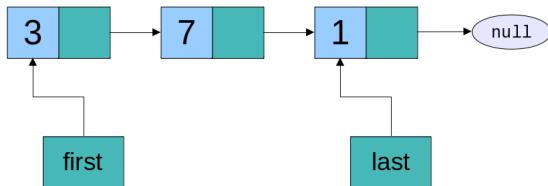


- Exemplo: lista com os elementos 3, 7 e 1.



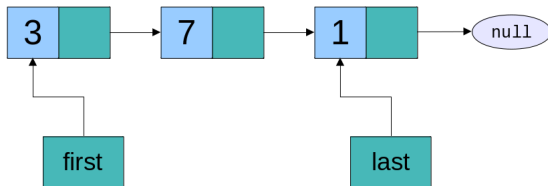
- A lista possui acesso direto ao primeiro e último elementos.
- É fácil acrescentar elementos no início e no fim da lista.
- É fácil remover elementos do início da lista.

- Exemplo: lista com os elementos 3, 7 e 1.



- A lista possui acesso direto ao primeiro e último elementos.
- É fácil acrescentar elementos no início e no fim da lista.
- É fácil remover elementos do início da lista.

- Exemplo: lista com os elementos 3, 7 e 1.



- A lista possui acesso direto ao primeiro e último elementos.
- É fácil acrescentar elementos no início e no fim da lista.
- É fácil remover elementos do início da lista.

# Nós para uma lista de inteiros

```
class NodeInt {  
  
    final int elem;  
    NodeInt next;  
  
    NodeInt(int e, NodeInt n) {  
        elem = e;  
        next = n;  
    }  
  
    NodeInt(int e) {  
        elem = e;  
        next = null;  
    }  
}
```

# Nós para uma lista de inteiros

```
class NodeInt {  
  
    final int elem;  
    NodeInt next;  
  
    NodeInt(int e, NodeInt n) {  
        elem = e;  
        next = n;  
    }  
  
    NodeInt(int e) {  
        elem = e;  
        next = null;  
    }  
}
```

# Lista ligada: tipo de dados abstracto

## Lista Ligada

Polimorfismo  
Paramétrico

Processamento  
recursivo de listas

- Nome do módulo:

- `LinkedList`

- Serviços:

- `addFirst()`: insere um elemento no início da lista.

- `addLast()`: insere um elemento no fim da lista.

- `First()`: devolve o primeiro elemento da lista.

- `Last()`: devolve o último elemento da lista.

- `removeFirst()`: remove o elemento no início da lista.

- `size()`: devolve a dimensão actual da lista.

- `isEmpty()`: verifica se a lista está vazia.

- `clear()`: limpa a lista (remove todos os elementos).

- Nome do módulo:
  - `LinkedList`
- Serviços:
  - `addFirst`: insere um elemento no início da lista.
  - `addLast`: insere um elemento no fim da lista.
  - `first`: devolve o primeiro elemento da lista.
  - `last`: devolve o último elemento da lista.
  - `removeFirst`: retira o elemento no início da lista.
  - `size`: devolve a dimensão actual da lista.
  - `isEmpty`: verifica se a lista está vazia.
  - `clear`: limpa a lista (remove todos os elementos).

- Nome do módulo:
  - LinkedList
- Serviços:
  - `addFirst`: insere um elemento no início da lista.
  - `addLast`: insere um elemento no fim da lista.
  - `first`: devolve o primeiro elemento da lista.
  - `last`: devolve o último elemento da lista.
  - `removeFirst`: retira o elemento no início da lista.
  - `size`: devolve a dimensão actual da lista.
  - `isEmpty`: verifica se a lista está vazia.
  - `clear`: limpa a lista (remove todos os elementos).



- Nome do módulo:
  - `LinkedList`
- Serviços:
  - `addFirst`: insere um elemento no início da lista.
  - `addLast`: insere um elemento no fim da lista.
  - `first`: devolve o primeiro elemento da lista.
  - `last`: devolve o último elemento da lista.
  - `removeFirst`: retira o elemento no início da lista.
  - `size`: devolve a dimensão actual da lista.
  - `isEmpty`: verifica se a lista está vazia.
  - `clear`: limpa a lista (remove todos os elementos).

- Nome do módulo:
  - `LinkedList`
- Serviços:
  - `addFirst`: insere um elemento no início da lista.
  - `addLast`: insere um elemento no fim da lista.
  - `first`: devolve o primeiro elemento da lista.
  - `last`: devolve o último elemento da lista.
  - `removeFirst`: retira o elemento no início da lista.
  - `size`: devolve a dimensão actual da lista.
  - `isEmpty`: verifica se a lista está vazia.
  - `clear`: limpa a lista (remove todos os elementos).

- Nome do módulo:
  - `LinkedList`
- Serviços:
  - `addFirst`: insere um elemento no início da lista.
  - `addLast`: insere um elemento no fim da lista.
  - `first`: devolve o primeiro elemento da lista.
  - `last`: devolve o último elemento da lista.
  - `removeFirst`: retira o elemento no início da lista.
  - `size`: devolve a dimensão actual da lista.
  - `isEmpty`: verifica se a lista está vazia.
  - `clear`: limpa a lista (remove todos os elementos).

- Nome do módulo:
  - `LinkedList`
- Serviços:
  - `addFirst`: insere um elemento no início da lista.
  - `addLast`: insere um elemento no fim da lista.
  - `first`: devolve o primeiro elemento da lista.
  - `last`: devolve o último elemento da lista.
  - `removeFirst`: retira o elemento no início da lista.
  - `size`: devolve a dimensão actual da lista.
  - `isEmpty`: verifica se a lista está vazia.
  - `clear`: limpa a lista (remove todos os elementos).

- Nome do módulo:
  - `LinkedList`
- Serviços:
  - `addFirst`: insere um elemento no início da lista.
  - `addLast`: insere um elemento no fim da lista.
  - `first`: devolve o primeiro elemento da lista.
  - `last`: devolve o último elemento da lista.
  - `removeFirst`: retira o elemento no início da lista.
  - `size`: devolve a dimensão actual da lista.
  - `isEmpty`: verifica se a lista está vazia.
  - `clear`: limpa a lista (remove todos os elementos).

- Nome do módulo:
  - `LinkedList`
- Serviços:
  - `addFirst`: insere um elemento no início da lista.
  - `addLast`: insere um elemento no fim da lista.
  - `first`: devolve o primeiro elemento da lista.
  - `last`: devolve o último elemento da lista.
  - `removeFirst`: retira o elemento no início da lista.
  - `size`: devolve a dimensão actual da lista.
  - `isEmpty`: verifica se a lista está vazia.
  - `clear`: limpa a lista (remove todos os elementos).

- Nome do módulo:
  - `LinkedList`
- Serviços:
  - `addFirst`: insere um elemento no início da lista.
  - `addLast`: insere um elemento no fim da lista.
  - `first`: devolve o primeiro elemento da lista.
  - `last`: devolve o último elemento da lista.
  - `removeFirst`: retira o elemento no início da lista.
  - `size`: devolve a dimensão actual da lista.
  - `isEmpty`: verifica se a lista está vazia.
  - `clear`: limpa a lista (remove todos os elementos).

- Nome do módulo:
  - `LinkedList`
- Serviços:
  - `addFirst`: insere um elemento no início da lista.
  - `addLast`: insere um elemento no fim da lista.
  - `first`: devolve o primeiro elemento da lista.
  - `last`: devolve o último elemento da lista.
  - `removeFirst`: retira o elemento no início da lista.
  - `size`: devolve a dimensão actual da lista.
  - `isEmpty`: verifica se a lista está vazia.
  - `clear`: limpa a lista (remove todos os elementos).



- Nome do módulo:
  - `LinkedList`
- Serviços:
  - `addFirst`: insere um elemento no início da lista.
  - `addLast`: insere um elemento no fim da lista.
  - `first`: devolve o primeiro elemento da lista.
  - `last`: devolve o último elemento da lista.
  - `removeFirst`: retira o elemento no início da lista.
  - `size`: devolve a dimensão actual da lista.
  - `isEmpty`: verifica se a lista está vazia.
  - `clear`: limpa a lista (remove todos os elementos).

- **addFirst(v)**
  - Pré-condição: !isEmpty() && (first() == null)
- **addLast(v)**
  - Pré-condição: !isEmpty() && (last() == null)
- **removeFirst()**
  - Pré-condição: !isEmpty()
- **first()**
  - Pré-condição: !isEmpty()

- **addFirst(v)**
  - Pós-condição: `!isEmpty() && (first() == v)`
- **addLast(v)**
  - Pós-condição: `!isEmpty() && (last() == v)`
- **removeFirst()**
  - Pré-condição: `!isEmpty()`
- **first()**
  - Pré-condição: `!isEmpty()`

- **addFirst(v)**
  - Pós-condição: `!isEmpty() && (first() == v)`
- **addLast(v)**
  - Pós-condição: `!isEmpty() && (last() == v)`
- **removeFirst()**
  - Pré-condição: `!isEmpty()`
- **first()**
  - Pré-condição: `!isEmpty()`

- **addFirst(v)**
  - Pós-condição: `!isEmpty() && (first() == v)`
- **addLast(v)**
  - Pós-condição: `!isEmpty() && (last() == v)`
- **removeFirst()**
  - Pré-condição: `!isEmpty()`
- **first()**
  - Pré-condição: `!isEmpty()`

- **addFirst(v)**
  - Pós-condição: `!isEmpty() && (first() == v)`
- **addLast(v)**
  - Pós-condição: `!isEmpty() && (last() == v)`
- **removeFirst()**
  - Pré-condição: `!isEmpty()`
- **first()**
  - Pré-condição: `!isEmpty()`

- **addFirst(v)**
  - Pós-condição: `!isEmpty() && (first() == v)`
- **addLast(v)**
  - Pós-condição: `!isEmpty() && (last() == v)`
- **removeFirst()**
  - Pré-condição: `!isEmpty()`
- **first()**
  - Pré-condição: `!isEmpty()`

- **addFirst(v)**
  - Pós-condição: `!isEmpty() && (first() == v)`
- **addLast(v)**
  - Pós-condição: `!isEmpty() && (last() == v)`
- **removeFirst()**
  - Pré-condição: `!isEmpty()`
- **first()**
  - Pré-condição: `!isEmpty()`



- **addFirst(v)**
  - Pós-condição: `!isEmpty() && (first() == v)`
- **addLast(v)**
  - Pós-condição: `!isEmpty() && (last() == v)`
- **removeFirst()**
  - Pré-condição: `!isEmpty()`
- **first()**
  - Pré-condição: `!isEmpty()`

- **addFirst(v)**
  - Pós-condição: `!isEmpty() && (first() == v)`
- **addLast(v)**
  - Pós-condição: `!isEmpty() && (last() == v)`
- **removeFirst()**
  - Pré-condição: `!isEmpty()`
- **first()**
  - Pré-condição: `!isEmpty()`

# Lista de inteiros: esqueleto da implementação

```
public class LinkedListInt {
    private NodeInt first=null, last=null;
    private int size;

    public LinkedListInt() { }
    public void addFirst(int e) {
        ...
        assert !isEmpty() && first()==e;
    }
    public void addLast(int e) {
        ...
        assert !isEmpty() && last()==e;
    }
    public int first() {
        assert !isEmpty();
        ...
    }
    public int last() {
        assert !isEmpty();
        ...
    }
    public void removeFirst() {
        assert !isEmpty();
        ...
    }
    public boolean isEmpty() { ... }
    public int size() { ... }
    public void clear() {
        ...
        assert isEmpty();
    }
}
```

# Lista de inteiros: esqueleto da implementação

```
public class LinkedListInt {
    private NodeInt first=null, last=null;
    private int size;

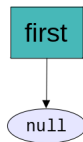
    public LinkedListInt() { }
    public void addFirst(int e) {
        ...
        assert !isEmpty() && first()==e;
    }
    public void addLast(int e) {
        ...
        assert !isEmpty() && last()==e;
    }
    public int first() {
        assert !isEmpty();
        ...
    }
    public int last() {
        assert !isEmpty();
        ...
    }
    public void removeFirst() {
        assert !isEmpty();
        ...
    }
    public boolean isEmpty() { ... }
    public int size() { ... }
    public void clear() {
        ...
        assert isEmpty();
    }
}
```

# Implementação de uma lista ligada

- `addFirst` - inserir o primeiro elemento.

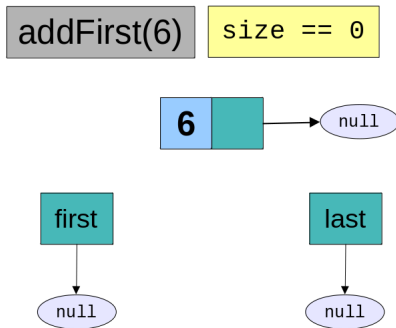
`addFirst(6)`

`size == 0`



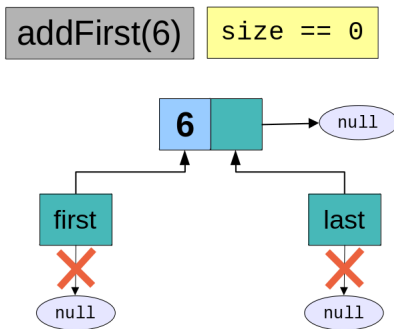
# Implementação de uma lista ligada

- `addFirst` - inserir o primeiro elemento.



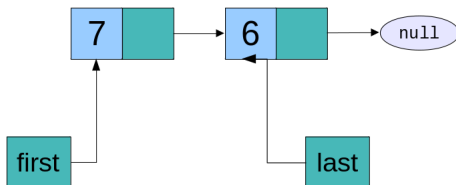
# Implementação de uma lista ligada

- `addFirst` - inserir o primeiro elemento.



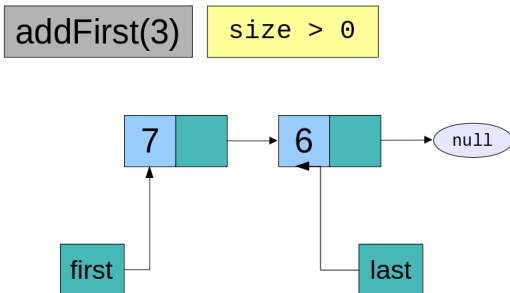
# Implementação de uma lista ligada

- `addFirst` - inserir novo elemento no início.

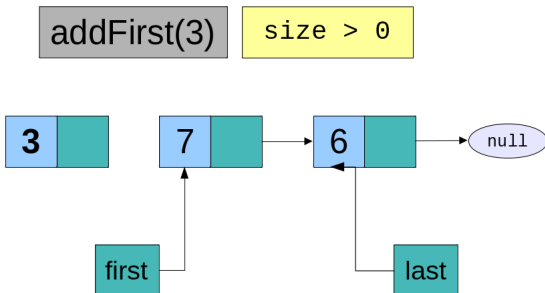




- `addFirst` - inserir novo elemento no início.

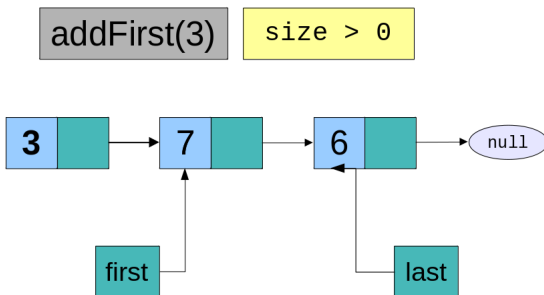


- `addFirst` - inserir novo elemento no início.



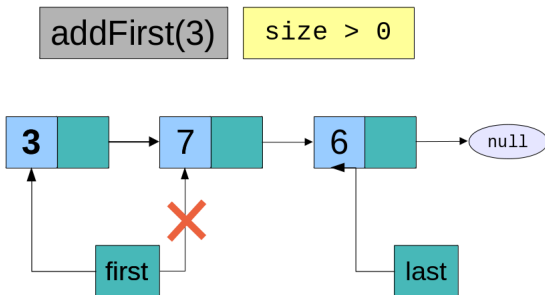
# Implementação de uma lista ligada

- `addFirst` - inserir novo elemento no início.



# Implementação de uma lista ligada

- `addFirst` - inserir novo elemento no início.

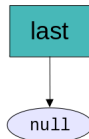
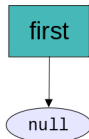


# Implementação de uma lista ligada

- `addLast` - acrescentar novo elemento no fim.
- Caso de lista vazia: similar a `addFirst`.

`addLast(1)`

`size == 0`



# Implementação de uma lista ligada

## Lista Ligada

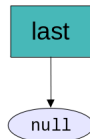
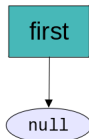
Polimorfismo  
Paramétrico

Processamento  
recursivo de listas

- `addLast` - acrescentar novo elemento no fim.
- Caso de lista vazia: similar a `addFirst`.

`addLast(1)`

`size == 0`

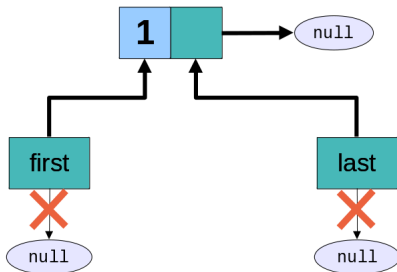


# Implementação de uma lista ligada

- `addLast` - acrescentar novo elemento no fim.
- Caso de lista vazia: similar a `addFirst`.

`addLast(1)`

`size == 0`



4

- `addLast` - acrescentar novo elemento no fim.

`addLast(4)`

`size > 0`

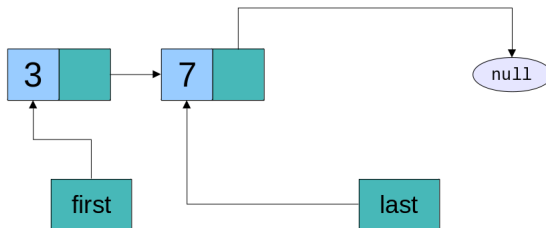


# Implementação de uma lista ligada

- `addLast` - acrescentar novo elemento no fim.

**`addLast(4)`**

`size > 0`

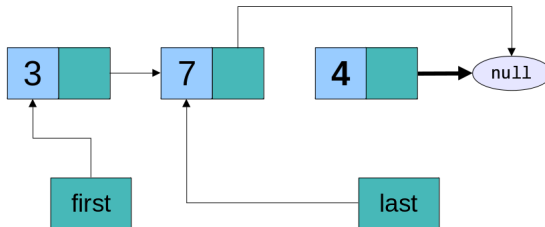


# Implementação de uma lista ligada

- `addLast` - acrescentar novo elemento no fim.

**`addLast(4)`**

`size > 0`

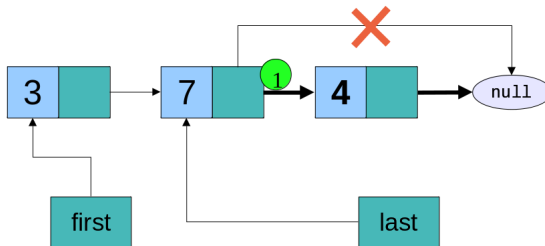


# Implementação de uma lista ligada

- `addLast` - acrescentar novo elemento no fim.

**`addLast(4)`**

`size > 0`



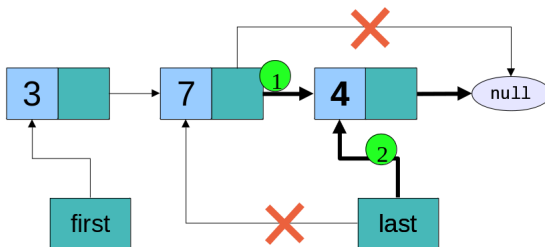
^

# Implementação de uma lista ligada

- `addLast` - acrescentar novo elemento no fim.

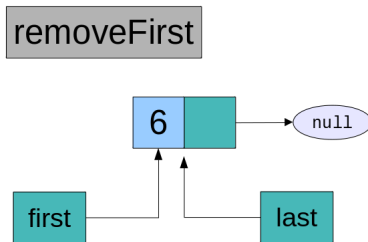
**`addLast(4)`**

`size > 0`



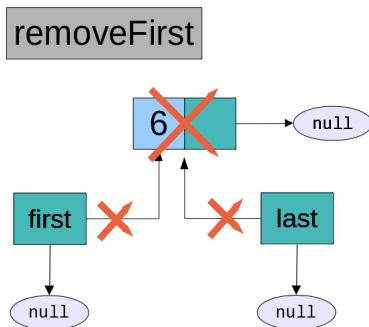
# Implementação de uma lista ligada

- `removeFirst` - remover o primeiro elemento.
- Quando `size==1`



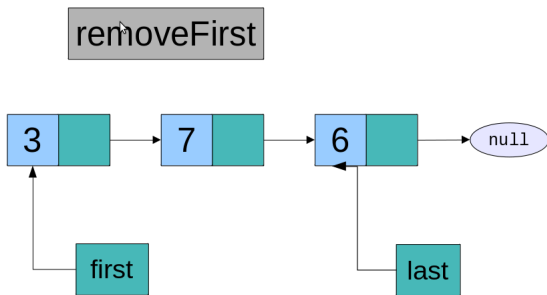
# Implementação de uma lista ligada

- `removeFirst` - remover o primeiro elemento.
- Quando `size==1`



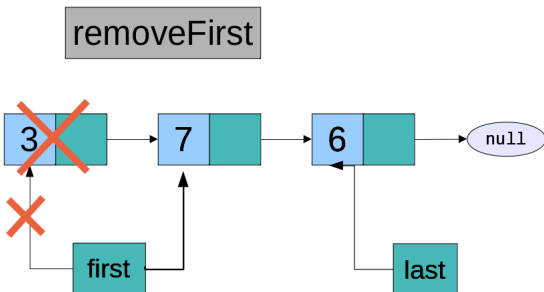
# Implementação de uma lista ligada

- `removeFirst` - remover o primeiro elemento.
- Quando `size > 1`



# Implementação de uma lista ligada

- `removeFirst` - remover o primeiro elemento.
- Quando `size > 1`





# Lista de inteiros: implementação completa

```
public class LinkedListInt {

    public void addFirst(int e) {
        first = new NodeInt(e, first);
        if (size == 0)
            last = first;
        size++;

        assert !isEmpty() && first() == e;
    }

    public void addLast(int e) {
        NodeInt n = new NodeInt(e);
        if (size == 0)
            first = n;
        else
            last.next = n;
        last = n;
        size++;

        assert !isEmpty() && last() == e;
    }

    public int size() {
        return size;
    }

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

```
    public void removeFirst() {
        assert !isEmpty();

        first = first.next;
        size--;
        if (first == null)
            last = null;
    }

    public int first() {
        assert !isEmpty();

        return first.elem;
    }

    public int last() {
        assert !isEmpty();

        return last.elem;
    }

    public void clear() {
        first = last = null;
        size = 0;
    }

    private NodeInt first = null;
    private NodeInt last = null;
    private int size = 0;
}
```

# Lista de inteiros: implementação completa

```
public class LinkedListInt {

    public void addFirst(int e) {
        first = new NodeInt(e, first);
        if (size == 0)
            last = first;
        size++;

        assert !isEmpty() && first() == e;
    }

    public void addLast(int e) {
        NodeInt n = new NodeInt(e);
        if (size == 0)
            first = n;
        else
            last.next = n;
        last = n;
        size++;

        assert !isEmpty() && last() == e;
    }

    public int size() {
        return size;
    }

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

```
    public void removeFirst() {
        assert !isEmpty();

        first = first.next;
        size--;
        if (first == null)
            last = null;
    }

    public int first() {
        assert !isEmpty();

        return first.elem;
    }

    public int last() {
        assert !isEmpty();

        return last.elem;
    }

    public void clear() {
        first = last = null;
        size = 0;
    }

    private NodeInt first = null;
    private NodeInt last = null;
    private int size = 0;
}
```

- **Problema:** A classe `LinkedListInt`:
  - Permite guardar apenas elementos inteiros.
  - Para termos listas com elementos de outros tipos, teríamos de duplicar o código e fazer pequenas alterações para adaptar ao tipo pretendido.
  - O código seria praticamente igual, mas não é possível ter esta "diversão" do código para cada novo elemento.
- **Solução:** Definir classes aplicáveis a qualquer tipo.
  - Deixar que são classes parametrizáveis por tipo, ou seja, o tipo do elemento passa a ser um parâmetro da classe.
  - As estruturas e funções passam a ser polimórficas.
  - Esta abordagem é conhecida como polimorfismo paramétrico.

- **Problema:** A classe `LinkedListInt`:
  - Permite guardar apenas elementos inteiros.
  - Para termos listas com elementos de outros tipos, teríamos de duplicar o código e fazer pequenas alterações para adaptar ao tipo pretendido.
  - O código seria praticamente igual, mas **não é prático** fazer esta “clonagem” de código para cada nova necessidade.
- **Solução:** Definir classes aplicáveis a qualquer tipo.
  - Diz-se que são classes parametrizadas por tipo, ou seja, o tipo de elemento passa a ser um parâmetro da classe.
  - As estruturas e funções passam a ser polimórficas.
  - Este mecanismo é conhecido como **polimorfismo paramétrico**.

- **Problema:** A classe `LinkedListInt`:
  - Permite guardar apenas elementos inteiros.
  - Para termos listas com elementos de outros tipos, teríamos de duplicar o código e fazer pequenas alterações para adaptar ao tipo pretendido.
  - O código seria praticamente igual, mas **não é prático** fazer esta “clonagem” de código para cada nova necessidade.
- **Solução:** Definir classes aplicáveis a qualquer tipo.
  - Diz-se que são classes parametrizadas por tipo, ou seja, o tipo de elemento passa a ser um parâmetro da classe.
  - As estruturas e funções passam a ser polimórficas.
  - Este mecanismo é conhecido como **polimorfismo paramétrico**.

- **Problema:** A classe `LinkedListInt`:
  - Permite guardar apenas elementos inteiros.
  - Para termos listas com elementos de outros tipos, teríamos de duplicar o código e fazer pequenas alterações para adaptar ao tipo pretendido.
  - O código seria praticamente igual, mas **não é prático** fazer esta “clonagem” de código para cada nova necessidade.
- **Solução:** Definir classes aplicáveis a qualquer tipo.
  - Diz-se que são classes parametrizadas por tipo, ou seja, o tipo de elemento passa a ser um parâmetro da classe.
  - As estruturas e funções passam a ser polimórficas.
  - Este mecanismo é conhecido como **polimorfismo paramétrico**.

- **Problema:** A classe `LinkedListInt`:
  - Permite guardar apenas elementos inteiros.
  - Para termos listas com elementos de outros tipos, teríamos de duplicar o código e fazer pequenas alterações para adaptar ao tipo pretendido.
  - O código seria praticamente igual, mas **não é prático** fazer esta “clonagem” de código para cada nova necessidade.
- **Solução:** Definir classes aplicáveis a qualquer tipo.
  - Diz-se que são classes parametrizadas por tipo, ou seja, o tipo de elemento passa a ser um parâmetro da classe.
  - As estruturas e funções passam a ser polimórficas.
  - Este mecanismo é conhecido como **polimorfismo paramétrico**.

- **Problema:** A classe `LinkedListInt`:
  - Permite guardar apenas elementos inteiros.
  - Para termos listas com elementos de outros tipos, teríamos de duplicar o código e fazer pequenas alterações para adaptar ao tipo pretendido.
  - O código seria praticamente igual, mas **não é prático** fazer esta “clonagem” de código para cada nova necessidade.
- **Solução:** Definir classes aplicáveis a qualquer tipo.
  - Diz-se que são classes parametrizadas por tipo, ou seja, o tipo de elemento passa a ser um parâmetro da classe.
  - As estruturas e funções passam a ser polimórficas.
  - Este mecanismo é conhecido como **polimorfismo paramétrico**.



- **Problema:** A classe `LinkedListInt`:
  - Permite guardar apenas elementos inteiros.
  - Para termos listas com elementos de outros tipos, teríamos de duplicar o código e fazer pequenas alterações para adaptar ao tipo pretendido.
  - O código seria praticamente igual, mas **não é prático** fazer esta “clonagem” de código para cada nova necessidade.
- **Solução:** Definir classes aplicáveis a qualquer tipo.
  - Diz-se que são classes parametrizadas por tipo, ou seja, o tipo de elemento passa a ser um parâmetro da classe.
  - As estruturas e funções passam a ser polimórficas.
  - Este mecanismo é conhecido como **polimorfismo paramétrico**.

- **Problema:** A classe `LinkedListInt`:
  - Permite guardar apenas elementos inteiros.
  - Para termos listas com elementos de outros tipos, teríamos de duplicar o código e fazer pequenas alterações para adaptar ao tipo pretendido.
  - O código seria praticamente igual, mas **não é prático** fazer esta “clonagem” de código para cada nova necessidade.
- **Solução:** Definir classes aplicáveis a qualquer tipo.
  - Diz-se que são classes parametrizadas por tipo, ou seja, o tipo de elemento passa a ser um parâmetro da classe.
  - As estruturas e funções passam a ser polimórficas.
  - Este mecanismo é conhecido como **polimorfismo paramétrico**.

- **Problema:** A classe `LinkedListInt`:
  - Permite guardar apenas elementos inteiros.
  - Para termos listas com elementos de outros tipos, teríamos de duplicar o código e fazer pequenas alterações para adaptar ao tipo pretendido.
  - O código seria praticamente igual, mas **não é prático** fazer esta “clonagem” de código para cada nova necessidade.
- **Solução:** Definir classes aplicáveis a qualquer tipo.
  - Diz-se que são classes parametrizadas por tipo, ou seja, o tipo de elemento passa a ser um parâmetro da classe.
  - As estruturas e funções passam a ser polimórficas.
  - Este mecanismo é conhecido como **polimorfismo paramétrico**.

# Tipos genéricos em Java

- Em Java, as classes que têm parâmetros que representam tipos são chamadas **classes genéricas**.
- Na *definição* de uma classe genérica, os **parâmetros de tipo** são indicados a seguir ao nome, entre < e >.

```
public class LinkedList<E> {           // generic class definition
    ...
    public void addFirst(E e) {        // use of type parameter E
        ...
    }
    ...
}
```

- Na *invocação e instanciação* de um tipo genérico os parâmetros são substituídos por **argumentos de tipo concretos**.

```
public static void main(String args[]) {
    ...
    LinkedList<Double> p1;              // generic type invocation
    p1 = new LinkedList<Double>();      // generic type instantiation
    ...
    LinkedList<Integer> p2 = new LinkedList<Integer>();
}
```

- Em Java, as classes que têm parâmetros que representam tipos são chamadas **classes genéricas**.
- Na *definição* de uma classe genérica, os **parâmetros de tipo** são indicados a seguir ao nome, entre < e >.

```
public class LinkedList<E> {           // generic class definition
    ...
    public void addFirst(E e) {        // use of type parameter E
        ...
    }
    ...
}
```

- Na *invocação e instanciação* de um tipo genérico os parâmetros são substituídos por **argumentos de tipo** concretos.

```
public static void main(String args[]) {
    ...
    LinkedList<Double> p1;              // generic type invocation
    p1 = new LinkedList<Double>();      // generic type instantiation
    ...
    LinkedList<Integer> p2 = new LinkedList<Integer>();
}
```

Lista Ligada

Polimorfismo  
Paramétrico

Processamento  
recursivo de listas

- Em Java, as classes que têm parâmetros que representam tipos são chamadas **classes genéricas**.
- Na *definição* de uma classe genérica, os **parâmetros de tipo** são indicados a seguir ao nome, entre < e >.

```
public class LinkedList<E> {           // generic class definition
    ...
    public void addFirst(E e) {        // use of type parameter E
        ...
    }
    ...
}
```

- Na *invocação e instanciação* de um tipo genérico os parâmetros são substituídos por **argumentos de tipo** concretos.

```
public static void main(String args[]) {
    ...
    LinkedList<Double> p1;              // generic type invocation
    p1 = new LinkedList<Double>();      // generic type instantiation
    ...
    LinkedList<Integer> p2 = new LinkedList<Integer>();
}
```

- Em Java, as classes que têm parâmetros que representam tipos são chamadas **classes genéricas**.
- Na *definição* de uma classe genérica, os **parâmetros de tipo** são indicados a seguir ao nome, entre < e >.

```
public class LinkedList<E> {           // generic class definition
    ...
    public void addFirst(E e) {        // use of type parameter E
        ...
    }
    ...
}
```

- Na *invocação e instanciação* de um tipo genérico os parâmetros são substituídos por **argumentos de tipo** concretos.

```
public static void main(String args[]) {
    ...
    LinkedList<Double> p1;              // generic type invocation
    p1 = new LinkedList<Double>();      // generic type instantiation
    ...
    LinkedList<Integer> p2 = new LinkedList<Integer>();
}
```

# Convenção sobre nomes de parâmetros de tipo

- Em Java, por convenção, usam-se letras maiúsculas para os nomes dos parâmetros de tipo. Por exemplo:
  - E - *element*
  - K - *key*
  - N - *number*
  - T - *type*
  - V - *value*
- Assim, mais facilmente se distingue um nome que representa um tipo de outro que representa uma variável ou método, que começam (também por convenção) com letra minúscula (exemplo: `numberOfElements`).
- Para informação mais detalhada pode consultar o [tutorial da Oracle sobre tipos genéricos](#).



- Em Java, por convenção, usam-se letras maiúsculas para os nomes dos parâmetros de tipo. Por exemplo:
  - E - *element*
  - K - *key*
  - N - *number*
  - T - *type*
  - V - *value*
- Assim, mais facilmente se distingue um nome que representa um tipo de outro que representa uma variável ou método, que começam (também por convenção) com letra minúscula (exemplo: `numberOfElements`).
- Para informação mais detalhada pode consultar o [tutorial da Oracle sobre tipos genéricos](#).

- Em Java, por convenção, usam-se letras maiúsculas para os nomes dos parâmetros de tipo. Por exemplo:
  - E - *element*
  - K - *key*
  - N - *number*
  - T - *type*
  - V - *value*
- Assim, mais facilmente se distingue um nome que representa um tipo de outro que representa uma variável ou método, que começam (também por convenção) com letra minúscula (exemplo: `numberOfElements`).
- Para informação mais detalhada pode consultar o [tutorial da Oracle sobre tipos genéricos](#).

- Em Java, por convenção, usam-se letras maiúsculas para os nomes dos parâmetros de tipo. Por exemplo:
  - E - *element*
  - K - *key*
  - N - *number*
  - T - *type*
  - V - *value*
- Assim, mais facilmente se distingue um nome que representa um tipo de outro que representa uma variável ou método, que começam (também por convenção) com letra minúscula (exemplo: `numberOfElements`).
- Para informação mais detalhada pode consultar o [tutorial da Oracle sobre tipos genéricos](#).

- **Problema:** Não é possível instanciar tipos genéricos com argumentos de tipos primitivos! (int, short, long, byte, boolean, char, float, double);

```
LinkedList<int> lst = new LinkedList<>(); // ERRO!
```

- **Solução:**

- Utilizar os tipos referências correspondentes (Integer, Double, etc.);

```
LinkedList<Integer> lst = new LinkedList<>(); // OK!
```

- A linguagem faz a conversão automática entre os tipos primitivos e os tipos referências correspondentes (Integer e int, etc.).

- **Problema:** Não é possível instanciar tipos genéricos com argumentos de tipos primitivos! (int, short, long, byte, boolean, char, float, double);

```
LinkedList<int> lst = new LinkedList<>(); // ERRO!
```

- **Solução:**
  - Utilizar os tipos referência correspondentes (Integer, Double, etc.).

```
LinkedList<Integer> lst = new LinkedList<>(); // OK!
```

- A linguagem faz a conversão automática entre os tipos primitivos e os tipos referência respectivos (*boxing* e *unboxing*).

- **Problema:** Não é possível instanciar tipos genéricos com argumentos de tipos primitivos! (int, short, long, byte, boolean, char, float, double);

```
LinkedList<int> lst = new LinkedList<>(); // ERRO!
```

- **Solução:**
  - Utilizar os tipos referência correspondentes (Integer, Double, etc.).

```
LinkedList<Integer> lst = new LinkedList<>(); // OK!
```

- A linguagem faz a conversão automática entre os tipos primitivos e os tipos referência respectivos (*boxing* e *unboxing*).

- **Problema:** Não é possível instanciar tipos genéricos com argumentos de tipos primitivos! (int, short, long, byte, boolean, char, float, double);

```
LinkedList<int> lst = new LinkedList<>(); // ERRO!
```

- **Solução:**
  - Utilizar os tipos referência correspondentes (Integer, Double, etc.).

```
LinkedList<Integer> lst = new LinkedList<>(); // OK!
```

- A linguagem faz a conversão automática entre os tipos primitivos e os tipos referência respectivos (*boxing* e *unboxing*).

- **Problema:** Não é possível instanciar tipos genéricos com argumentos de tipos primitivos! (`int`, `short`, `long`, `byte`, `boolean`, `char`, `float`, `double`);

```
LinkedList<int> lst = new LinkedList<>(); // ERRO!
```

- **Solução:**
  - Utilizar os tipos referência correspondentes (`Integer`, `Double`, etc.).

```
LinkedList<Integer> lst = new LinkedList<>(); // OK!
```

- A linguagem faz a conversão automática entre os tipos primitivos e os tipos referência respectivos (*boxing* e *unboxing*).



# Tipos genéricos em Java: limitação 2

- **Problema:** Não é possível criar arrays genéricos!

```
T[] a = new T[maxSize]; // ERRO!
```

- **Solução:**

• Criar arrays de elementos do tipo `Object` e fazer a conversão de tipo para o array genérico.

```
T[] a = (T[]) new Object[maxSize];
```

• Para obter o array genérico pelo compilador como resultado desta conversão pode-se declarar as variáveis onde a conversão é feita a seguinte sintaxe:

```
Object[] objetos = new Object[100];  
genérico<T> genérico = (T[]) objetos;
```

- O tutorial oficial tem mais informação sobre estas e outras restrições na utilização de genéricos.

- **Problema:** Não é possível criar arrays genéricos!

```
T[] a = new T[maxSize]; // ERRO!
```

- **Solução:**

- Criar arrays de elementos do tipo Object e fazer a coerção de tipo para o *array* genérico:

```
T[] a = (T[]) new Object[maxSize];
```

- Para evitar o aviso gerado pelo compilador como resultado desta coerção pode-se associar ao método onde a coerção é feita a seguinte anotação:

```
@SuppressWarnings("unchecked")  
public Matrix<T>() { ... }
```

- O tutorial oficial tem mais informação sobre estas e outras restrições na utilização de genéricos.

- **Problema:** Não é possível criar arrays genéricos!

```
T[] a = new T[maxSize]; // ERRO!
```

- **Solução:**
  - Criar arrays de elementos do tipo Object e fazer a coerção de tipo para o *array* genérico:

```
T[] a = (T[]) new Object[maxSize];
```

- Para evitar o aviso gerado pelo compilador como resultado desta coerção pode-se associar ao método onde a coerção é feita a seguinte anotação:

```
@SuppressWarnings("unchecked")  
public Matrix<T>() { ... }
```

- O tutorial oficial tem mais informação sobre estas e outras restrições na utilização de genéricos.

- **Problema:** Não é possível criar arrays genéricos!

```
T[] a = new T[maxSize]; // ERRO!
```

- **Solução:**
  - Criar arrays de elementos do tipo Object e fazer a coerção de tipo para o *array* genérico:

```
T[] a = (T[]) new Object[maxSize];
```

- Para evitar o aviso gerado pelo compilador como resultado desta coerção pode-se associar ao método onde a coerção é feita a seguinte anotação:

```
@SuppressWarnings("unchecked")  
public Matrix<T>() { ... }
```

- O tutorial oficial tem mais informação sobre estas e outras restrições na utilização de genéricos.

- **Problema:** Não é possível criar arrays genéricos!

```
T[] a = new T[maxSize]; // ERRO!
```

- **Solução:**
  - Criar arrays de elementos do tipo Object e fazer a coerção de tipo para o *array* genérico:

```
T[] a = (T[]) new Object[maxSize];
```

- Para evitar o aviso gerado pelo compilador como resultado desta coerção pode-se associar ao método onde a coerção é feita a seguinte anotação:

```
@SuppressWarnings("unchecked")  
public Matrix<T>() { ... }
```

- O tutorial oficial tem mais informação sobre estas e outras restrições na utilização de genéricos.

- **Problema:** Não é possível criar arrays genéricos!

```
T[] a = new T[maxSize]; // ERRO!
```

- **Solução:**
  - Criar arrays de elementos do tipo Object e fazer a coerção de tipo para o *array* genérico:

```
T[] a = (T[]) new Object[maxSize];
```

- Para evitar o aviso gerado pelo compilador como resultado desta coerção pode-se associar ao método onde a coerção é feita a seguinte anotação:

```
@SuppressWarnings("unchecked")  
public Matrix<T>() { ... }
```

- O tutorial oficial tem mais informação sobre estas e outras restrições na utilização de genéricos.

# Lista ligada genérica: implementação completa

```
public class LinkedList<E> {  
  
    public void addFirst(E e) {  
        first = new Node<>(e, first);  
        if (size == 0)  
            last = first;  
        size++;  
  
        assert !isEmpty() && first().equals(e);  
    }  
  
    public void addLast(E e) {  
        Node<E> n = new Node<>(e);  
        if (size == 0)  
            first = n;  
        else  
            last.next = n;  
        last = n;  
        size++;  
  
        assert !isEmpty() && last().equals(e);  
    }  
  
    public int size() {  
        return size;  
    }  
  
    public boolean isEmpty() {  
        return size() == 0;  
    }  
}
```

```
    public void removeFirst() {  
        assert !isEmpty();  
  
        first = first.next;  
        size--;  
        if (isEmpty())  
            last = null;  
    }  
  
    public E first() {  
        assert !isEmpty();  
  
        return first.elem;  
    }  
  
    public E last() {  
        assert !isEmpty();  
  
        return last.elem;  
    }  
  
    public void clear() {  
        first = last = null;  
        size = 0;  
    }  
  
    private Node<E> first = null;  
    private Node<E> last = null;  
    private int size = 0;  
}
```

# Lista ligada genérica: implementação completa

```
public class LinkedList<E> {

    public void addFirst(E e) {
        first = new Node<>(e, first);
        if (size == 0)
            last = first;
        size++;

        assert !isEmpty() && first().equals(e);
    }

    public void addLast(E e) {
        Node<E> n = new Node<>(e);
        if (size == 0)
            first = n;
        else
            last.next = n;
        last = n;
        size++;

        assert !isEmpty() && last().equals(e);
    }

    public int size() {
        return size;
    }

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

```
    public void removeFirst() {
        assert !isEmpty();

        first = first.next;
        size--;
        if (isEmpty())
            last = null;
    }

    public E first() {
        assert !isEmpty();

        return first.elem;
    }

    public E last() {
        assert !isEmpty();

        return last.elem;
    }

    public void clear() {
        first = last = null;
        size = 0;
    }

    private Node<E> first = null;
    private Node<E> last = null;
    private int size = 0;
}
```



- Quando a acção a realizar implica aceder ao meio da lista, é preciso percorrer a lista até ao nó que vai ser alterado.
- Sendo uma estrutura recursiva, as listas prestam-se naturalmente à utilização de algoritmos recursivos.
- **Exemplo:** saber se um elemento  $e$  existe na lista.

Prova por indução de existência da recursividade:

- Variável local: pointer do nó actual ( $n$ ) ou seguinte ( $n->next$ ).
- Condição de base: está neste nó, desde que haja forma de detectar o fim da lista.

- Quando a acção a realizar implica aceder ao meio da lista, é preciso percorrer a lista até ao nó que vai ser alterado.
- Sendo uma estrutura recursiva, as listas prestam-se naturalmente à utilização de algoritmos recursivos.
- **Exemplo:** saber se um elemento  $e$  existe na lista.
  - Condições de terminação da recursividade:
    - Base: quando a lista é vazia ( $l == []$ )
    - Recursão: quando o elemento a procurar é igual ao actual ( $e == l.data$ )
  - Variabilidade: passar do nó actual ( $n$ ) ao seguinte ( $n.next$ ).
  - Convergência: está garantida, desde que haja forma de detetar o fim da lista.

- Quando a acção a realizar implica aceder ao meio da lista, é preciso percorrer a lista até ao nó que vai ser alterado.
- Sendo uma estrutura recursiva, as listas prestam-se naturalmente à utilização de algoritmos recursivos.
- **Exemplo:** saber se um elemento  $e$  existe na lista.
  - Condições de terminação da recursividade:
    - Base:  $e$  é igual ao primeiro elemento da lista.
    - Recursão:  $e$  é igual ao primeiro elemento da lista seguinte.
  - Variabilidade: passar do nó actual ( $n$ ) ao seguinte ( $n.next$ ).
  - Convergência: está garantida, desde que haja forma de detetar o fim da lista.

- Quando a acção a realizar implica aceder ao meio da lista, é preciso percorrer a lista até ao nó que vai ser alterado.
- Sendo uma estrutura recursiva, as listas prestam-se naturalmente à utilização de algoritmos recursivos.
- **Exemplo:** saber se um elemento  $e$  existe na lista.
  - Condições de terminação da recursividade:
    - Chegou ao fim da lista (devolve `false`), ou
    - Encontrou o elemento  $e$  (devolve `true`).
  - Variabilidade: passar do nó actual ( $n$ ) ao seguinte ( $n.next$ ).
  - Convergência: está garantida, desde que haja forma de detetar o fim da lista.

- Quando a acção a realizar implica aceder ao meio da lista, é preciso percorrer a lista até ao nó que vai ser alterado.
- Sendo uma estrutura recursiva, as listas prestam-se naturalmente à utilização de algoritmos recursivos.
- **Exemplo:** saber se um elemento *e* existe na lista.
  - Condições de terminação da recursividade:
    - Chegou ao fim da lista (devolve *false*), ou
    - Encontrou o elemento *e* (devolve *true*).
  - Variabilidade: passar do nó actual (*n*) ao seguinte (*n.next*).
  - Convergência: está garantida, desde que haja forma de detetar o fim da lista.

- Quando a acção a realizar implica aceder ao meio da lista, é preciso percorrer a lista até ao nó que vai ser alterado.
- Sendo uma estrutura recursiva, as listas prestam-se naturalmente à utilização de algoritmos recursivos.
- **Exemplo:** saber se um elemento  $e$  existe na lista.
  - Condições de terminação da recursividade:
    - Chegou ao fim da lista (devolve `false`), ou
    - Encontrou o elemento  $e$  (devolve `true`).
  - Variabilidade: passar do nó actual ( $n$ ) ao seguinte ( $n.next$ ).
  - Convergência: está garantida, desde que haja forma de detetar o fim da lista.

- Quando a acção a realizar implica aceder ao meio da lista, é preciso percorrer a lista até ao nó que vai ser alterado.
- Sendo uma estrutura recursiva, as listas prestam-se naturalmente à utilização de algoritmos recursivos.
- **Exemplo:** saber se um elemento  $e$  existe na lista.
  - Condições de terminação da recursividade:
    - Chegou ao fim da lista (devolve `false`), ou
    - Encontrou o elemento  $e$  (devolve `true`).
  - Variabilidade: passar do nó actual ( $n$ ) ao seguinte ( $n.next$ ).
  - Convergência: está garantida, desde que haja forma de detetar o fim da lista.

- Quando a acção a realizar implica aceder ao meio da lista, é preciso percorrer a lista até ao nó que vai ser alterado.
- Sendo uma estrutura recursiva, as listas prestam-se naturalmente à utilização de algoritmos recursivos.
- **Exemplo:** saber se um elemento *e* existe na lista.
  - Condições de terminação da recursividade:
    - Chegou ao fim da lista (devolve `false`), ou
    - Encontrou o elemento *e* (devolve `true`).
  - Variabilidade: passar do nó actual (*n*) ao seguinte (*n.next*).
  - Convergência: está garantida, desde que haja forma de detetar o fim da lista.



- Quando a acção a realizar implica aceder ao meio da lista, é preciso percorrer a lista até ao nó que vai ser alterado.
- Sendo uma estrutura recursiva, as listas prestam-se naturalmente à utilização de algoritmos recursivos.
- **Exemplo:** saber se um elemento  $e$  existe na lista.
  - Condições de terminação da recursividade:
    - Chegou ao fim da lista (devolve `false`), ou
    - Encontrou o elemento  $e$  (devolve `true`).
  - Variabilidade: passar do nó actual ( $n$ ) ao seguinte ( $n.next$ ).
  - Convergência: está garantida, desde que haja forma de detetar o fim da lista.

## Exemplo: lista contém elemento

- Versão recursiva:

```
public boolean contains(E e) {  
    return contains(first, e);  
}  
  
private boolean contains(Node<E> n, E e) {  
    if (n == null) return false;           // condicao terminacao 1  
    if (n.elem.equals(e)) return true;     // condicao terminacao 2  
    return contains(n.next, e);           // chamada recursiva  
}
```

- Versão iterativa:

```
public boolean contains(E e) {  
    Node<E> n = first;  
    while (n != null) {                    // condicao terminacao 1  
        if (n.elem.equals(e)) return true; // condicao terminacao 2  
        n = n.next;                       // continuacao  
    }  
    return false;  
}
```

## Exemplo: lista contém elemento

- Versão recursiva:

```
public boolean contains(E e) {  
    return contains(first, e);  
}  
  
private boolean contains(Node<E> n, E e) {  
    if (n == null) return false;           // condicao terminacao 1  
    if (n.elem.equals(e)) return true;     // condicao terminacao 2  
    return contains(n.next, e);           // chamada recursiva  
}
```

- Versão iterativa:

```
public boolean contains(E e) {  
    Node<E> n = first;  
    while (n != null) {                    // condicao terminacao 1  
        if (n.elem.equals(e)) return true; // condicao terminacao 2  
        n = n.next;                       // continuacao  
    }  
    return false;  
}
```

## Exemplo: lista contém elemento

- Versão recursiva:

```
public boolean contains(E e) {  
    return contains(first, e);  
}  
  
private boolean contains(Node<E> n, E e) {  
    if (n == null) return false;           // condicao terminacao 1  
    if (n.elem.equals(e)) return true;     // condicao terminacao 2  
    return contains(n.next, e);           // chamada recursiva  
}
```

- Versão iterativa:

```
public boolean contains(E e) {  
    Node<E> n = first;  
    while (n != null) {                    // condicao terminacao 1  
        if (n.elem.equals(e)) return true; // condicao terminacao 2  
        n = n.next;                       // continuacao  
    }  
    return false;  
}
```

# Um padrão que se repete...

- Muitas funções sobre listas fazem uma travessia da lista.
- Essa travessia segue um padrão que convém desde já assimilar.

## Implementação Iterativa

```
public class LinkedList<E> {  
    ...  
    public ... xpto(...) {  
        Node<E> n = first;  
        ...  
        while (n!=null && ...) {  
            ...  
            n = n.next;  
        }  
        return ...;  
    }  
    ...  
}
```

## Implementação Recursiva

```
public class LinkedList<E> {  
    ...  
    public ... xpto(...) {  
        return xpto(first, ...);  
    }  
    private ... xpto(Node<E> n, ...) {  
        if (n == null) return ...;  
        ...  
        ... xpto(n.next, ...);  
        return ...  
    }  
    ...  
}
```

# Um padrão que se repete...

- Muitas funções sobre listas fazem uma travessia da lista.
- Essa travessia segue um padrão que convém desde já assimilar.

## Implementação Iterativa

```
public class LinkedList<E> {  
    ...  
    public ... xpto(...) {  
        Node<E> n = first;  
        ...  
        while (n!=null && ...) {  
            ...  
            n = n.next;  
        }  
        return ...;  
    }  
    ...  
}
```

## Implementação Recursiva

```
public class LinkedList<E> {  
    ...  
    public ... xpto(...) {  
        return xpto(first, ...);  
    }  
    private ... xpto(Node<E> n, ...) {  
        if (n == null) return ...;  
        ...  
        ... xpto(n.next, ...);  
        return ...  
    }  
    ...  
}
```

## Um padrão que se repete...

- Muitas funções sobre listas fazem uma travessia da lista.
- Essa travessia segue um padrão que convém desde já assimilar.

### Implementação Iterativa

```
public class LinkedList<E> {  
    ...  
    public ... xpto(...) {  
        Node<E> n = first;  
        ...  
        while (n!=null && ...) {  
            ...  
            n = n.next;  
        }  
        return ...;  
    }  
    ...  
}
```

### Implementação Recursiva

```
public class LinkedList<E> {  
    ...  
    public ... xpto(...) {  
        return xpto(first, ...);  
    }  
    private ... xpto(Node<E> n, ...) {  
        if (n == null) return ...;  
        ...  
        ... xpto(n.next, ...);  
        return ...  
    }  
    ...  
}
```

## Um padrão que se repete...

- Muitas funções sobre listas fazem uma travessia da lista.
- Essa travessia segue um padrão que convém desde já assimilar.

### Implementação Iterativa

```
public class LinkedList<E> {  
    ...  
    public ... xpto(...) {  
        Node<E> n = first;  
        ...  
        while (n!=null && ...) {  
            ...  
            n = n.next;  
        }  
        return ...;  
    }  
    ...  
}
```

### Implementação Recursiva

```
public class LinkedList<E> {  
    ...  
    public ... xpto(...) {  
        return xpto(first, ...);  
    }  
    private ... xpto(Node<E> n, ...) {  
        if (n == null) return ...;  
        ...  
        ... xpto(n.next, ...);  
        return ...  
    }  
    ...  
}
```



## Um padrão que se repete...

- Muitas funções sobre listas fazem uma travessia da lista.
- Essa travessia segue um padrão que convém desde já assimilar.

### Implementação Iterativa

```
public class LinkedList<E> {  
    ...  
    public ... xpto(...) {  
        Node<E> n = first;  
        ...  
        while (n!=null && ...) {  
            ...  
            n = n.next;  
        }  
        return ...;  
    }  
    ...  
}
```

### Implementação Recursiva

```
public class LinkedList<E> {  
    ...  
    public ... xpto(...) {  
        return xpto(first, ...);  
    }  
    private ... xpto(Node<E> n, ...) {  
        if (n == null) return ...;  
        ...  
        ... xpto(n.next, ...);  
        return ...  
    }  
    ...  
}
```

# Um padrão que se repete...

- Muitas funções sobre listas fazem uma travessia da lista.
- Essa travessia segue um padrão que convém desde já assimilar.

## Implementação Iterativa

```
public class LinkedList<E> {  
    ...  
    public ... xpto(...) {  
        Node<E> n = first;  
        ...  
        while (n!=null && ...) {  
            ...  
            n = n.next;  
        }  
        return ...;  
    }  
    ...  
}
```

## Implementação Recursiva

```
public class LinkedList<E> {  
    ...  
    public ... xpto(...) {  
        return xpto(first, ...);  
    }  
    private ... xpto(Node<E> n, ...) {  
        if (n == null) return ...;  
        ...  
        ... xpto(n.next, ...);  
        return ...  
    }  
    ...  
}
```