

# Aula 08

## Recursão versus Iteração

*Recursão e Iteração em Estruturas Ordenadas*

Programação II, 2020-2021

2021-05-05

DETI, Universidade de Aveiro

08.1

### Conteúdo

<b>1</b>	<b>Recursão: implementação</b>	<b>1</b>
<b>2</b>	<b>Conversão entre recursão e iteração</b>	<b>3</b>
2.1	Iteração para recursão . . . . .	3
2.2	Recursão para iteração . . . . .	3
<b>3</b>	<b>Travessia de listas: recursão e iteração</b>	<b>4</b>
<b>4</b>	<b>Travessia de vectores: recursão e iteração</b>	<b>5</b>
<b>5</b>	<b>Gestão de listas e vectores ordenados</b>	<b>5</b>

08.2

### 1 Recursão: implementação

- Não há suporte direto para a recursão nas *linguagens de máquina*, isto é, linguagens que são directamente executadas pelos processadores (CPU) existentes nos computadores;
- Assim, para que este mecanismo funcione é necessária uma adequada implementação pelos compiladores (ou interpretadores) das linguagens de programação de mais alto nível (como o Java);

**Problema:** É necessário garantir uma separação clara entre o contexto do cliente (que invoca o método) e o contexto do método, de forma a impedir a interferência entre diferentes invocações do método.

08.3

### Recursão: implementação

- Este objetivo pode ser atingido fazendo com que cada execução de um método tenha um *contexto de execução* próprio onde são armazenadas as suas variáveis locais e parâmetros.
- É um processo análogo à instanciação de objectos, com uma diferença importante:
  - As variáveis do método são criadas quando o método inicia a sua execução e são descartadas quando termina.
- Uma forma que conseguir contextos de execução independentes baseia-se numa estrutura de dados composta designada por *pilha de chamada* (*call stack*), que se caracteriza por uma gestão do tipo *LIFO* (*Last In First Out*).

08.4

## Exemplo

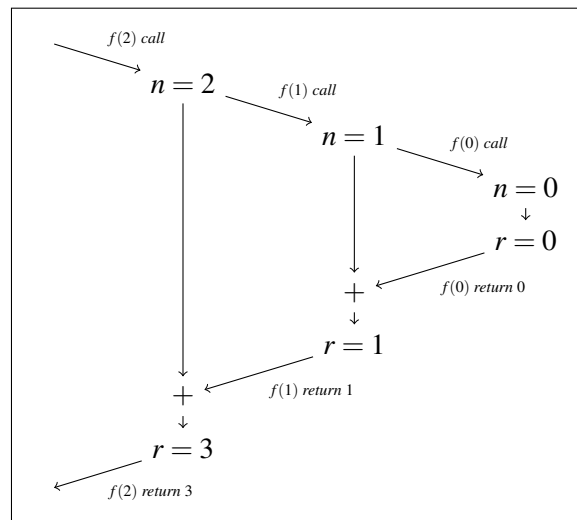
- Vejamos, como exemplo, a seguinte função recursiva  $f(n)$ , que devolve o somatório dos números de 0 a  $n$ :

```
static int f(int n) {  
    assert n >= 0;  
    //out.printf("f(%d)... \n", n);  
    int r = 0;  
    if (n > 0)  
        r = n + f(n-1);  
    //out.printf("f(%d) = %d\n", n, r);  
    return r;  
}
```

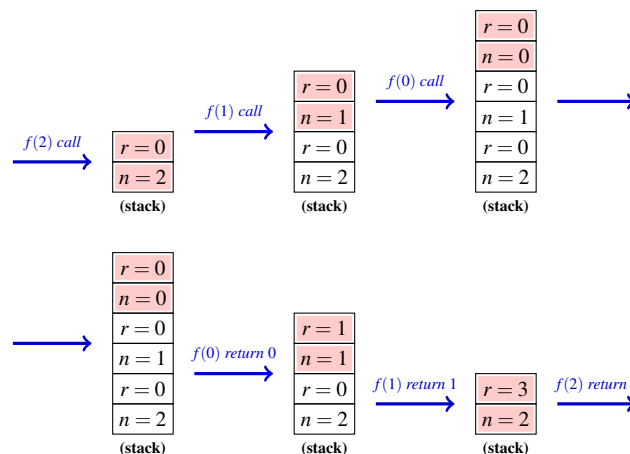
08.5

### Exemplo: execução de $f(2)$

(Experimente [este exemplo no Java Tutor](#).)



08.6



08.7

Note que a representação acima está um pouco simplificada. Na implementação real, para cada execução da função  $f$ , além das variáveis locais  $n$  e  $r$ , a pilha guarda também os endereços para aonde retornar.

## 2 Conversão entre recursão e iteração

### 2.1 Iteração para recursão

- Como já foi referido, um algoritmo recursivo tem sempre uma versão iterativa e vice-versa.
- Uma forma genérica de converter um ciclo (estruturado) numa função recursiva é a seguinte:

Implementação Iterativa	Implementação Recursiva
<pre>for (INIT; COND; UPDATE) {     BODY } ...</pre>	<pre>INIT loopEquiv (ARGS) ...  static void loopEquiv(ARGS decl) {     if (COND) {         BODY         UPDATE         loopEquiv (ARGS);     } }</pre>

- A função recursiva tem de declarar argumentos formais correspondentes às variáveis utilizadas no ciclo.
- Os valores dessas variáveis têm de ser passados como argumentos da função.
- Se precisarmos dos valores finais de algumas variáveis, então a função recursiva deverá *devolver* esses valores como resultado ou modificando atributos do objeto.
- Se o ciclo não for estruturado, ou seja, se contiver instruções do tipo “salto” (`break`, `continue` ou `return`), então terá de ser transformado num ciclo estruturado antes da conversão.

08.8

#### Iteração para recursão: exemplo

Implementação Iterativa	Implementação Recursiva
<pre>// int[] arr for(int i=0; i&lt;arr.length; i++)     out.println(arr[i]); ...</pre>	<pre>int i = 0; loopEquiv(arr, i); ...  static void loopEquiv(int[] arr, int i) {     if (i &lt; arr.length) {         out.println(arr[i]);         i++;         loopEquiv(arr, i);     } }</pre>

- Podemos melhorar esta implementação substituindo o incremento de `i` pela passagem de `i+1` para a função.

08.9

### 2.2 Recursão para iteração

- A conversão de algoritmos recursivos para ciclos é, em geral, mais complexa do que a transformação inversa.
- Uma forma geral de fazer essa conversão faz uso de uma *pilha* para armazenar explicitamente os contextos de execução da função recursiva (contendo os argumentos e variáveis locais da função) e substitui as chamadas das funções por instruções do tipo *salto* (*goto*).
- No entanto, o algoritmo resultante fica muito *menos legível*.
- Alguns tipos particulares de recursividade, como é o caso da *recursão de cauda* (*tail recursion*) prestam-se a optimizações interessantes (já que podemos prescindir do armazenamento de algum contexto).
- Mas isso sai fora do âmbito desta disciplina.

08.10

## Recursão para iteração: exemplo

- Certas funções recursivas (como o cálculo dos números de Fibonacci ou o factorial) são, no entanto, facilmente convertidas em ciclos:
  - Basta fazer a iteração desde o(s) caso(s) limite até ao valor desejado, e ir armazenando os valores calculados num array.
  - E substituir as invocações recursivas por acessos ao array.

Implementação Recursiva	Implementação Iterativa (com array)
<pre>static int factorial(int n) {     assert n &gt;= 0;     int res;      if (n &lt;= 1)         res = 1;     else         res = n * factorial(n-1);      return res; }</pre>	<pre>static int factorial(int n) {     assert n &gt;= 0;     int[] arr = new int[n+1];     for(int i = 0; i &lt;= n; i++) {         if (i &lt;= 1) // casos limite             arr[i] = 1;         else             arr[i] = i * arr[i-1];     }     return arr[n]; }</pre>

08.11

Por vezes, pode não ser necessário armazenar todos os valores anteriores. Nesses casos, pode otimizar-se o algoritmo iterativo para usar menos memória. (Pode fazer isso no exemplo acima.)

## 3 Travessia de listas: recursão e iteração

### Travessia de listas: recursão e iteração

- Embora as listas sejam estruturas de dados recursivas, é possível utilizar algoritmos iterativos.
- Vejamos novamente a função `contains()` da classe `LinkedList`, da aula anterior, comparando com uma versão iterativa equivalente.

Implementação Iterativa	Implementação Recursiva
<pre>public class LinkedList&lt;E&gt; {     ...     public boolean contains(E e) {         boolean found = false;         Node&lt;E&gt; n = first;         while (n!=null &amp;&amp; !found) {             if (n.elem.equals(e))                 found = true;             n = n.next;         }         return found;     }     ... }</pre>	<pre>public class LinkedList&lt;E&gt; {     ...     public boolean contains(E e) {         return contains(first, e);     }     private boolean contains(Node&lt;E&gt; n, E e) {         if (n == null) return false;         if (n.elem.equals(e)) return true;         return contains(n.next, e);     }     ... }</pre>

08.12

## Um padrão que se repete ...

- Muitas funções têm de fazer uma *travessia* da lista.
- Essa travessia segue um padrão que convém 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, e);     }     private ... xpto(Node&lt;E&gt; n, ...) {         if (n == null) return ...;         ...         ... xpto(n.next, ...);         return ...     }     ... }</pre>

**Travessia (= percurso):** Algoritmo que percorre potencialmente todos os elementos de uma estrutura de dados visitando cada um apenas uma vez.

08.13

## 4 Travessia de vectores: recursão e iteração

### Travessia de vectores: recursão e iteração

- Como faríamos uma pesquisa sequencial num vector?
- Aqui, em vez de passarmos de `n` a `n.next`, passamos de `i` a `i+1`.
- E, em vez de compararmos com `n.elem`, comparamos com o elemento `v[i]` do vector.

Implementação Iterativa	Implementação Recursiva
<pre>public static boolean contains(E[] v, E e) {     int i=0;     while (i &lt; v.length) {         if (v[i].equals(e))             return true;         i++;     }     return false; }</pre>	<pre>public static boolean contains(E[] v, E e) {     return contains(v, e, 0); }  private static boolean contains(E[] v, E e, int i) {     if (i &gt;= v.length) return false;     if (v[i].equals(e)) return true;     return contains(v, e, i+1); }</pre>

08.14

## 5 Gestão de listas e vectores ordenados

- Em muitas aplicações, dá jeito ter estruturas ordenadas.
  - O problema coloca-se quer para vectores, quer para listas.
- Na próxima aula, vamos ver diversos algoritmos de ordenação.
- Um problema mais simples é o de criar e manter uma estrutura sempre ordenada.
  - Dependendo da aplicação, pode ser preferível.
- Por simplicidade, vamos trabalhar com listas e vectores de elementos inteiros.

08.15

## Lista ligada ordenada: semântica

- **insert(e)** - inserir o elemento dado.
  - Pré-condição: `isSorted()`
  - Pós-condição: `contains(e) && isSorted()`
- **removeFirst()** - remover o primeiro elemento.
  - Pré-condição: `!isEmpty()`
- **first()** - consultar o primeiro elemento.
  - Pré-condição: `!isEmpty()`
- **remove(e)** - remover o elemento dado.
  - Pré-condição: `contains(e) && isSorted()`
  - Pós-condição: `isSorted()`

08.16

## Vector ordenado: semântica

- **insert(v, ne, e)** - inserir o elemento dado.
  - Pré-condição: `isSorted(v, ne) && !isFull(v, ne)`
  - Pós-cond.: `contains(v, ne, e) && isSorted(v, ne)`
- **removeFirst(v, ne)** - remover o primeiro elemento.
  - Pré-condição: `!isEmpty(v, ne)`
- **first(v)** - consultar o primeiro elemento.
  - Pré-condição: `!isEmpty(v, ne)`
- **remove(v, ne, e)** - remover o elemento dado.
  - Pré-cond.: `contains(v, ne, e) && isSorted(v, ne)`
  - Pós-condição: `isSorted(v, ne) && !isFull(v, ne)`
- ( `v` = vector, `ne` = número de elementos, `e` = elemento )

08.17

## Verificar se uma lista está ordenada: recursão e iteração

- Numa lista ordenada, qualquer função deve manter a lista ordenada.
- Precisamos assim de uma função que verifique isso.
- Essa verificação pode ser usada em asserções.
- Em cada passo, precisamos de conhecer o elemento anterior (`p`).

Implementação Iterativa	Implementação Recursiva
<pre>public class SortedListInt {     ...     public boolean isSorted() {         if (size &lt; 2)             return true;         NodeInt p = first;         NodeInt n = first.next;         while (n!=null) {             if (n.elem&lt;p.elem)                 return false             p = n; //previous             n = n.next;         }         return true;     }     ... }</pre>	<pre>public class SortedListInt {     ...     public boolean isSorted() {         if (size &lt; 2) return true;         return isSorted(first, first.next);     }     private     boolean isSorted(NodeInt p, NodeInt n) {         if (n == null) return true;         if (n.elem &lt; p.elem) return false;         return isSorted(n, n.next);     }     ... }</pre>



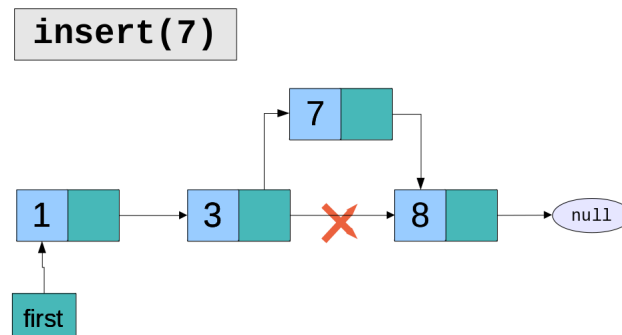
## Verificar se um vector está ordenado: recursão e iteração

Implementação Iterativa	Implementação Recursiva
<pre> public static boolean isSorted(int[] v) {     if (v.length &lt; 2)         return true;     int i = 1;     boolean sorted = true;     while (i!=v.length &amp;&amp; sorted) {         if (v[i] &lt; v[i-1])             sorted = false;         i++;     }     return sorted; } </pre>	<pre> public static boolean isSorted(int[] v) {     if (v.length &lt; 2)         return true;     return isSorted(v, 1); } private static boolean isSorted(int[] v, int i) {     if (i==v.length) return true;     if (v[i] &lt; v[i-1]) return false;     return isSorted(v, i+1); } </pre>

08.19

## Inserção numa lista ordenada

- Inserção no meio da lista:



- Quando o elemento fica no início, funciona como addFirst
- Quando o elemento fica no fim, funciona como addLast

08.20

## Inserção numa lista ordenada: recursão e iteração

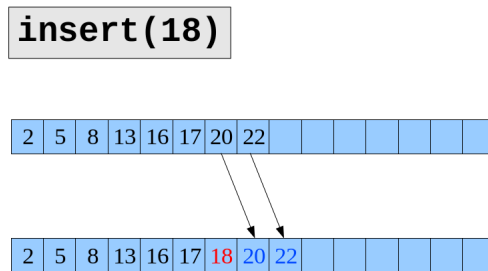
Implementação Iterativa	Implementação Recursiva
<pre> public class SortedListInt {     ...     public void insert(int e) {          if (first==null  e&lt;first.elem)             first = new NodeInt(e, first);         else {             NodeInt p = first;             NodeInt n = first.next;             while (n!=null &amp;&amp; e&gt;n.elem) {                 p = n;                 n = n.next;             }             p.next = new NodeInt(e, n);             size++;         }         ...     } } </pre>	<pre> public class SortedListInt {     ...     public void insert(int e) {         first = insert(first, e);         size++;     }     private     NodeInt insert(NodeInt n, int e) {         if (n==null    e&lt;n.elem)             return new NodeInt(e, n);         n.next = insert(n.next, e);         return n;     }     ... } </pre>

08.21



## Inserção num vector ordenado

- Inserção no meio do vector:



08.22

## Inserção num vector ordenado: recursão e iteração

- Inserir um elemento *e* num vector *v* com *ne* elementos

Implementação Iterativa	Implementação Recursiva
<pre>public static int insert(int[] v, int ne, int e) {     int i=ne;     while (i&gt;0 &amp;&amp; e&lt;v[i-1]) {         v[i] = v[i-1];         i--;     }     v[i] = e;     return ne+1; }</pre>	<pre>public static int insert(int[] v, int ne, int e) {     shiftInsert(v, e, ne);     return ne+1; }  public static void shiftInsert(int[] v, int e, int i) {     if (i==0    e&gt;v[i-1]) v[i] = e;     else {         v[i] = v[i-1];         shiftInsert(v, e, i-1);     } }</pre>

08.23

## Implementação de uma lista ordenada genérica

- Qualquer objecto Java tem o método `equals()`.
- No entanto, só alguns objectos têm o método `compareTo()` necessário para manter uma lista ordenada.
- Podemos definir classes genéricas em que os parâmetros de tipo são declarados como “comparáveis”.

```
public class SortedList<E extends Comparable<E>> {
    ...
    public void insert(E e) {
        ...
    }
    ...
}
...
public static void main(String args[]) {
    ...
    SortedList<Double> p1 = new SortedList<Double>();
    SortedList<Integer> p2 = new SortedList<Integer>();
    ...
}
```

08.24

