

# Aula 07

## Estruturas de dados recursivas

### Listas ligadas

Programação II, 2019-2020

v1.4, 29-03-2020

DETI, Universidade de Aveiro

07.1

#### Objectivos:

- Estruturas de dados recursivas: listas ligadas;
- Polimorfismo paramétrico (tipos genéricos);
- Funções recursivas (continuação).

## Conteúdo

<b>1</b>	<b>Lista Ligada</b>	<b>2</b>
1.1	Implementação: <code>addFirst</code>	4
1.2	Implementação: <code>addLast</code>	5
1.3	Implementação: <code>removeFirst</code>	5
<b>2</b>	<b>Polimorfismo Paramétrico</b>	<b>7</b>
<b>3</b>	<b>Processamento recursivo de listas</b>	<b>9</b>

07.2

As estruturas de dados servem não só para registar e aceder a informação, como também para disciplinar essas utilizações. Em linguagens de programação com um sistema de tipos estático, como é o caso da linguagem Java, a correcção formal dessas utilizações é garantida em tempo de compilação, evitando as dificuldades envolvidas na depuração do programa em tempo de execução.

O sistema de tipos dá, grosso modo, duas garantias a um programa:

1. compatibilidade de tipos na atribuição de valores;
2. correcção na utilização (formal) de um membro da classe.

A primeira aplica-se tanto à instrução de atribuição propriamente dita, como também à passagem de argumentos a uma função, que pode ser vista como a atribuição de valores dos argumentos aos parâmetros formais correspondentes. A segunda garante que quando se utiliza um membro de uma classe (método ou campo), ele tem de existir e ser compatível no número e tipos dos eventuais argumentos (no caso de métodos).

Vamos seguir uma abordagem modular na apresentação e implementação de algumas estruturas de dados de propósito geral. Assim, começaremos por definir o seu tipo de dados abstracto (a sua interface e os respectivos contratos), partindo depois para algumas possíveis concretizações.

Nesta aula, apresentamos uma dessas estrutura de dados de propósito geral, a *lista ligada*. Em aulas seguintes, veremos as pilhas e filas, bem como diferentes tipos de dicionários.

# 1 Lista Ligada

## 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 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.

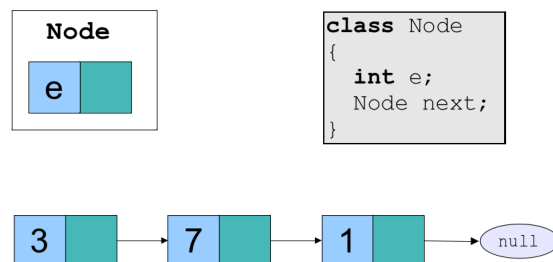
07.3

## Lista Ligada

- 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.

07.4

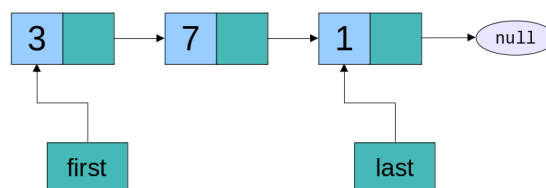
## Lista ligada simples: exemplo



07.5

## Lista ligada com dupla entrada

- 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.

07.6

## 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;
    }
}
```

07.7

## Lista ligada: tipo de dados abstracto

- 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).

07.8

## Lista ligada: semântica

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

07.9

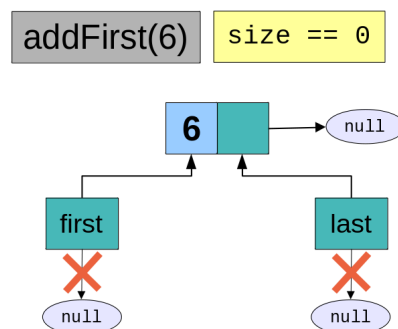
## Lista de inteiros: esqueleto da implementação

```
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();  
    }  
}
```

07.10

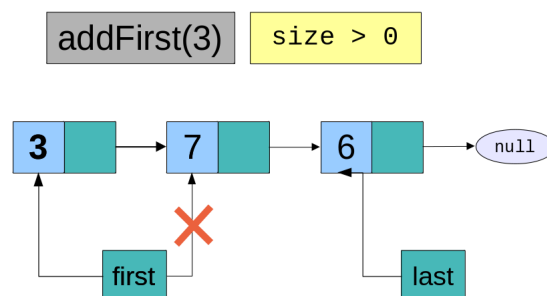
### 1.1 Implementação: addFirst

- addFirst - inserir o primeiro elemento.



07.11

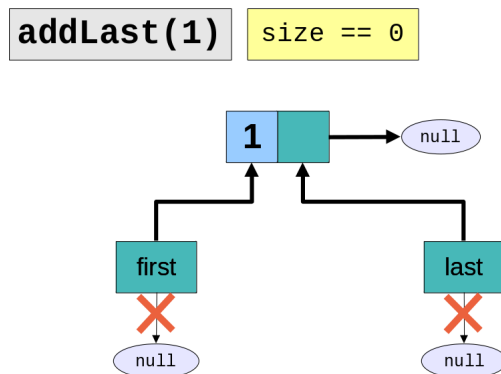
- addFirst - inserir novo elemento no início.



07.12

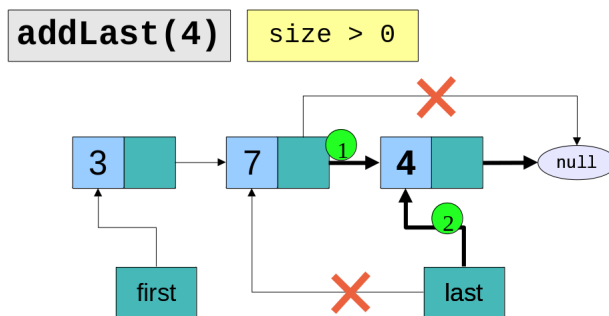
## 1.2 Implementação: addLast

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



07.13

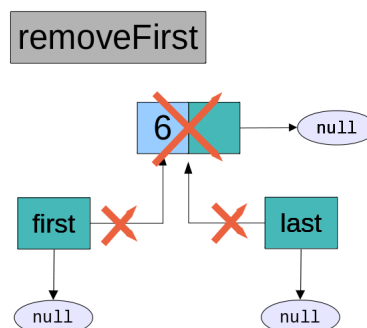
- addLast - acrescentar novo elemento no fim.



07.14

## 1.3 Implementação: removeFirst

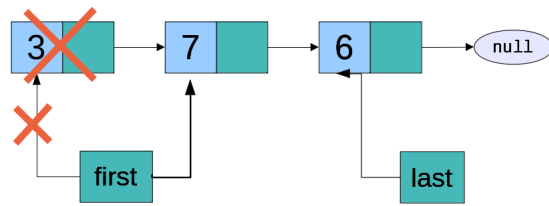
- removeFirst - remover o primeiro elemento.
- Quando size==1



07.15

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

removeFirst



07.16

## 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;  
}
```

07.17

## 2 Polimorfismo Paramétrico

### 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**.

07.18

### 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>();
}

```

---

07.19

## 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](#).

07.20

## Tipos genéricos em Java: limitação 1

- *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*).

07.21

## 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 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](#).

07.22



## 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;  
}
```

07.23

## 3 Processamento recursivo de listas

### Processamento recursivo de listas

- 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.

07.24

### 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) {  
        if (n.elem.equals(e)) return true; // condicao terminacao 2  
        n = n.next; // continuacao  
    }  
    return false;  
}
```

07.25

## 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	Implementação Recursiva
<pre>public class LinkedList&lt;E&gt; {     ...     public ... xpto(...) {         Node&lt;E&gt; n = first;         ...         while (n!=null &amp;&amp; ...) {             ...             n = n.next;         }         return ...;     }     ... }</pre>	<pre>public class LinkedList&lt;E&gt; {     ...     public ... xpto(...) {         return xpto(first, ...);     }     private ... xpto(Node&lt;E&gt; n, ...) {         if (n == null) return ...;         ...         ... xpto(n.next, ...);         return ...     }     ... }</pre>

07.26