Aula 08

Recursão versus Iteração

Recursão e Iteração em Estruturas Ordenadas

Programação II, 2016-2017

v1.1, 02-04-2017

DETI, Universidade de Aveiro

Sumário

Sumário

Recursão: implementação

- Não há suporte directo 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

Permitir uma separação clara entre o código do cliente (que invoca o método) e o código do método, impedindo a interferência (indesejada) entre diferentes invocações do método (incluindo possíveis invocações recursivas).

Recursão: implementação

- Não há suporte directo 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

Permitir uma separação clara entre o código do cliente (que invoca o método) e o código do método, impedindo a interferência (indesejada) entre diferentes invocações do método (incluindo possíveis invocações recursivas).

Recursão: implementação

- Não há suporte directo 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

Permitir uma separação clara entre o código do cliente (que invoca o método) e o código do método, impedindo a interferência (indesejada) entre diferentes invocações do método (incluindo possíveis invocações recursivas).

Recursão: implementação

- Não há suporte directo 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

Permitir uma separação clara entre o código do cliente (que invoca o método) e o código do método, impedindo a interferência (indesejada) entre diferentes invocações do método (incluindo possíveis invocações recursivas).

Recursão: implementação

- Não há suporte directo 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

Permitir uma separação clara entre o código do cliente (que invoca o método) e o código do método, impedindo a interferência (indesejada) entre diferentes invocações do método (incluindo possíveis invocações recursivas).

Recursão: implementação

- Este objectivo pode ser atingido fazendo com que os métodos, sempre que são invocados, funcionem com contextos de execução próprios onde são armazenadas as suas variáveis (argumentos, variáveis locais e resultado da função).
- Podemos fazer uma analogia com a instanciação de objectos, com a diferença de as variáveis do método só existirem durante a execução do método.
  - As variáveis são criadas quando o método inicia a sua execução e destruídas quando termina.
- A implementação mais eficiente para este fim assenta numa estrutura de dados composta designada por *Pilha* (*stack*), que se caracteriza por uma gestão do tipo *LIFO* (*Last In First Out*);

Recursão: implementação

- Este objectivo pode ser atingido fazendo com que os métodos, sempre que são invocados, funcionem com contextos de execução próprios onde são armazenadas as suas variáveis (argumentos, variáveis locais e resultado da função).
- Podemos fazer uma analogia com a instanciação de objectos, com a diferença de as variáveis do método só existirem durante a execução do método.
  - As variáveis são criadas quando o método inicia a sua execução e descartadas quando termina.
- A implementação mais eficiente para este fim assenta numa estrutura de dados composta designada por *Pilha* (*stack*), que se caracteriza por uma gestão do tipo *LIFO* (*Last In First Out*);

Recursão: implementação

- Este objectivo pode ser atingido fazendo com que os métodos, sempre que são invocados, funcionem com contextos de execução próprios onde são armazenadas as suas variáveis (argumentos, variáveis locais e resultado da função).
- Podemos fazer uma analogia com a instanciação de objectos, com a diferença de as variáveis do método só existirem durante a execução do método.
  - As variáveis são criadas quando o método inicia a sua execução e descartadas quando termina.
- A implementação mais eficiente para este fim assenta numa estrutura de dados composta designada por *Pilha* (*stack*), que se caracteriza por uma gestão do tipo *LIFO* (*Last In First Out*);

Recursão: implementação

- Este objectivo pode ser atingido fazendo com que os métodos, sempre que são invocados, funcionem com contextos de execução próprios onde são armazenadas as suas variáveis (argumentos, variáveis locais e resultado da função).
- Podemos fazer uma analogia com a instanciação de objectos, com a diferença de as variáveis do método só existirem durante a execução do método.
  - As variáveis são criadas quando o método inicia a sua execução e descartadas quando termina.
- A implementação mais eficiente para este fim assenta numa estrutura de dados composta designada por *Pilha* (*stack*), que se caracteriza por uma gestão do tipo *LIFO* (*Last In First Out*);

Recursão: implementação

- Este objectivo pode ser atingido fazendo com que os métodos, sempre que são invocados, funcionem com contextos de execução próprios onde são armazenadas as suas variáveis (argumentos, variáveis locais e resultado da função).
- Podemos fazer uma analogia com a instanciação de objectos, com a diferença de as variáveis do método só existirem durante a execução do método.
  - As variáveis são criadas quando o método inicia a sua execução e descartadas quando termina.
- A implementação mais eficiente para este fim assenta numa estrutura de dados composta designada por *Pilha* (*stack*), que se caracteriza por uma gestão do tipo *LIFO* (*Last In First Out*);

Exemplo

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

```
static int m(int n)
{
   assert n >= 0;

   out.println("n = "+n);
   int r = 0;
   if (n > 0)
      r = n + m(n-1);
   out.println("r = "+r);
   return r;
}
```

Exemplo

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

```
static int m(int n)
{
   assert n >= 0;

   out.println("n = "+n);
   int r = 0;
   if (n > 0)
      r = n + m(n-1);
   out.println("r = "+r);
   return r;
}
```

Exemplo

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

```
static int m(int n)
{
   assert n >= 0;

   out.println("n = "+n);
   int r = 0;
   if (n > 0)
      r = n + m(n-1);
   out.println("r = "+r);
   return r;
}
```

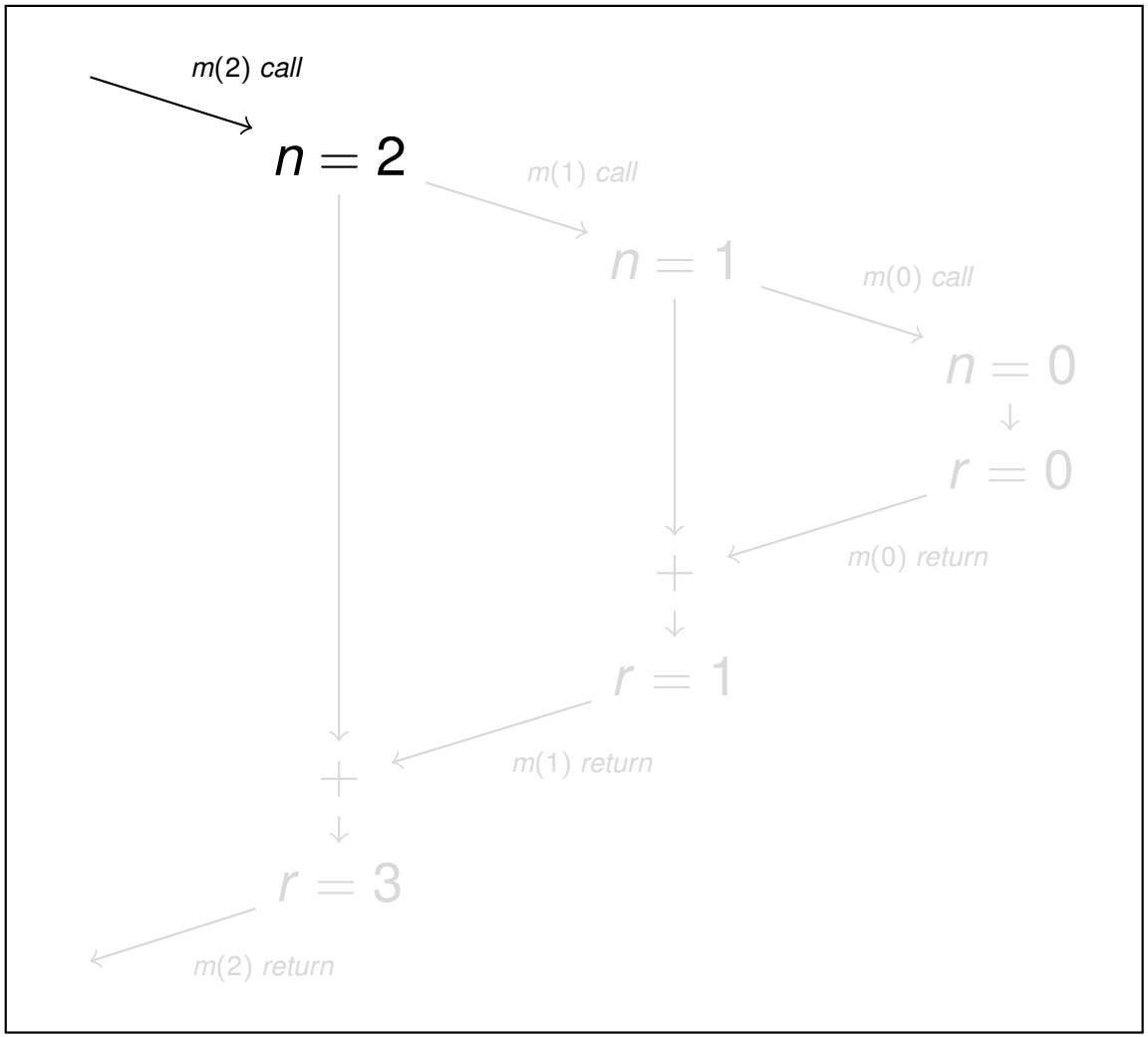
Exemplo

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

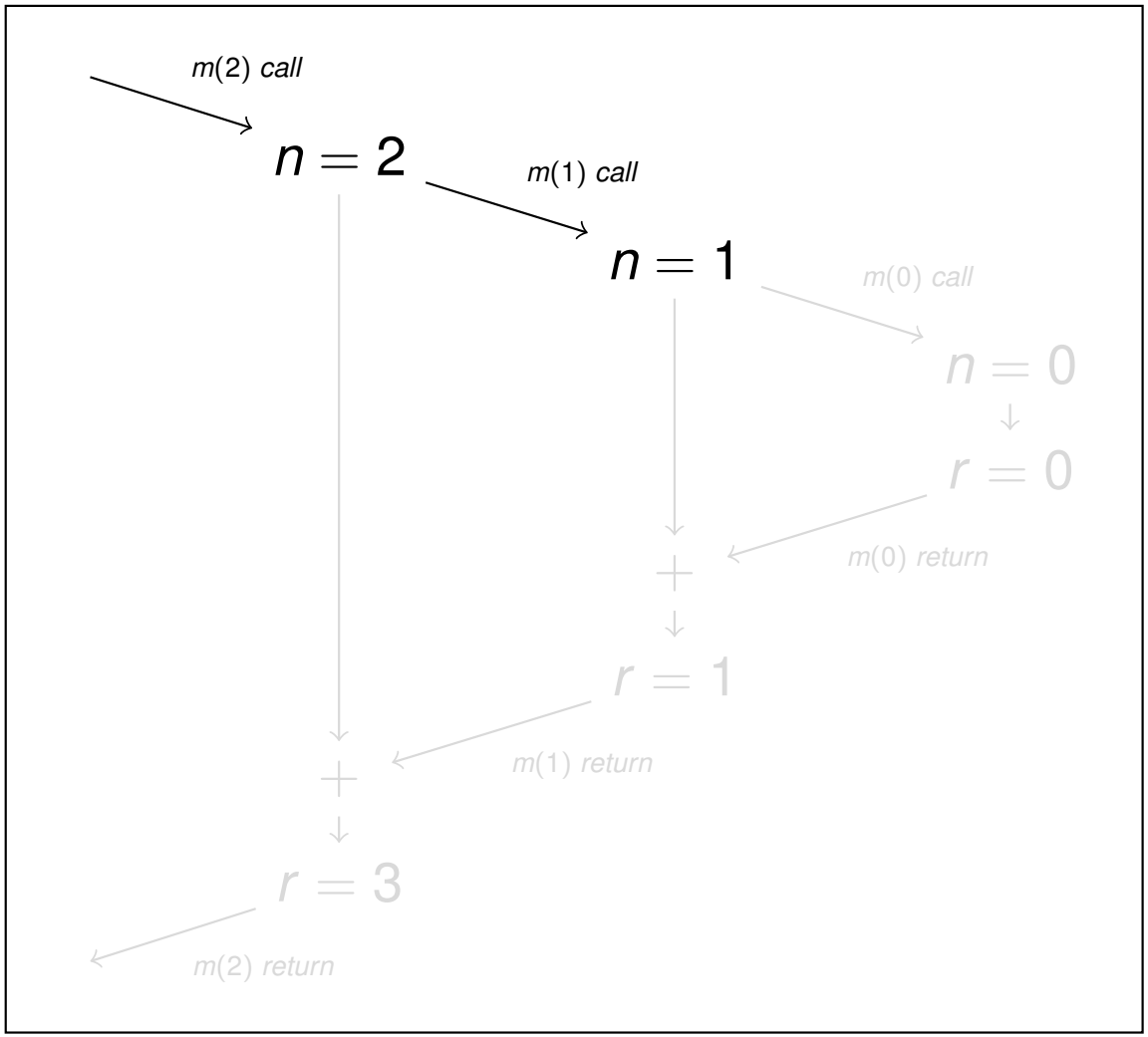
```
static int m(int n)
{
   assert n >= 0;

   out.println("n = "+n);
   int r = 0;
   if (n > 0)
      r = n + m(n-1);
   out.println("r = "+r);
   return r;
}
```

Exemplo: execução de $m(2)$

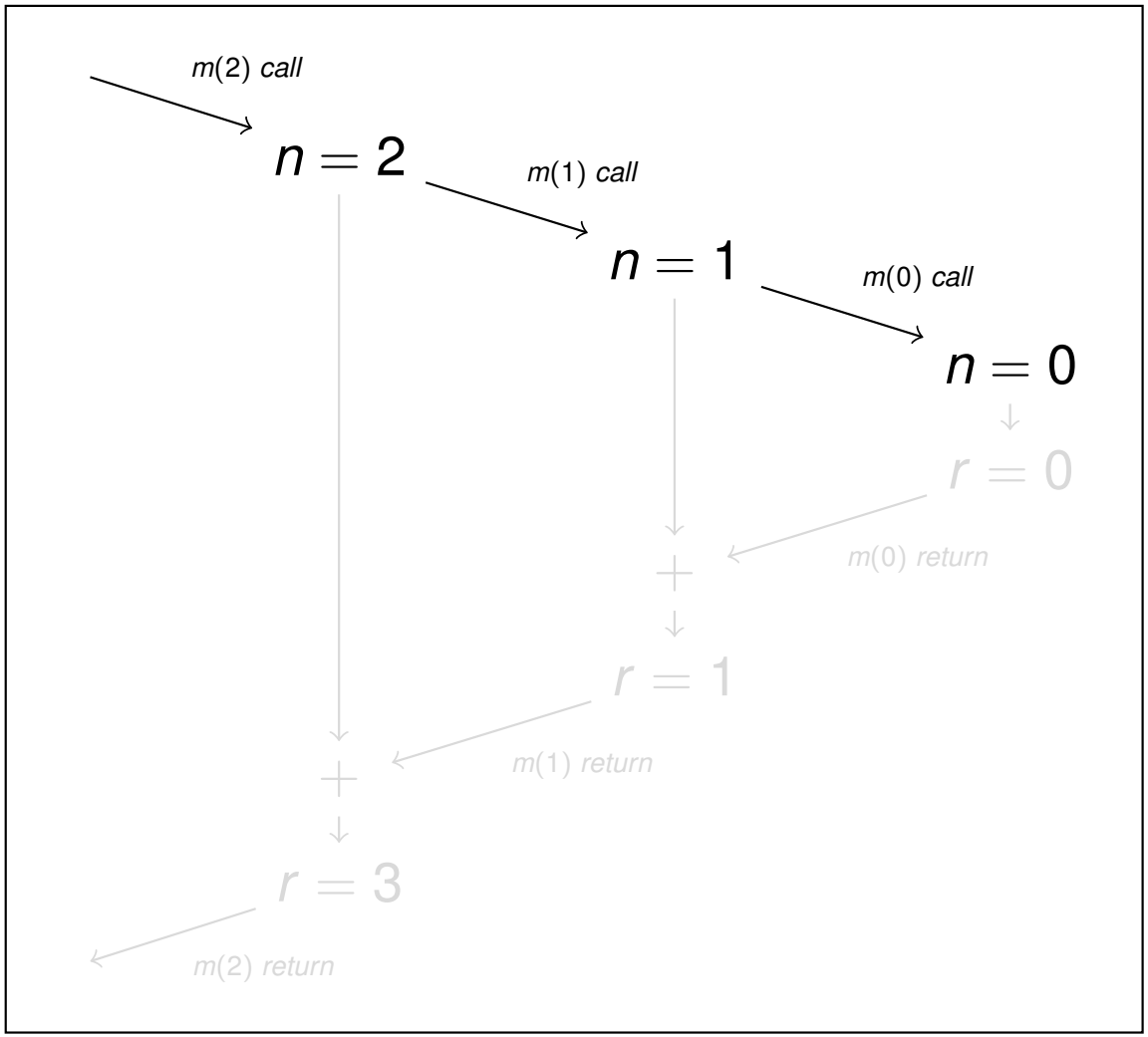
Exemplo: execução de $m(2)$



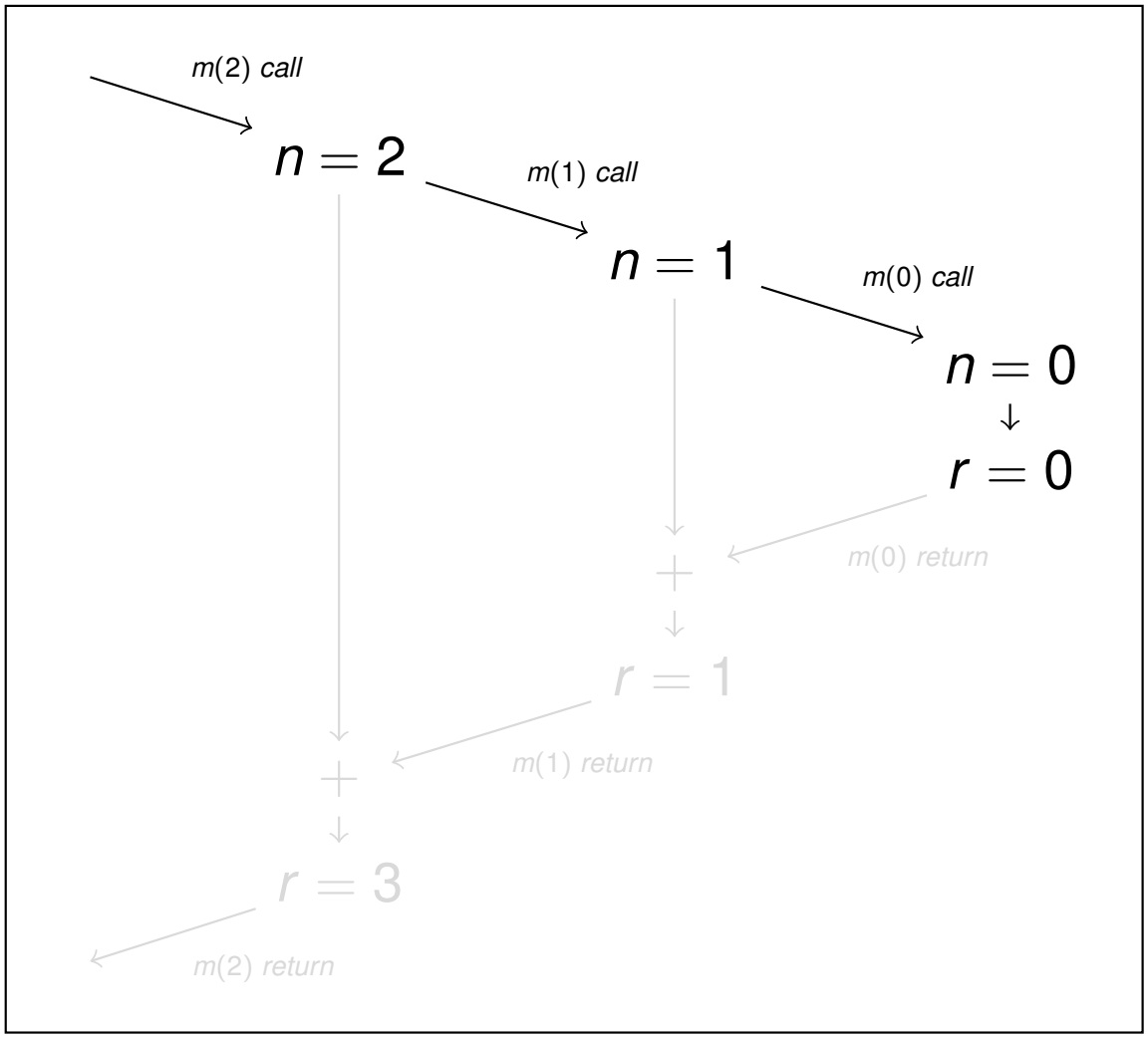
Exemplo: execução de $m(2)$



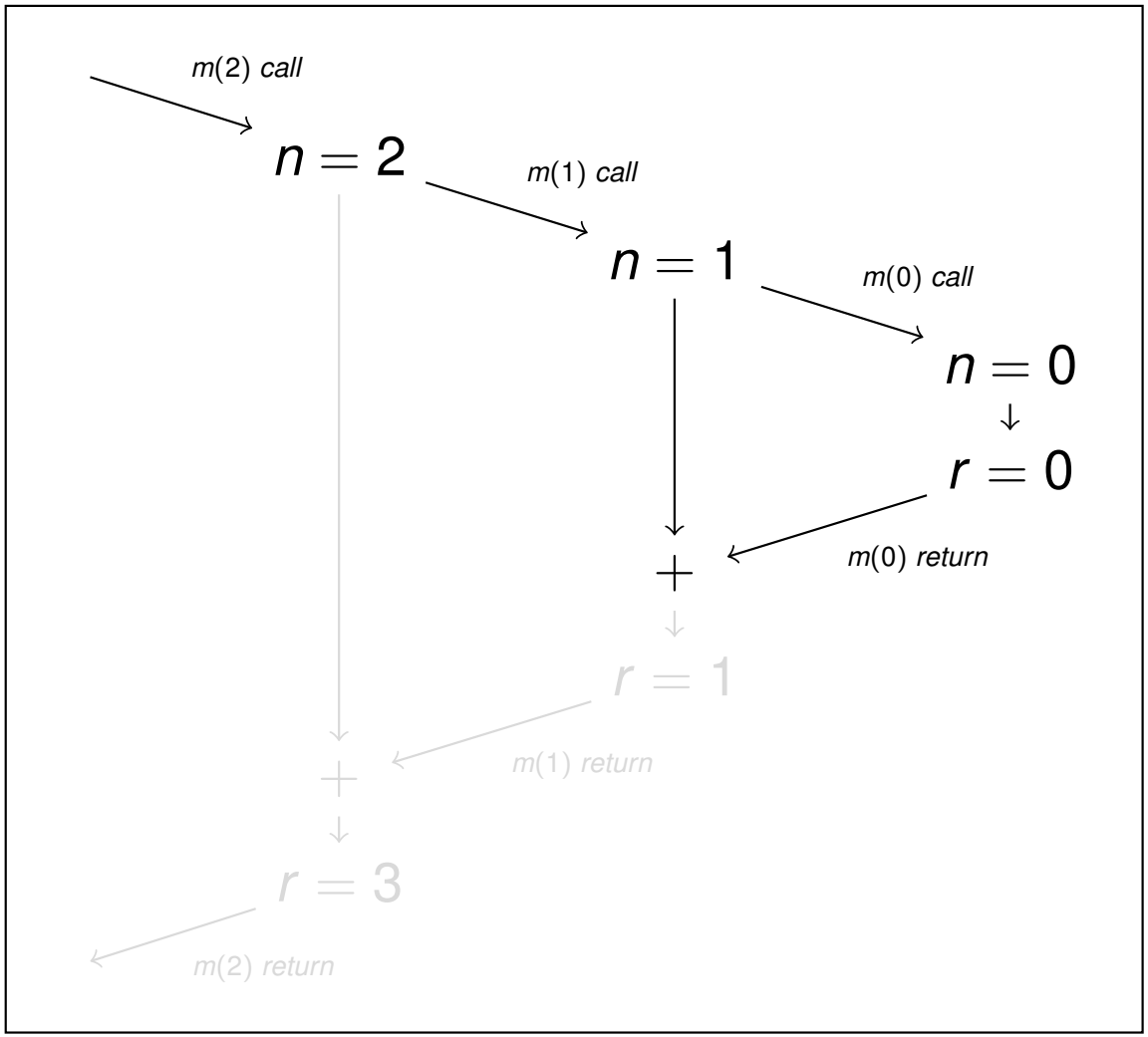
Exemplo: execução de $m(2)$



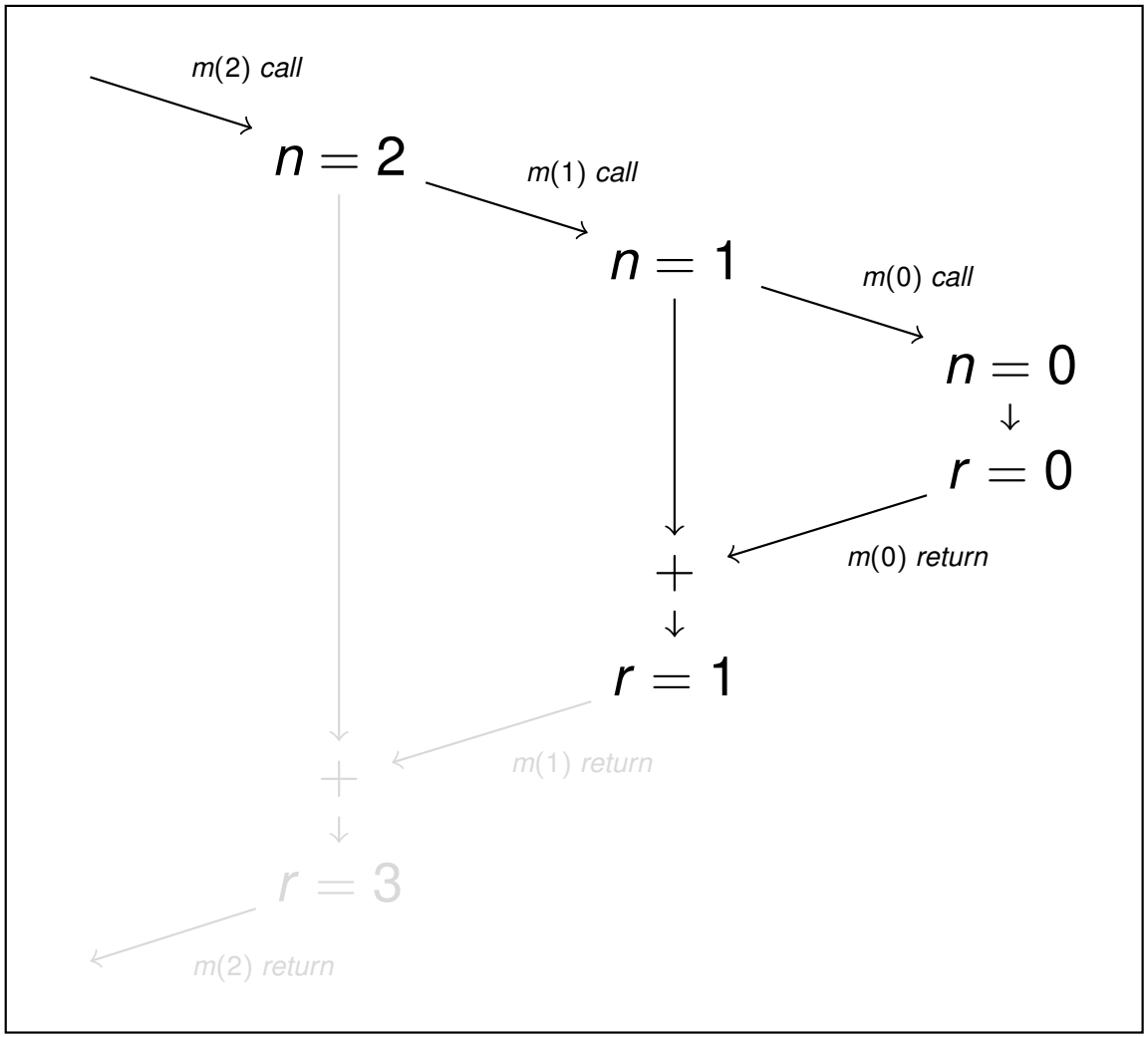
Exemplo: execução de $m(2)$



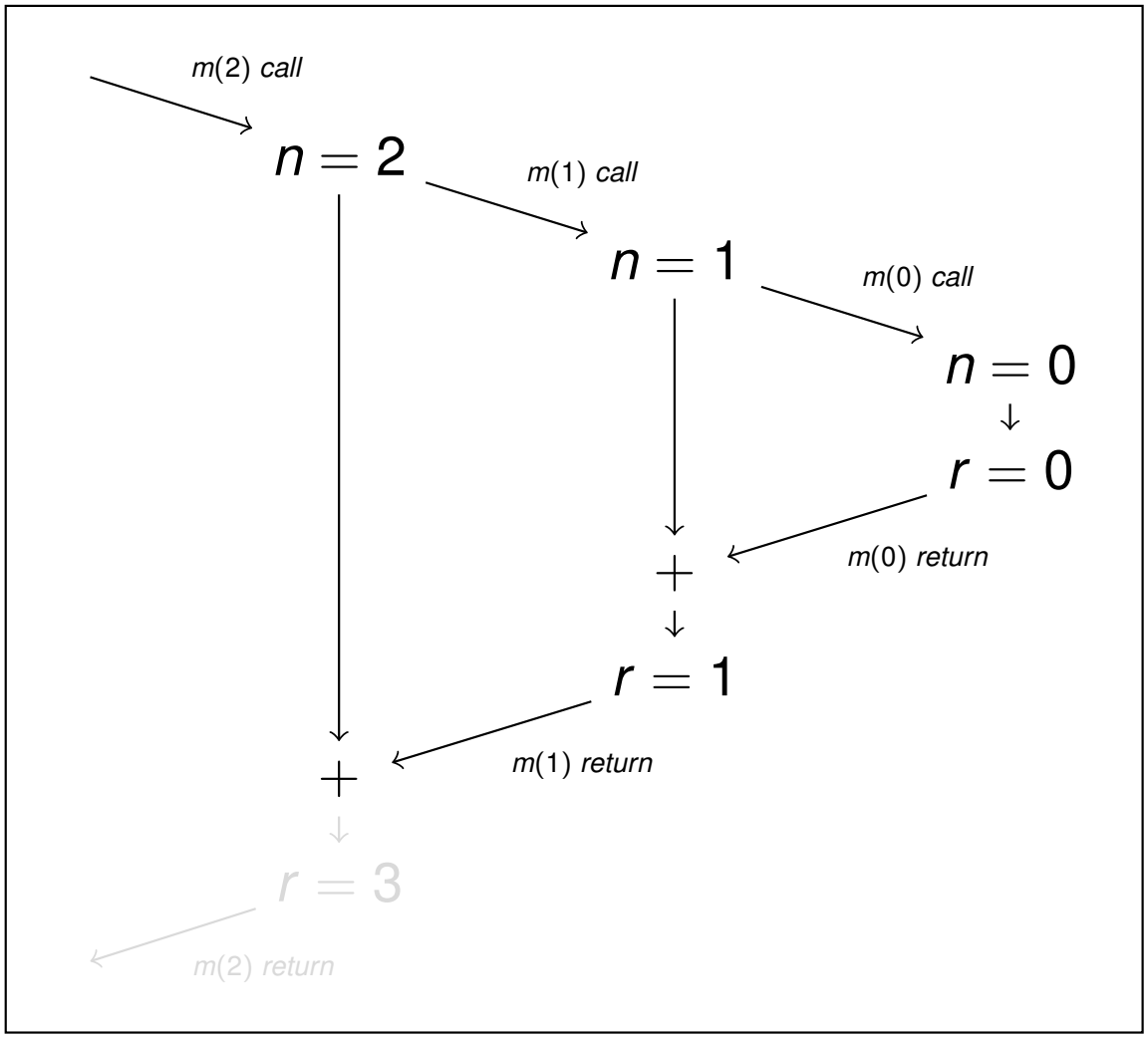
Exemplo: execução de $m(2)$



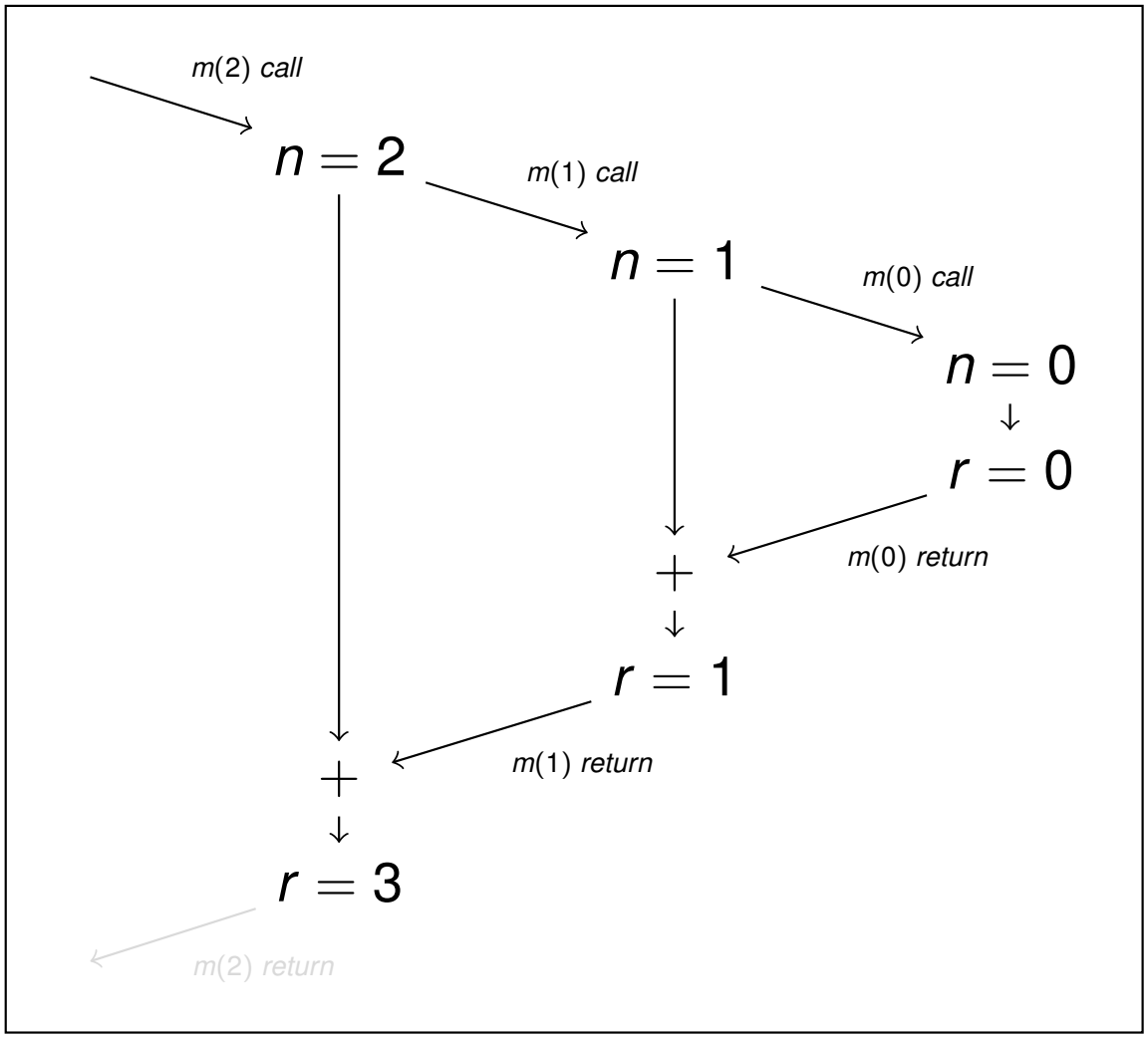
Exemplo: execução de $m(2)$



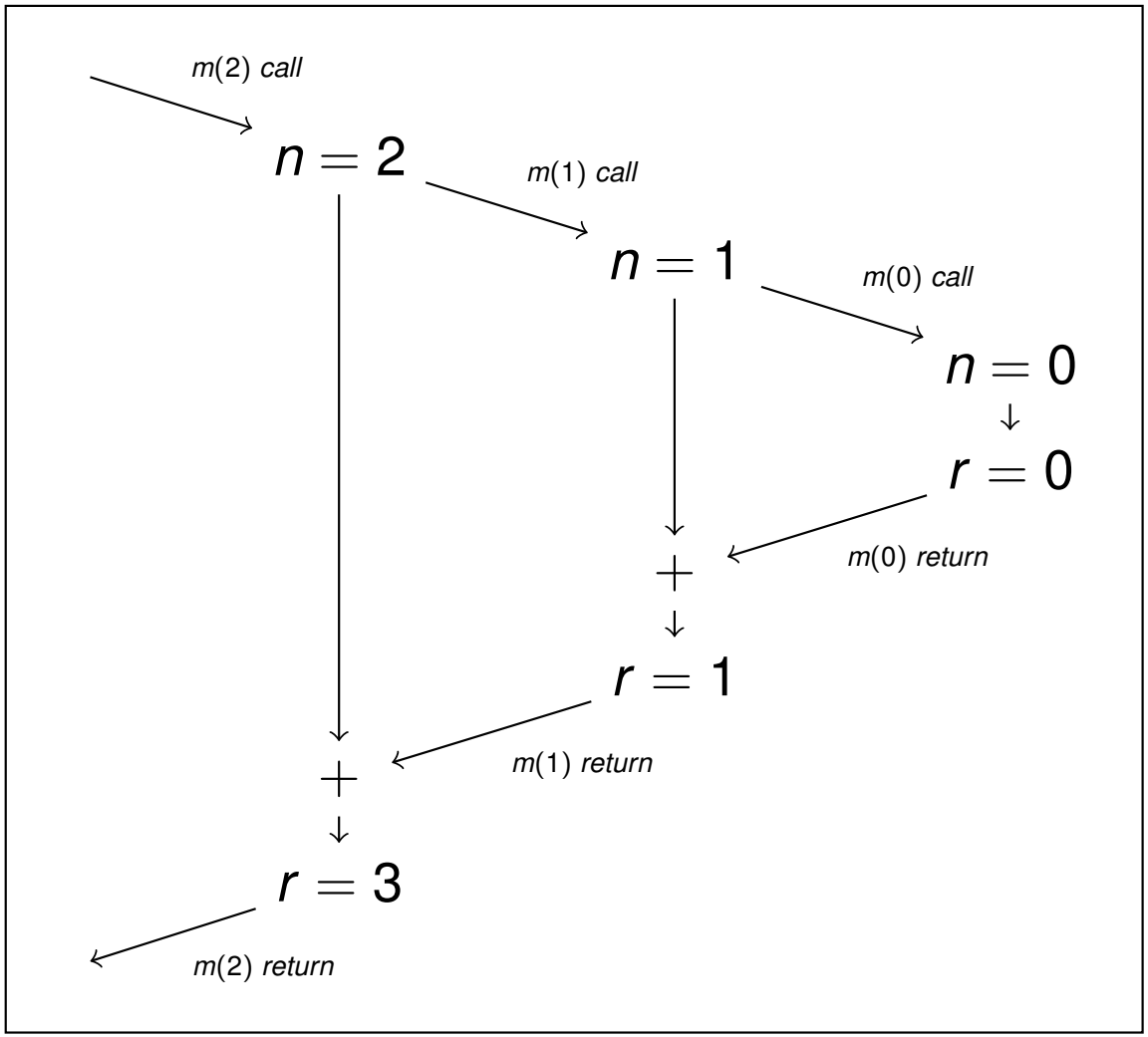
Exemplo: execução de $m(2)$



Exemplo: execução de $m(2)$

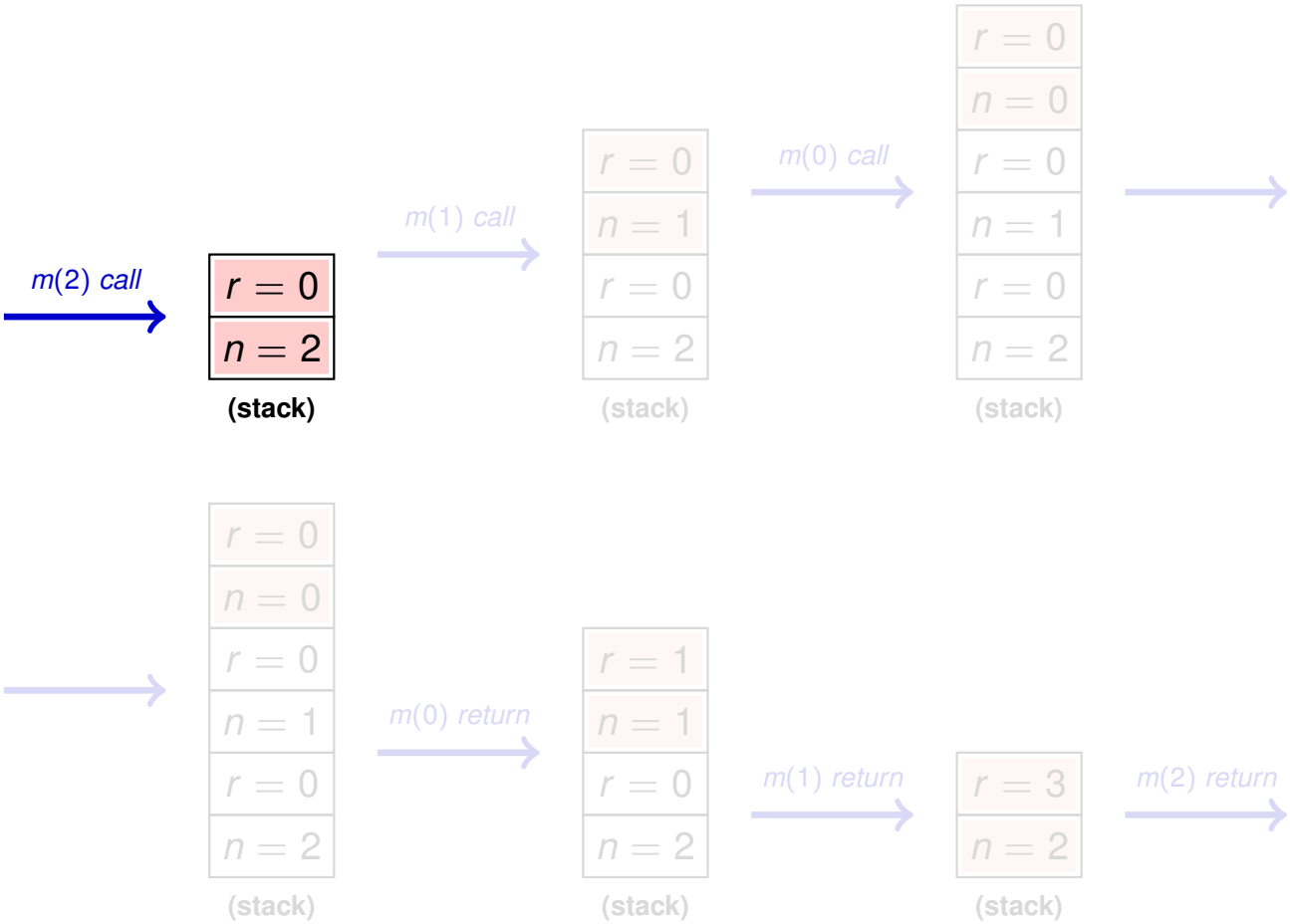


Exemplo: execução de $m(2)$

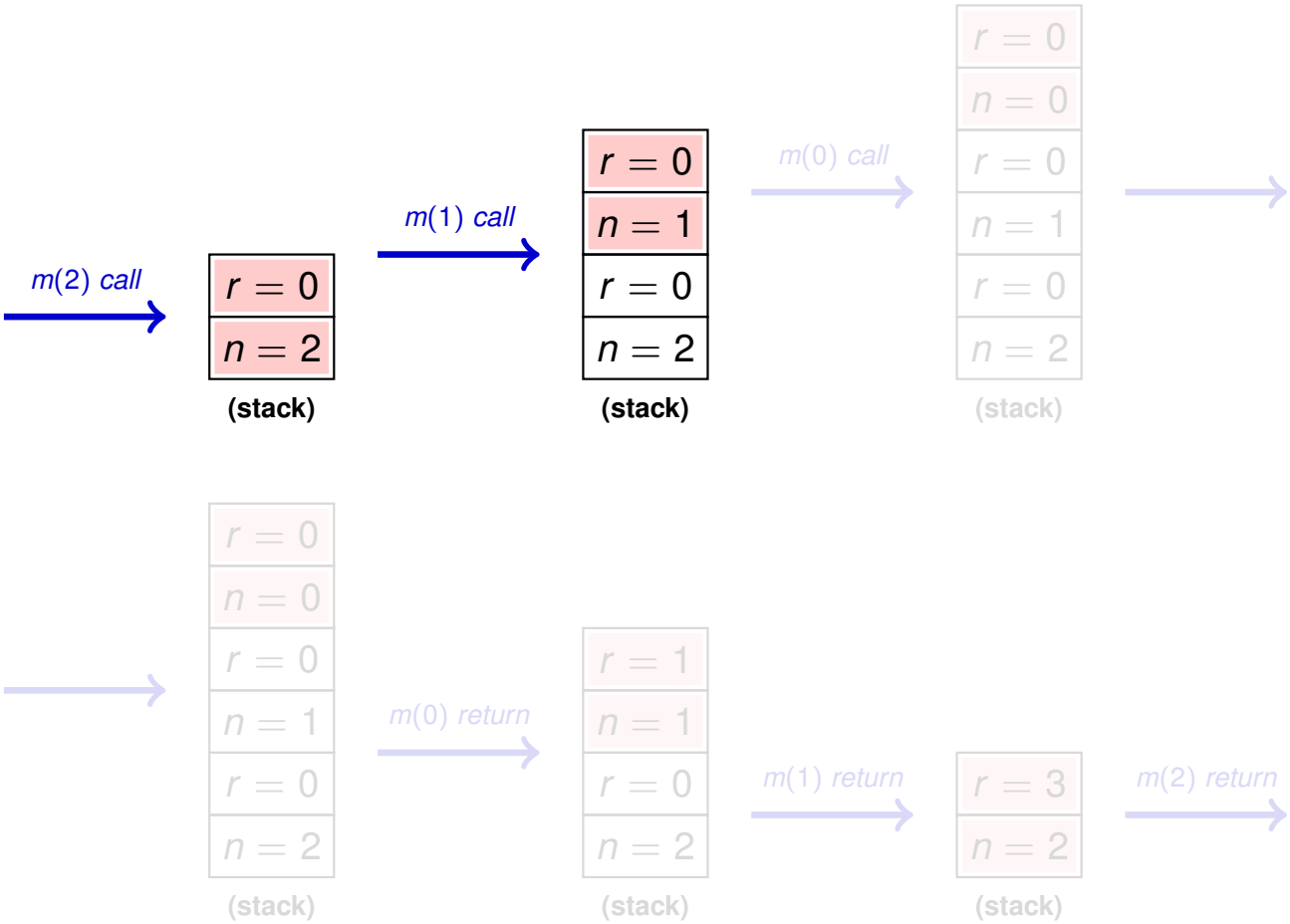


Exemplo: execução de $m(2)$

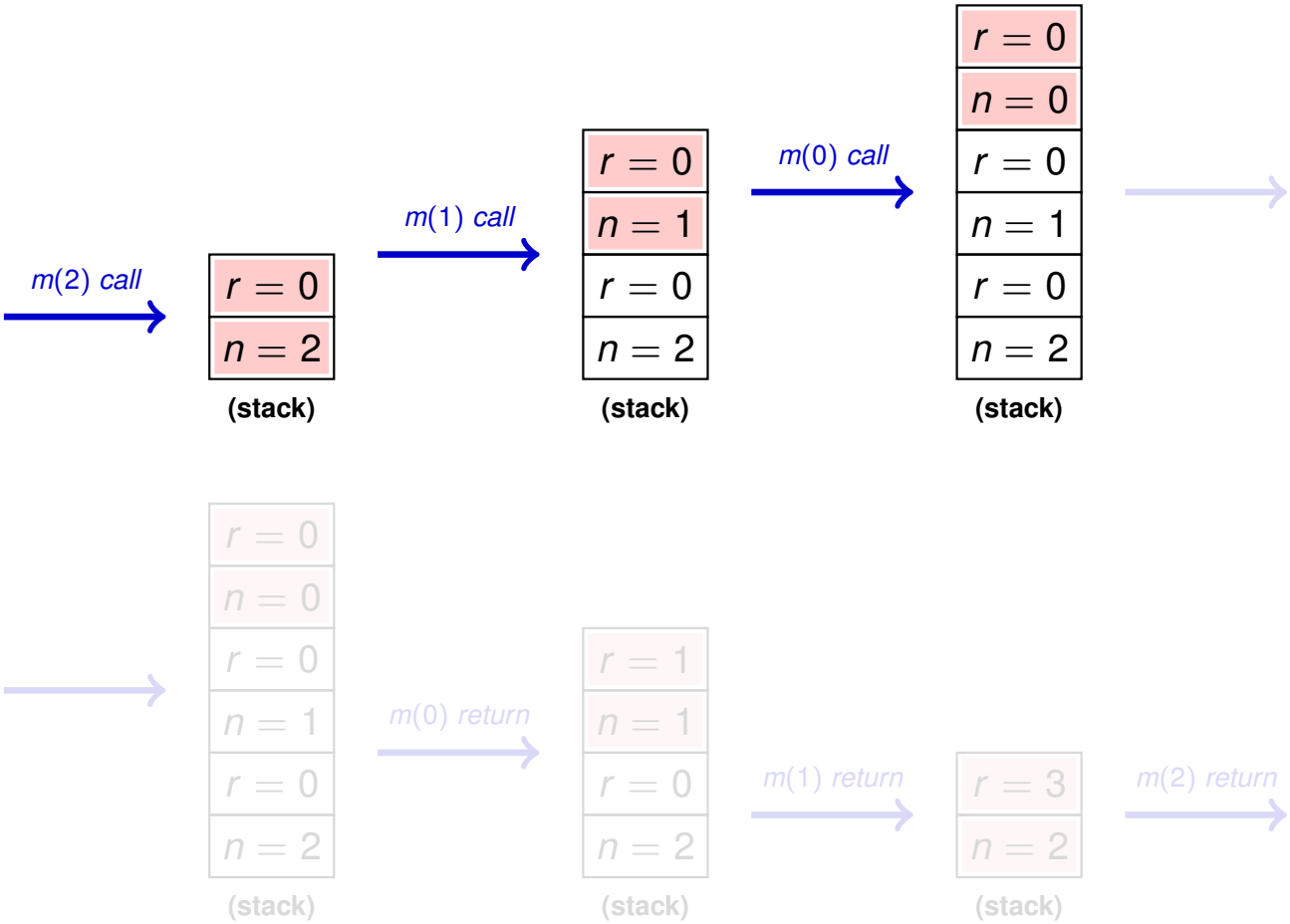
Exemplo: execução de $m(2)$



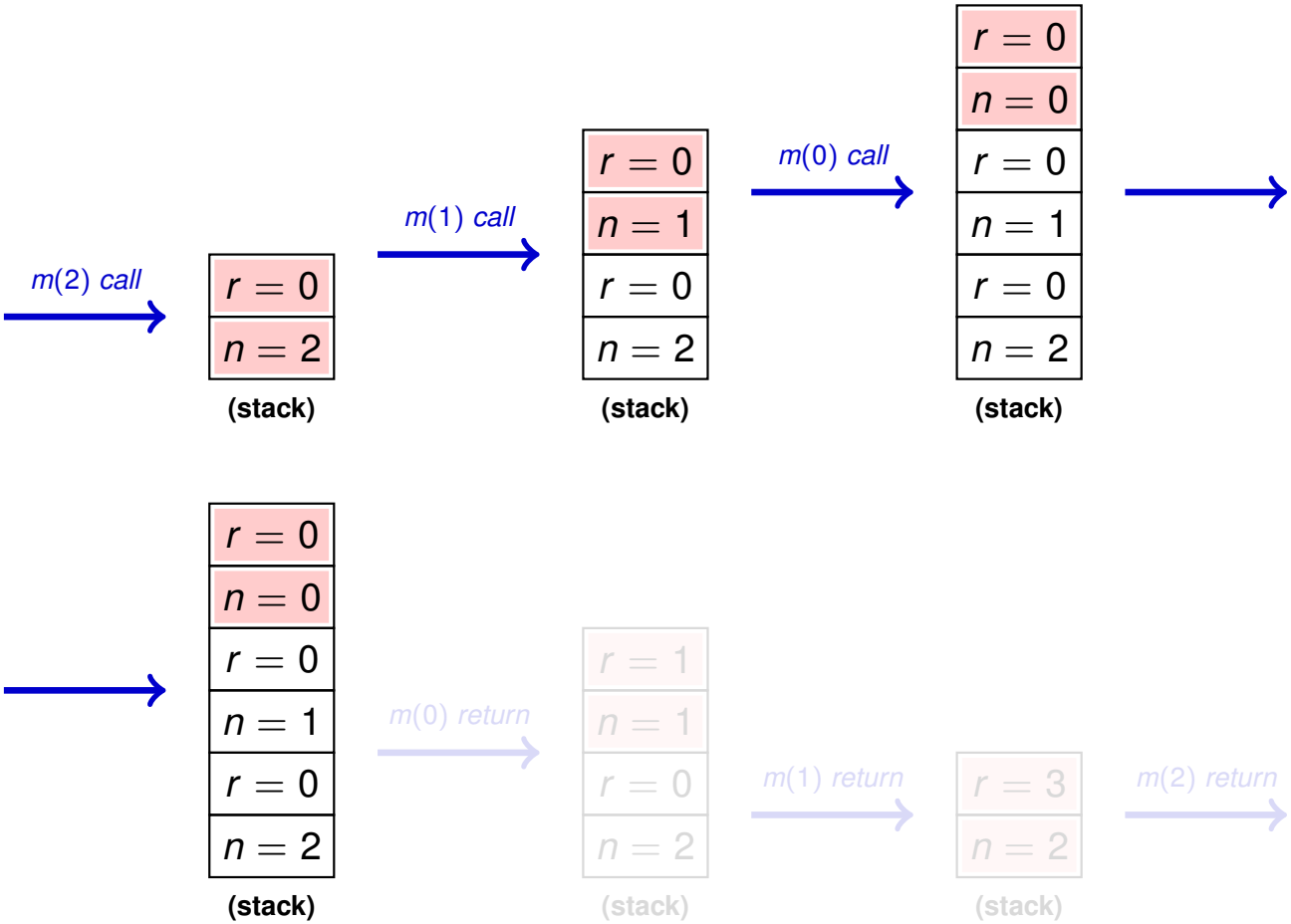
Exemplo: execução de $m(2)$



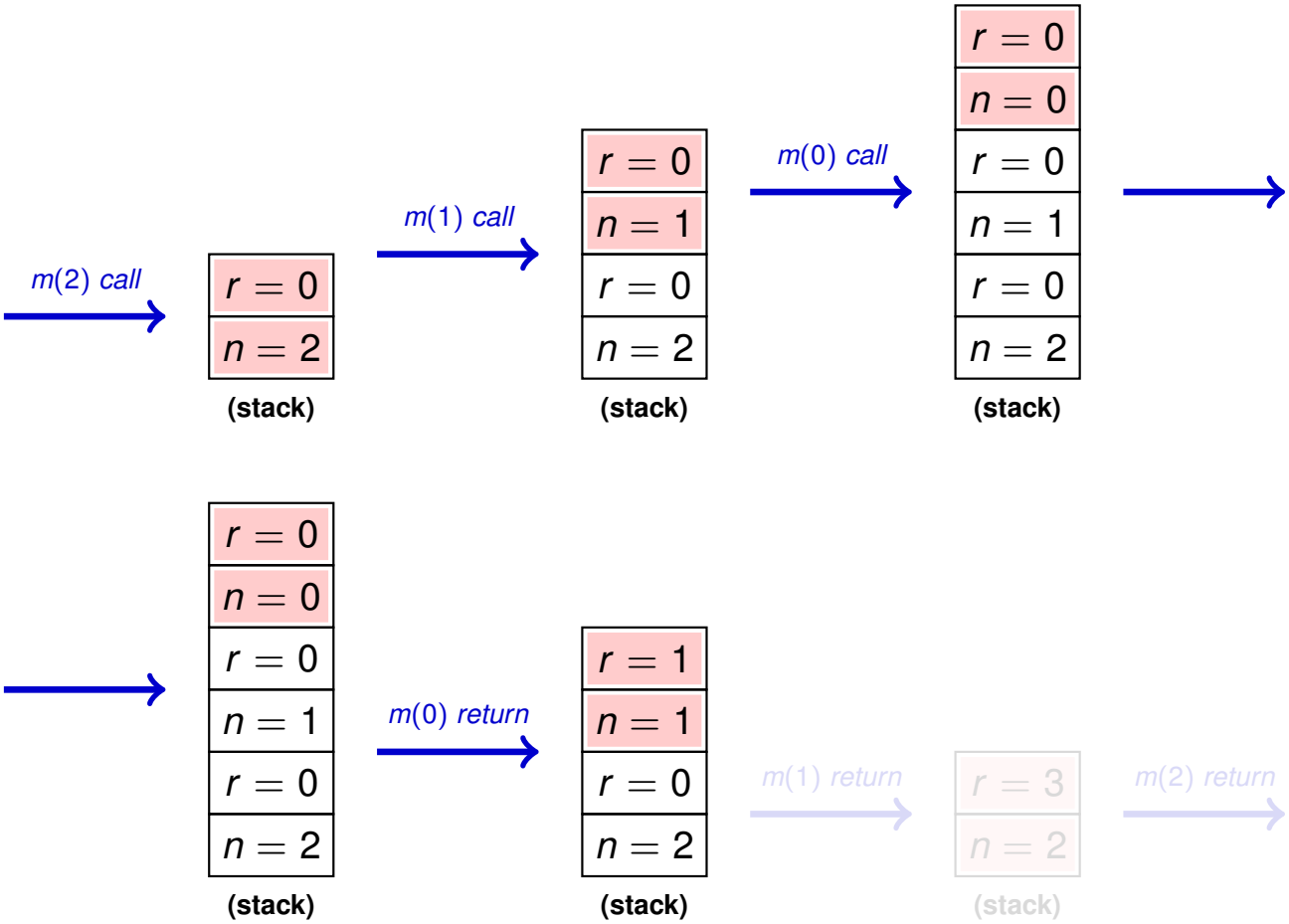
Exemplo: execução de $m(2)$



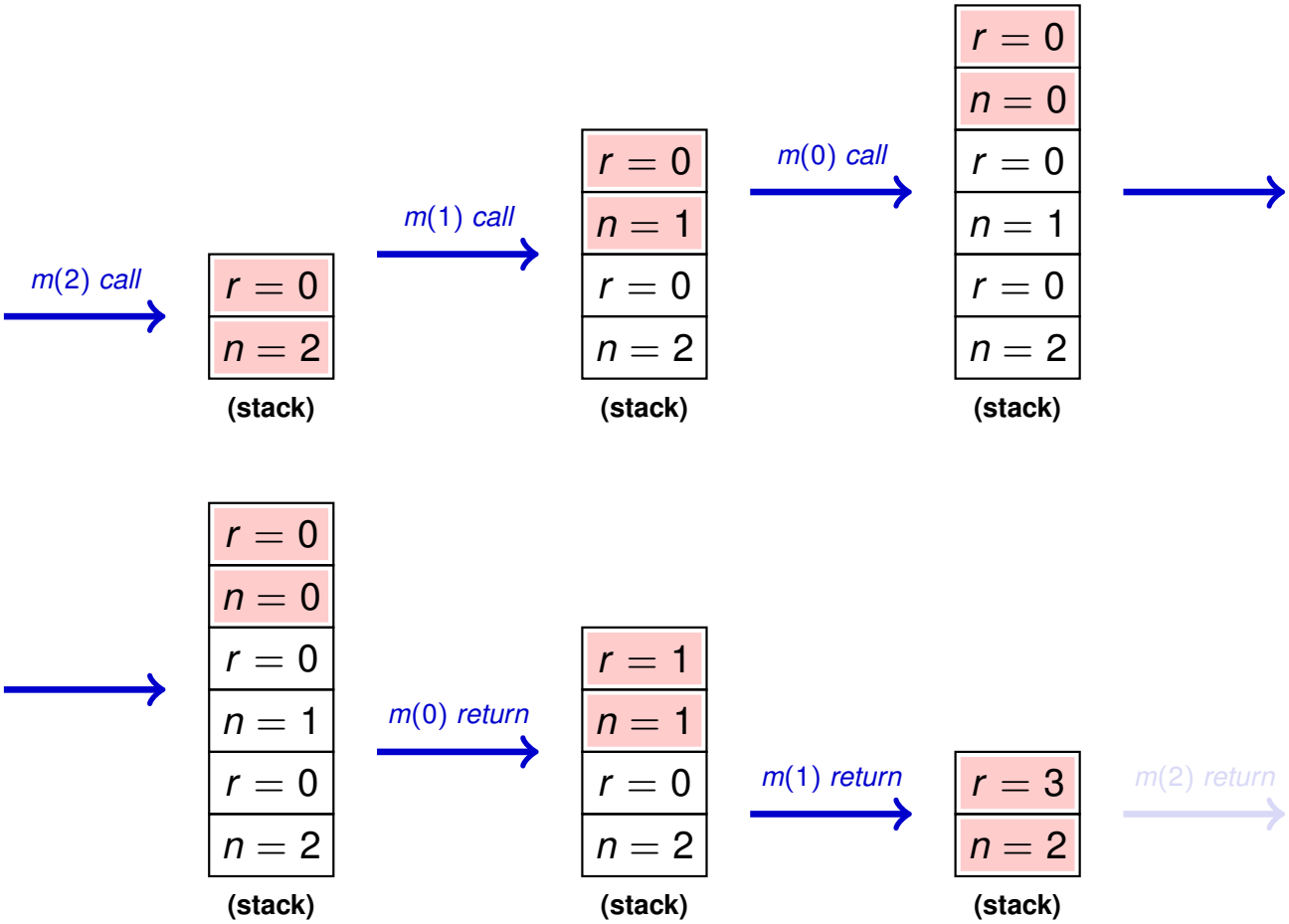
Exemplo: execução de $m(2)$



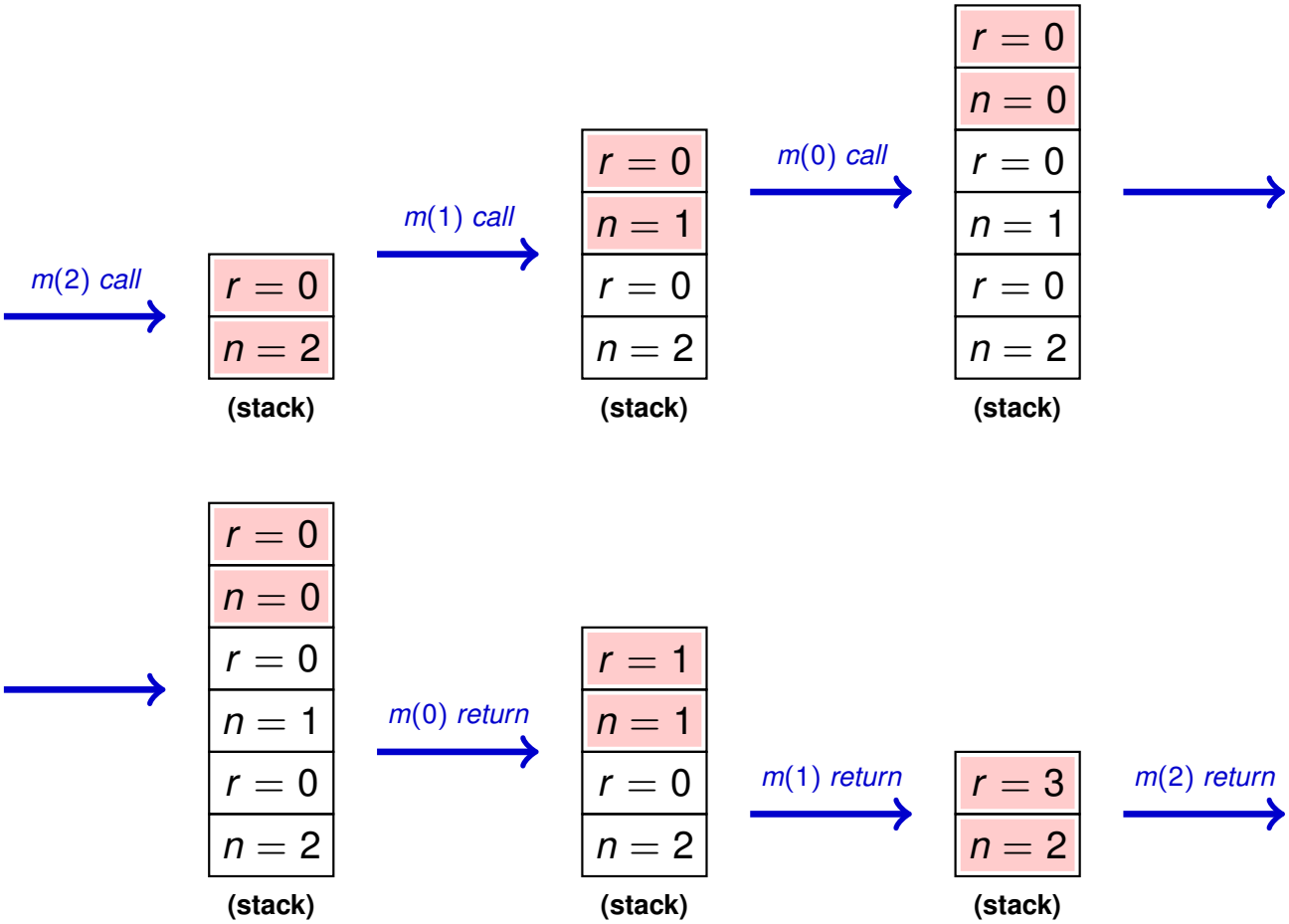
Exemplo: execução de $m(2)$



Exemplo: execução de $m(2)$



Exemplo: execução de $m(2)$



Iteração para recursão

- Como já foi referido, um algoritmo recursivo tem sempre uma versão iterativa e vice-versa.
- Um algoritmo genérico que permite converter um ciclo (estruturado!) para uma função recursiva é o seguinte:

Implementação Iterativa

```
for (INIT; COND; INC) {
   BODY
}
```

Implementação Recursiva

```
INIT
loopEquiv(args)
...

static void loopEquiv(args decl) {
   if (COND) {
      BODY
      INC
      loopEquiv(args);
   }
}
```

- Os argumentos a definir na função recursiva correspondem somente às variáveis utilizadas no ciclo.
- Argumentos ou variáveis locais necessitam de ser passados para a função.

Iteração para recursão

- Como já foi referido, um algoritmo recursivo tem sempre uma versão iterativa e vice-versa.
- Um algoritmo genérico que permite converter um ciclo (estruturado!) para uma função recursiva é o seguinte:

Implementação Iterativa

```
for (INIT; COND; INC) {
   BODY
}
```

Implementação Recursiva

```
INIT
loopEquiv(args)
...

static void loopEquiv(args decl) {
   if (COND) {
      BODY
      INC
      loopEquiv(args);
   }
}
```

- Os argumentos a definir na função recursiva correspondem somente às variáveis utilizadas no ciclo.
- Argumentos ou variáveis locais necessitam de ser passados para a função.

Iteração para recursão

- Como já foi referido, um algoritmo recursivo tem sempre uma versão iterativa e vice-versa.
- Um algoritmo genérico que permite converter um ciclo (estruturado!) para uma função recursiva é o seguinte:

Implementação Iterativa

```
for (INIT; COND; INC) {
   BODY
}
```

Implementação Recursiva

```
INIT
loopEquiv(args)
...

static void loopEquiv(args decl) {
   if (COND) {
      BODY
      INC
      loopEquiv(args);
   }
}
```

- Os argumentos a definir na função recursiva correspondem somente às variáveis utilizadas no ciclo.
- Argumentos ou variáveis locais necessitam de ser passados para a função.

Iteração para recursão

- Como já foi referido, um algoritmo recursivo tem sempre uma versão iterativa e vice-versa.
- Um algoritmo genérico que permite converter um ciclo (estruturado!) para uma função recursiva é o seguinte:

Implementação Iterativa

```
for (INIT; COND; INC) {
   BODY
}
```

Implementação Recursiva

```
INIT
loopEquiv(args)
...

static void loopEquiv(args decl) {
   if (COND) {
      BODY
      INC
      loopEquiv(args);
   }
}
```

- Os argumentos a definir na função recursiva correspondem somente às variáveis utilizadas no ciclo.
- Argumentos ou variáveis locais necessitam de ser passados para a função.

Iteração para recursão

- Como já foi referido, um algoritmo recursivo tem sempre uma versão iterativa e vice-versa.
- Um algoritmo genérico que permite converter um ciclo (estruturado!) para uma função recursiva é o seguinte:

Implementação Iterativa

```
for (INIT; COND; INC) {
   BODY
}
```

Implementação Recursiva

```
INIT
loopEquiv(args)
...

static void loopEquiv(args decl) {
   if (COND) {
      BODY
      INC
      loopEquiv(args);
   }
}
```

- Os argumentos a definir na função recursiva correspondem somente às variáveis utilizadas no ciclo.
- Argumentos ou variáveis locais necessitam de ser passados para a função.

Iteração para recursão

- Como já foi referido, um algoritmo recursivo tem sempre uma versão iterativa e vice-versa.
- Um algoritmo genérico que permite converter um ciclo (estruturado!) para uma função recursiva é o seguinte:

Implementação Iterativa

```
for (INIT; COND; INC) {
   BODY
}
```

Implementação Recursiva

```
INIT
loopEquiv(args)
...

static void loopEquiv(args decl) {
   if (COND) {
      BODY
      INC
      loopEquiv(args);
   }
}
```

- Os argumentos a definir na função recursiva correspondem somente às variáveis utilizadas no ciclo.
- Argumentos ou variáveis locais necessitam de ser passados para a função.

Iteração para recursão

- Como já foi referido, um algoritmo recursivo tem sempre uma versão iterativa e vice-versa.
- Um algoritmo genérico que permite converter um ciclo (estruturado!) para uma função recursiva é o seguinte:

Implementação Iterativa

```
for (INIT; COND; INC) {
   BODY
}
```

Implementação Recursiva

```
INIT
loopEquiv(args)
...

static void loopEquiv(args decl) {
   if (COND) {
      BODY
      INC
      loopEquiv(args);
   }
}
```

- Os argumentos a definir na função recursiva correspondem somente às variáveis utilizadas no ciclo.
- Argumentos ou variáveis locais necessitam de ser passados para a função.

Iteração para recursão

- Como já foi referido, um algoritmo recursivo tem sempre uma versão iterativa e vice-versa.
- Um algoritmo genérico que permite converter um ciclo (estruturado!) para uma função recursiva é o seguinte:

Implementação Iterativa

```
for (INIT; COND; INC) {
   BODY
}
```

Implementação Recursiva

```
INIT
loopEquiv(args)
...

static void loopEquiv(args decl) {
   if (COND) {
      BODY
      INC
      loopEquiv(args);
   }
}
```

- Os argumentos a definir na função recursiva correspondem somente às variáveis utilizadas no ciclo.
- Argumentos ou variáveis locais necessitam de ser passados para a função.

Iteração para recursão: exemplo

Implementação Iterativa

```
// int[] arr
for(int i=0; i<arr.length; i++)
   out.println(arr[i]);
```

Implementação Recursiva

```
int i = 0;
loopEquiv(arr, i);
...

static void loopEquiv(int[] arr,int i) {
   if (i < arr.length) {
      out.println(arr[i]);
      i++;
      loopEquiv(arr, i);
   }
}
```

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

Iteração para recursão: exemplo

Implementação Iterativa

```
// int[] arr
for(int i=0; i<arr.length; i++)
   out.println(arr[i]);
```

Implementação Recursiva

```
int i = 0;
loopEquiv(arr, i);
...

static void loopEquiv(int[] arr,int i) {
   if (i < arr.length) {
      out.println(arr[i]);
      i++;
      loopEquiv(arr, i);
   }
}
```

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

Iteração para recursão: exemplo

Implementação Iterativa

```
// int[] arr
for(int i=0; i<arr.length; i++)
   out.println(arr[i]);
```

Implementação Recursiva

```
int i = 0;
loopEquiv(arr, i);
...

static void loopEquiv(int[] arr,int i) {
   if (i < arr.length) {
      out.println(arr[i]);
      i++;
      loopEquiv(arr, i);
   }
}
```

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

Iteração para recursão: exemplo

Implementação Iterativa

```
// int[] arr
for(int i=0; i<arr.length; i++)
   out.println(arr[i]);
```

Implementação Recursiva

```
int i = 0;
loopEquiv(arr, i);
...

static void loopEquiv(int[] arr,int i) {
   if (i < arr.length) {
      out.println(arr[i]);
      i++;
      loopEquiv(arr, i);
   }
}
```

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

Recursão para iteração

- A conversão de algoritmos recursivos para ciclos (estruturados) é, em geral, bem mais complexa do que a transformação inversa.
- Um algoritmo geral para fazer essa conversão faz uso de uma *pilha* para armazenar os contextos de execução da função recursiva (composto pelos argumentos, variáveis locais e resultado da função) e implementar as chamadas das funções por instruções (não estruturadas) do tipo *salto* (*goto*).
- No entanto, o preço a pagar pode ser bem elevado em termos de legibilidade e até mesmo de correcção do algoritmo.
- Alguns tipos em particular de recursividade, como é o caso da recursão do tipo *cauda* (*tail recursion*) prestam-se a optimizações interessantes (já que podemos prescindir do armazenamento de algum contexto).
- Esta matéria, no entanto, sai fora do âmbito desta disciplina pelo que não a vamos abordar.

Recursão para iteração

- A conversão de algoritmos recursivos para ciclos (estruturados) é, em geral, bem mais complexa do que a transformação inversa.
- Um algoritmo geral para fazer essa conversão faz uso de uma *pilha* para armazenar os contextos de execução da função recursiva (composto pelos argumentos, variáveis locais e resultado da função) e implementar as chamadas das funções por instruções (não estruturadas) do tipo *salto* (*goto*).
- No entanto, o preço a pagar pode ser bem elevado em termos de legibilidade e até mesmo de correcção do algoritmo.
- Alguns tipos em particular de recursividade, como é o caso da recursão do tipo *cauda* (*tail recursion*) prestam-se a optimizações interessantes (já que podemos prescindir do armazenamento de algum contexto).
- Esta matéria, no entanto, sai fora do âmbito desta disciplina pelo que não a vamos abordar.

Recursão para iteração

- A conversão de algoritmos recursivos para ciclos (estruturados) é, em geral, bem mais complexa do que a transformação inversa.
- Um algoritmo geral para fazer essa conversão faz uso de uma *pilha* para armazenar os contextos de execução da função recursiva (composto pelos argumentos, variáveis locais e resultado da função) e implementar as chamadas das funções por instruções (não estruturadas) do tipo *salto* (*goto*).
- No entanto, o preço a pagar pode ser bem elevado em termos de legibilidade e até mesmo de correcção do algoritmo.
- Alguns tipos em particular de recursividade, como é o caso da recursão do tipo *cauda* (*tail recursion*) prestam-se a optimizações interessantes (já que podemos prescindir do armazenamento de algum contexto).
- Esta matéria, no entanto, sai fora do âmbito desta disciplina pelo que não a vamos abordar.

Recursão para iteração

- A conversão de algoritmos recursivos para ciclos (estruturados) é, em geral, bem mais complexa do que a transformação inversa.
- Um algoritmo geral para fazer essa conversão faz uso de uma *pilha* para armazenar os contextos de execução da função recursiva (composto pelos argumentos, variáveis locais e resultado da função) e implementar as chamadas das funções por instruções (não estruturadas) do tipo *salto* (*goto*).
- No entanto, o preço a pagar pode ser bem elevado em termos de legibilidade e até mesmo de correcção do algoritmo.
- Alguns tipos em particular de recursividade, como é o caso da recursão do tipo *cauda* (*tail recursion*) prestam-se a optimizações interessantes (já que podemos prescindir do armazenamento de algum contexto).
- Esta matéria, no entanto, sai fora do âmbito desta disciplina pelo que não a vamos abordar.

Recursão para iteração

- A conversão de algoritmos recursivos para ciclos (estruturados) é, em geral, bem mais complexa do que a transformação inversa.
- Um algoritmo geral para fazer essa conversão faz uso de uma *pilha* para armazenar os contextos de execução da função recursiva (composto pelos argumentos, variáveis locais e resultado da função) e implementar as chamadas das funções por instruções (não estruturadas) do tipo *salto* (*goto*).
- No entanto, o preço a pagar pode ser bem elevado em termos de legibilidade e até mesmo de correcção do algoritmo.
- Alguns tipos em particular de recursividade, como é o caso da recursão do tipo *cauda* (*tail recursion*) prestam-se a optimizações interessantes (já que podemos prescindir do armazenamento de algum contexto).
- Esta matéria, no entanto, sai fora do âmbito desta disciplina pelo que não a vamos abordar.

Recursão para iteração

- A conversão de algoritmos recursivos para ciclos (estruturados) é, em geral, bem mais complexa do que a transformação inversa.
- Um algoritmo geral para fazer essa conversão faz uso de uma *pilha* para armazenar os contextos de execução da função recursiva (composto pelos argumentos, variáveis locais e resultado da função) e implementar as chamadas das funções por instruções (não estruturadas) do tipo *salto* (*goto*).
- No entanto, o preço a pagar pode ser bem elevado em termos de legibilidade e até mesmo de correcção do algoritmo.
- Alguns tipos em particular de recursividade, como é o caso da recursão do tipo *cauda* (*tail recursion*) prestam-se a optimizações interessantes (já que podemos prescindir do armazenamento de algum contexto).
- Esta matéria, no entanto, sai fora do âmbito desta disciplina pelo que não a vamos abordar.

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.
  - As invocações recursivas são assim imediatamente convertíveis em acessos ao array.

Implementação Recursiva

```
static int factorial(int n) {
   assert n >= 0;

   int res = 1;

   if (n > 1)
      res = n * factorial(n-1);

   return res;
}
```

Implementação Iterativa (com array)

```
static int factorial(int n) {
   assert n >= 0;

   int[] arr = new int[n+1];
   for(int i = 0; i <= n; i++) {
      if (i < 2) // casos limite
         arr[i] = 1;
      else
         arr[i] = i * arr[i-1];
   }

   return arr[n];
}
```

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.
  - As invocações recursivas são assim imediatamente convertíveis em acessos ao array.

Implementação Recursiva

```
static int factorial(int n) {
   assert n >= 0;

   int res = 1;

   if (n > 1)
      res = n * factorial(n-1);

   return res;
}
```

Implementação Iterativa (com array)

```
static int factorial(int n) {
   assert n >= 0;

   int[] arr = new int[n+1];
   for(int i = 0; i <= n; i++) {
      if (i < 2) // casos limite
         arr[i] = 1;
      else
         arr[i] = i * arr[i-1];
   }

   return arr[n];
}
```

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.
  - As invocações recursivas são assim imediatamente convertíveis em acessos ao array.

Implementação Recursiva

```
static int factorial(int n) {
   assert n >= 0;

   int res = 1;

   if (n > 1)
      res = n * factorial(n-1);

   return res;
}
```

Implementação Iterativa (com array)

```
static int factorial(int n) {
   assert n >= 0;

   int[] arr = new int[n+1];
   for(int i = 0; i <= n; i++) {
      if (i < 2) // casos limite
         arr[i] = 1;
      else
         arr[i] = i * arr[i-1];
   }

   return arr[n];
}
```

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.
  - As invocações recursivas são assim imediatamente convertíveis em acessos ao array.

Implementação Recursiva

```
static int factorial(int n) {
   assert n >= 0;

   int res = 1;

   if (n > 1)
      res = n * factorial(n-1);

   return res;
}
```

Implementação Iterativa (com array)

```
static int factorial(int n) {
   assert n >= 0;

   int[] arr = new int[n+1];
   for(int i = 0; i <= n; i++) {
      if (i < 2) // casos limite
         arr[i] = 1;
      else
         arr[i] = i * arr[i-1];
   }

   return arr[n];
}
```

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.
  - As invocações recursivas são assim imediatamente convertíveis em acessos ao array.

Implementação Recursiva

```
static int factorial(int n) {
   assert n >= 0;

   int res = 1;

   if (n > 1)
      res = n * factorial(n-1);

   return res;
}
```

Implementação Iterativa (com array)

```
static int factorial(int n) {
   assert n >= 0;

   int[] arr = new int[n+1];
   for(int i = 0; i <= n; i++) {
      if (i < 2) // casos limite
         arr[i] = 1;
      else
         arr[i] = i * arr[i-1];
   }

   return arr[n];
}
```

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.
  - As invocações recursivas são assim imediatamente convertíveis em acessos ao array.

Implementação Recursiva

```
static int factorial(int n) {
   assert n >= 0;

   int res = 1;

   if (n > 1)
      res = n * factorial(n-1);

   return res;
}
```

Implementação Iterativa (com array)

```
static int factorial(int n) {
   assert n >= 0;

   int[] arr = new int[n+1];
   for(int i = 0; i <= n; i++) {
      if (i < 2) // casos limite
         arr[i] = 1;
      else
         arr[i] = i * arr[i-1];
   }

   return arr[n];
}
```

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.
  - As invocações recursivas são assim imediatamente convertíveis em acessos ao array.

Implementação Recursiva

```
static int factorial(int n) {
   assert n >= 0;

   int res = 1;

   if (n > 1)
      res = n * factorial(n-1);

   return res;
}
```

Implementação Iterativa (com array)

```
static int factorial(int n) {
   assert n >= 0;

   int[] arr = new int[n+1];
   for(int i = 0; i <= n; i++) {
      if (i < 2) // casos limite
         arr[i] = 1;
      else
         arr[i] = i * arr[i-1];
   }

   return arr[n];
}
```

Procura de um elemento numa lista: 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 iterativa equivalente.

Implementação Iterativa

```
public class LinkedList<E> {
  ...
  public boolean contains(E e) {
     boolean found = false;
     Node<E> n = first;
     while (n!=null && !found) {
        if (n.elem.equals(e))
           found = true;
        else n = n.next;
     }
     return found;
  }
  ...
}
```

Implementação Recursiva

```
public class LinkedList<E> {
  ...
  public boolean contains(E e) {
     return contains(first,e);
  }
  private boolean contains(Node<E> n, E e) {
     if (n == null) return false;
     if (n.elem.equals(e)) return true;
     return contains(n.next,e);
  }
  ...
}
```

Procura de um elemento numa lista: 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 iterativa equivalente.

Implementação Iterativa

```
public class LinkedList<E> {
  ...
  public boolean contains(E e) {
     boolean found = false;
     Node<E> n = first;
     while (n!=null && !found) {
        if (n.elem.equals(e))
           found = true;
        else n = n.next;
     }
     return found;
  }
  ...
}
```

Implementação Recursiva

```
public class LinkedList<E> {
  ...
  public boolean contains(E e) {
     return contains(first,e);
  }
  private boolean contains(Node<E> n, E e) {
     if (n == null) return false;
     if (n.elem.equals(e)) return true;
     return contains(n.next,e);
  }
  ...
}
```

Procura de um elemento numa lista: 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 iterativa equivalente.

Implementação Iterativa

```
public class LinkedList<E> {
  ...
  public boolean contains(E e) {
     boolean found = false;
     Node<E> n = first;
     while (n!=null && !found) {
        if (n.elem.equals(e))
           found = true;
        else n = n.next;
     }
     return found;
  }
  ...
}
```

Implementação Recursiva

```
public class LinkedList<E> {
  ...
  public boolean contains(E e) {
     return contains(first,e);
  }
  private boolean contains(Node<E> n, E e) {
     if (n == null) return false;
     if (n.elem.equals(e)) return true;
     return contains(n.next,e);
  }
  ...
}
```

Procura de um elemento numa lista: 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 iterativa equivalente.

Implementação Iterativa

```
public class LinkedList<E> {
  ...
  public boolean contains(E e) {
     boolean found = false;
     Node<E> n = first;
     while (n!=null && !found) {
        if (n.elem.equals(e))
           found = true;
        else n = n.next;
     }
     return found;
  }
  ...
}
```

Implementação Recursiva

```
public class LinkedList<E> {
  ...
  public boolean contains(E e) {
     return contains(first,e);
  }
  private boolean contains(Node<E> n, E e) {
     if (n == null) return false;
     if (n.elem.equals(e)) return true;
     return contains(n.next,e);
  }
  ...
}
```

Procura de um elemento numa lista: 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 iterativa equivalente.

Implementação Iterativa

```
public class LinkedList<E> {
  ...
  public boolean contains(E e) {
     boolean found = false;
     Node<E> n = first;
     while (n!=null && !found) {
        if (n.elem.equals(e))
           found = true;
        else n = n.next;
     }
     return found;
  }
  ...
}
```

Implementação Recursiva

```
public class LinkedList<E> {
  ...
  public boolean contains(E e) {
     return contains(first,e);
  }
  private boolean contains(Node<E> n, E e) {
     if (n == null) return false;
     if (n.elem.equals(e)) return true;
     return contains(n.next,e);
  }
  ...
}
```

Procura de um elemento numa lista: 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 iterativa equivalente.

Implementação Iterativa

```
public class LinkedList<E> {
  ...
  public boolean contains(E e) {
     boolean found = false;
     Node<E> n = first;
     while (n!=null && !found) {
        if (n.elem.equals(e))
           found = true;
        else n = n.next;
     }
     return found;
  }
  ...
}
```

Implementação Recursiva

```
public class LinkedList<E> {
  ...
  public boolean contains(E e) {
     return contains(first,e);
  }
  private boolean contains(Node<E> n, E e) {
     if (n == null) return false;
     if (n.elem.equals(e)) return true;
     return contains(n.next,e);
  }
  ...
}
```

Um padrão que se repete ...

- Muitas funções sobre listas fazem um percurso da lista.
- Esse percurso 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,e);
   }
   private ... xpto(Node<E> n, ...) {
      if (n == null) return ...;
      ...
      ... xpto(n.next,...);
      return ...
   }
   ...
}
```

Um padrão que se repete ...

- Muitas funções sobre listas fazem um percurso da lista.
- Esse percurso 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,e);
  }
  private ... xpto(Node<E> n, ...) {
     if (n == null) return ...;
     ...
     ... xpto(n.next,...);
     return ...
  }
  ...
}
```

Um padrão que se repete ...

- Muitas funções sobre listas fazem um percurso da lista.
- Esse percurso 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,e);
  }
  private ... xpto(Node<E> n, ...) {
     if (n == null) return ...;
     ...
     ... xpto(n.next,...);
     return ...
  }
  ...
}
```

Um padrão que se repete ...

- Muitas funções sobre listas fazem um percurso da lista.
- Esse percurso 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,e);
  }
  private ... xpto(Node<E> n, ...) {
     if (n == null) return ...;
     ...
     ... xpto(n.next,...);
     return ...
  }
  ...
}
```

Um padrão que se repete ...

- Muitas funções sobre listas fazem um percurso da lista.
- Esse percurso 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,e);
  }
  private ... xpto(Node<E> n, ...) {
     if (n == null) return ...;
     ...
     ... xpto(n.next,...);
     return ...
  }
  ...
}
```

Um padrão que se repete ...

- Muitas funções sobre listas fazem um percurso da lista.
- Esse percurso 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,e);
  }
  private ... xpto(Node<E> n, ...) {
     if (n == null) return ...;
     ...
     ... xpto(n.next,...);
     return ...
  }
  ...
}
```

Procura de um elemento num vector: recursão e iteração

- Como faríamos o mesmo 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

```
public static
boolean contains(E[] v,E e) {

   int i=0;
   boolean found = false;
   while (i!=v.length && !found) {
      if (v[i].equals(e))
         found = true;
      else i = i+1; // ou: i++;
   }
   return found;
}
```

Implementação Recursiva

```
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==v.length) return false;
  if (v[i].equals(e)) return true;
  return contains(v,e,i+1);
}
```

Procura de um elemento num vector: recursão e iteração

- Como faríamos o mesmo 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

```
public static
boolean contains(E[] v,E e) {

   int i=0;
   boolean found = false;
   while (i!=v.length && !found) {
      if (v[i].equals(e))
         found = true;
      else i = i+1; // ou: i++;
   }
   return found;
}
```

Implementação Recursiva

```
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==v.length) return false;
  if (v[i].equals(e)) return true;
  return contains(v,e,i+1);
}
```

Procura de um elemento num vector: recursão e iteração

- Como faríamos o mesmo 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

```
public static
boolean contains(E[] v,E e) {

   int i=0;
   boolean found = false;
   while (i!=v.length && !found) {
      if (v[i].equals(e))
         found = true;
      else i = i+1; // ou: i++;
   }
   return found;
}
```

Implementação Recursiva

```
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==v.length) return false;
  if (v[i].equals(e)) return true;
  return contains(v,e,i+1);
}
```

Procura de um elemento num vector: recursão e iteração

- Como faríamos o mesmo 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

```
public static
boolean contains(E[] v,E e) {

   int i=0;
   boolean found = false;
   while (i!=v.length && !found) {
      if (v[i].equals(e))
         found = true;
      else i = i+1; // ou: i++;
   }
   return found;
}
```

Implementação Recursiva

```
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==v.length) return false;
  if (v[i].equals(e)) return true;
  return contains(v,e,i+1);
}
```

Procura de um elemento num vector: recursão e iteração

- Como faríamos o mesmo 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

```
public static
boolean contains(E[] v,E e) {

   int i=0;
   boolean found = false;
   while (i!=v.length && !found) {
      if (v[i].equals(e))
         found = true;
      else i = i+1; // ou: i++;
   }
   return found;
}
```

Implementação Recursiva

```
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==v.length) return false;
  if (v[i].equals(e)) return true;
  return contains(v,e,i+1);
}
```

Procura de um elemento num vector: recursão e iteração

- Como faríamos o mesmo 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

```
public static
boolean contains(E[] v,E e) {

   int i=0;
   boolean found = false;
   while (i!=v.length && !found) {
      if (v[i].equals(e))
         found = true;
      else i = i+1; // ou: i++;
   }
   return found;
}
```

Implementação Recursiva

```
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==v.length) return false;
  if (v[i].equals(e)) return true;
  return contains(v,e,i+1);
}
```

Procura de um elemento num vector: recursão e iteração

- Como faríamos o mesmo 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

```
public static
boolean contains(E[] v,E e) {

   int i=0;
   boolean found = false;
   while (i!=v.length && !found) {
      if (v[i].equals(e))
         found = true;
      else i = i+1; // ou: i++;
   }
   return found;
}
```

Implementação Recursiva

```
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==v.length) return false;
  if (v[i].equals(e)) return true;
  return contains(v,e,i+1);
}
```

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.

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.

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.

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.

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.

Lista ligada ordenada: semântica

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()`

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()`

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()`

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()`

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()`

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()`

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()`

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()`

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()`

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()`

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: `!contains(v,ne,e) && isSorted(v,ne)`
- (`v` = vector, `ne` = número de elementos, `e` = elemento)

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)

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)

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)

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)

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)

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)

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)

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)

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)

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)

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)

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

```
public class SortedListInt {
  ...
  public boolean isSorted() {
     if (size < 2)
        return true;
     NodeInt p = first;//previous
     NodeInt n = first.next;
     boolean sorted = true;
     while (n!=null && sorted) {
        if (n.elem<p.elem)
          sorted = false
        p = n;
        n = n.next;
     }
     return sorted;
  }
  ...
}
```

Implementação Recursiva

```
public class SortedListInt {
  ...
  public boolean isSorted() {
     if (size < 2) return true;
     return isSorted(first,first.next);
  }
  private
  boolean isSorted(NodeInt p,NodeInt n) {
     if (n == null) return true;
     if (n.elem < p.elem) return false;
     return isSorted(n,n.next);
  }
  ...
}
```

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

```
public class SortedListInt {
  ...
  public boolean isSorted() {
     if (size < 2)
        return true;
     NodeInt p = first;//previous
     NodeInt n = first.next;
     boolean sorted = true;
     while (n!=null && sorted) {
        if (n.elem<p.elem)
          sorted = false
        p = n;
        n = n.next;
     }
     return sorted;
  }
  ...
}
```

Implementação Recursiva

```
public class SortedListInt {
  ...
  public boolean isSorted() {
     if (size < 2) return true;
     return isSorted(first,first.next);
  }
  private
  boolean isSorted(NodeInt p,NodeInt n) {
     if (n == null) return true;
     if (n.elem < p.elem) return false;
     return isSorted(n,n.next);
  }
  ...
}
```

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

```
public class SortedListInt {
  ...
  public boolean isSorted() {
     if (size < 2)
        return true;
     NodeInt p = first;//previous
     NodeInt n = first.next;
     boolean sorted = true;
     while (n!=null && sorted) {
        if (n.elem<p.elem)
          sorted = false
        p = n;
        n = n.next;
     }
     return sorted;
  }
  ...
}
```

Implementação Recursiva

```
public class SortedListInt {
  ...
  public boolean isSorted() {
     if (size < 2) return true;
     return isSorted(first,first.next);
  }
  private
  boolean isSorted(NodeInt p,NodeInt n) {
     if (n == null) return true;
     if (n.elem < p.elem) return false;
     return isSorted(n,n.next);
  }
  ...
}
```

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

```
public class SortedListInt {
  ...
  public boolean isSorted() {
     if (size < 2)
        return true;
     NodeInt p = first;//previous
     NodeInt n = first.next;
     boolean sorted = true;
     while (n!=null && sorted) {
        if (n.elem<p.elem)
          sorted = false
        p = n;
        n = n.next;
     }
     return sorted;
  }
  ...
}
```

Implementação Recursiva

```
public class SortedListInt {
  ...
  public boolean isSorted() {
     if (size < 2) return true;
     return isSorted(first,first.next);
  }
  private
  boolean isSorted(NodeInt p,NodeInt n) {
     if (n == null) return true;
     if (n.elem < p.elem) return false;
     return isSorted(n,n.next);
  }
  ...
}
```

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

```
public class SortedListInt {
  ...
  public boolean isSorted() {
     if (size < 2)
        return true;
     NodeInt p = first;//previous
     NodeInt n = first.next;
     boolean sorted = true;
     while (n!=null && sorted) {
        if (n.elem<p.elem)
          sorted = false
        p = n;
        n = n.next;
     }
     return sorted;
  }
  ...
}
```

Implementação Recursiva

```
public class SortedListInt {
  ...
  public boolean isSorted() {
     if (size < 2) return true;
     return isSorted(first,first.next);
  }
  private
  boolean isSorted(NodeInt p,NodeInt n) {
     if (n == null) return true;
     if (n.elem < p.elem) return false;
     return isSorted(n,n.next);
  }
  ...
}
```

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

```
public class SortedListInt {
  ...
  public boolean isSorted() {
     if (size < 2)
        return true;
     NodeInt p = first;//previous
     NodeInt n = first.next;
     boolean sorted = true;
     while (n!=null && sorted) {
        if (n.elem<p.elem)
          sorted = false
        p = n;
        n = n.next;
     }
     return sorted;
  }
  ...
}
```

Implementação Recursiva

```
public class SortedListInt {
  ...
  public boolean isSorted() {
     if (size < 2) return true;
     return isSorted(first,first.next);
  }
  private
  boolean isSorted(NodeInt p,NodeInt n) {
     if (n == null) return true;
     if (n.elem < p.elem) return false;
     return isSorted(n,n.next);
  }
  ...
}
```

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

```
public class SortedListInt {
  ...
  public boolean isSorted() {
     if (size < 2)
        return true;
     NodeInt p = first;//previous
     NodeInt n = first.next;
     boolean sorted = true;
     while (n!=null && sorted) {
        if (n.elem<p.elem)
          sorted = false
        p = n;
        n = n.next;
     }
     return sorted;
  }
  ...
}
```

Implementação Recursiva

```
public class SortedListInt {
  ...
  public boolean isSorted() {
     if (size < 2) return true;
     return isSorted(first,first.next);
  }
  private
  boolean isSorted(NodeInt p,NodeInt n) {
     if (n == null) return true;
     if (n.elem < p.elem) return false;
     return isSorted(n,n.next);
  }
  ...
}
```

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

```
public class SortedListInt {
  ...
  public boolean isSorted() {
     if (size < 2)
        return true;
     NodeInt p = first;//previous
     NodeInt n = first.next;
     boolean sorted = true;
     while (n!=null && sorted) {
        if (n.elem<p.elem)
          sorted = false
        p = n;
        n = n.next;
     }
     return sorted;
  }
  ...
}
```

Implementação Recursiva

```
public class SortedListInt {
  ...
  public boolean isSorted() {
     if (size < 2) return true;
     return isSorted(first,first.next);
  }
  private
  boolean isSorted(NodeInt p,NodeInt n) {
     if (n == null) return true;
     if (n.elem < p.elem) return false;
     return isSorted(n,n.next);
  }
  ...
}
```

Verificar se um vector está ordenado: recursão e iteração

Implementação Iterativa

```
public static
boolean isSorted(int[] v)
{
   if (v.length < 2)
      return true;
   int i = 1;
   boolean sorted = true;
   while (i!=v.length && sorted){
      if (v[i] < v[i-1])
         sorted = false;
      i++;
   }
   return sorted;
}
```

Implementação Recursiva

```
public static
boolean isSorted(int[] v)
{
   if (v.length < 2)
      return true;
   return isSorted(v,1);
}
private static
boolean isSorted(int[] v,int i)
{
   if (i==v.length) return true;
   if (v[i] < v[i-1]) return false;
   return isSorted(v,i+1);
}
```

Verificar se um vector está ordenado: recursão e iteração

Implementação Iterativa

```
public static
boolean isSorted(int[] v)
{
   if (v.length < 2)
      return true;
   int i = 1;
   boolean sorted = true;
   while (i!=v.length && sorted){
      if (v[i] < v[i-1])
         sorted = false;
      i++;
   }
   return sorted;
}
```

Implementação Recursiva

```
public static
boolean isSorted(int[] v)
{
   if (v.length < 2)
      return true;
   return isSorted(v,1);
}
private static
boolean isSorted(int[] v,int i)
{
   if (i==v.length) return true;
   if (v[i] < v[i-1]) return false;
   return isSorted(v,i+1);
}
```

Verificar se um vector está ordenado: recursão e iteração

Implementação Iterativa

```
public static
boolean isSorted(int[] v)
{
   if (v.length < 2)
      return true;
   int i = 1;
   boolean sorted = true;
   while (i!=v.length && sorted){
      if (v[i] < v[i-1])
         sorted = false;
      i++;
   }
   return sorted;
}
```

Implementação Recursiva

```
public static
boolean isSorted(int[] v)
{
   if (v.length < 2)
      return true;
   return isSorted(v,1);
}
private static
boolean isSorted(int[] v,int i)
{
   if (i==v.length) return true;
   if (v[i] < v[i-1]) return false;
   return isSorted(v,i+1);
}
```

Verificar se um vector está ordenado: recursão e iteração

Implementação Iterativa

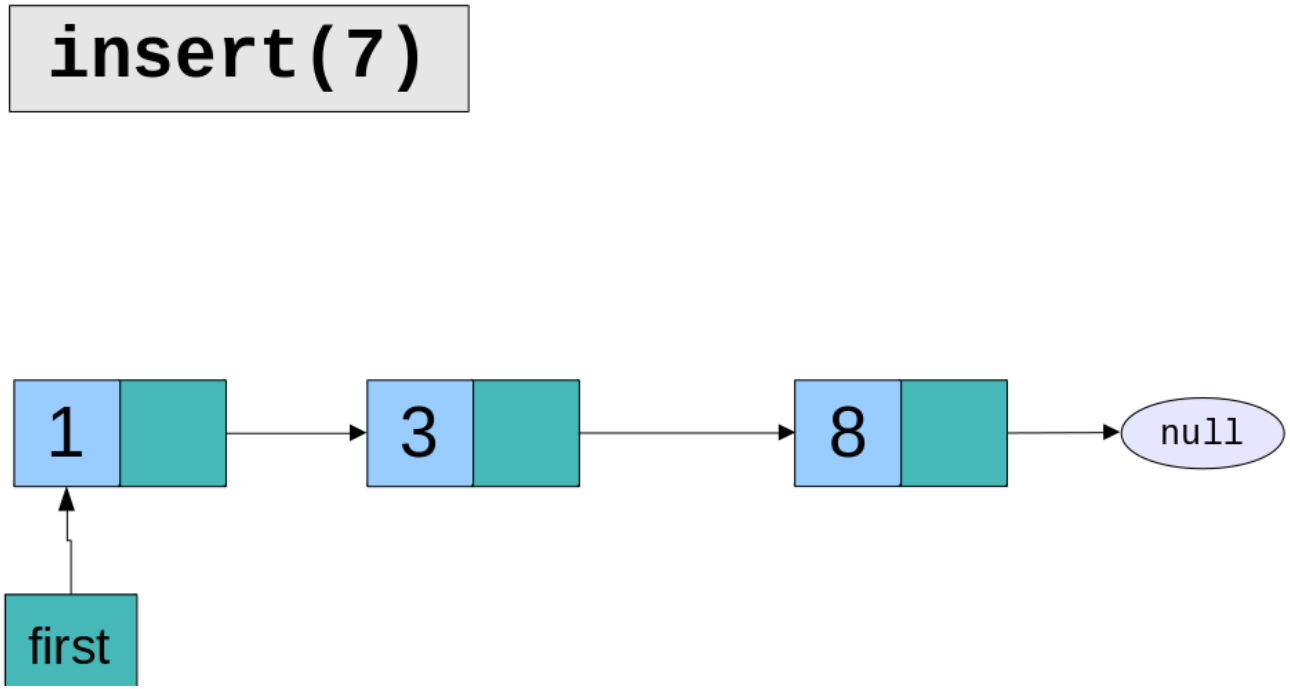
```
public static
boolean isSorted(int[] v)
{
   if (v.length < 2)
      return true;
   int i = 1;
   boolean sorted = true;
   while (i!=v.length && sorted){
      if (v[i] < v[i-1])
         sorted = false;
      i++;
   }
   return sorted;
}
```

Implementação Recursiva

```
public static
boolean isSorted(int[] v)
{
   if (v.length < 2)
      return true;
   return isSorted(v,1);
}
private static
boolean isSorted(int[] v,int i)
{
   if (i==v.length) return true;
   if (v[i] < v[i-1]) return false;
   return isSorted(v,i+1);
}
```

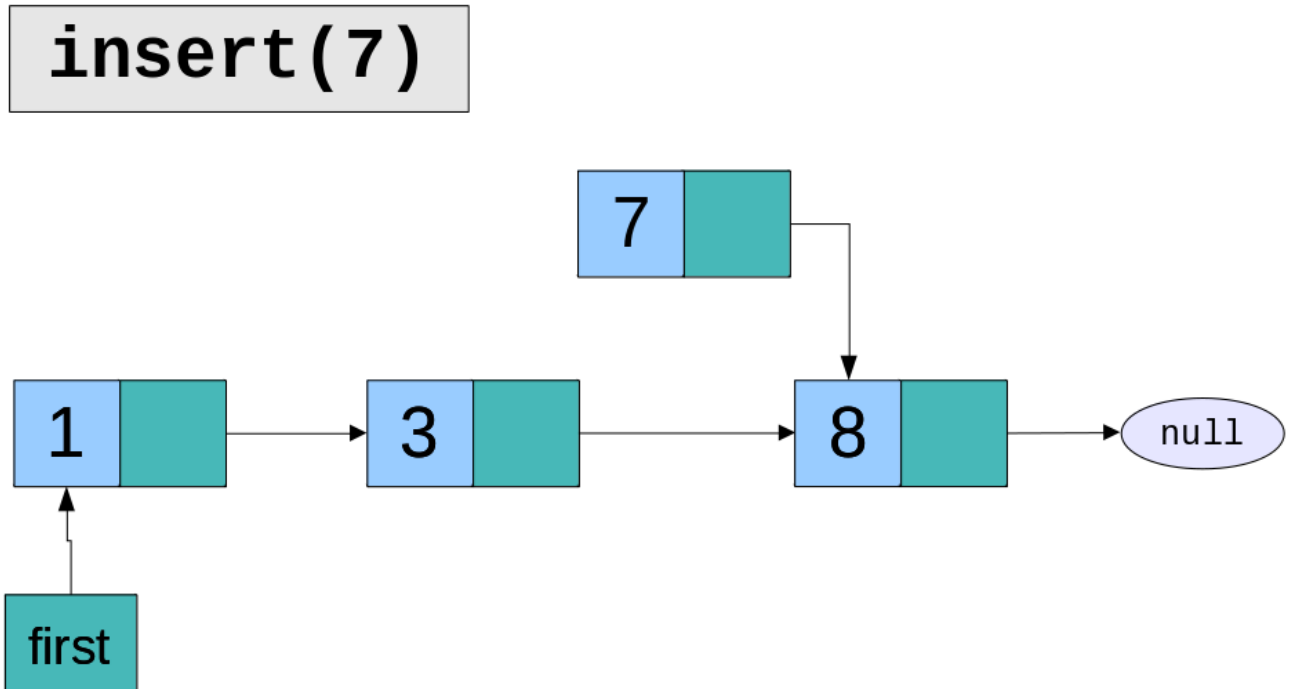
Inserção numa lista ordenada

- Inserção no meio da lista:



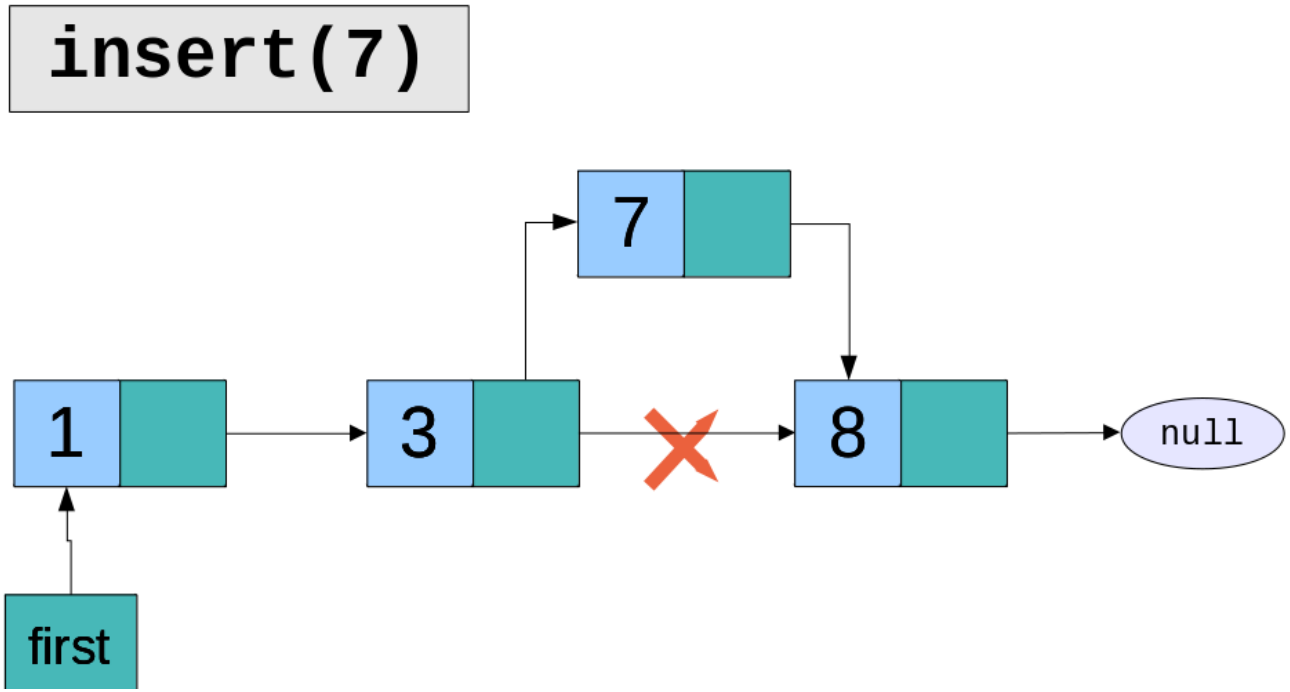
Inserção numa lista ordenada

- Inserção no meio da lista:



Inserção numa lista ordenada

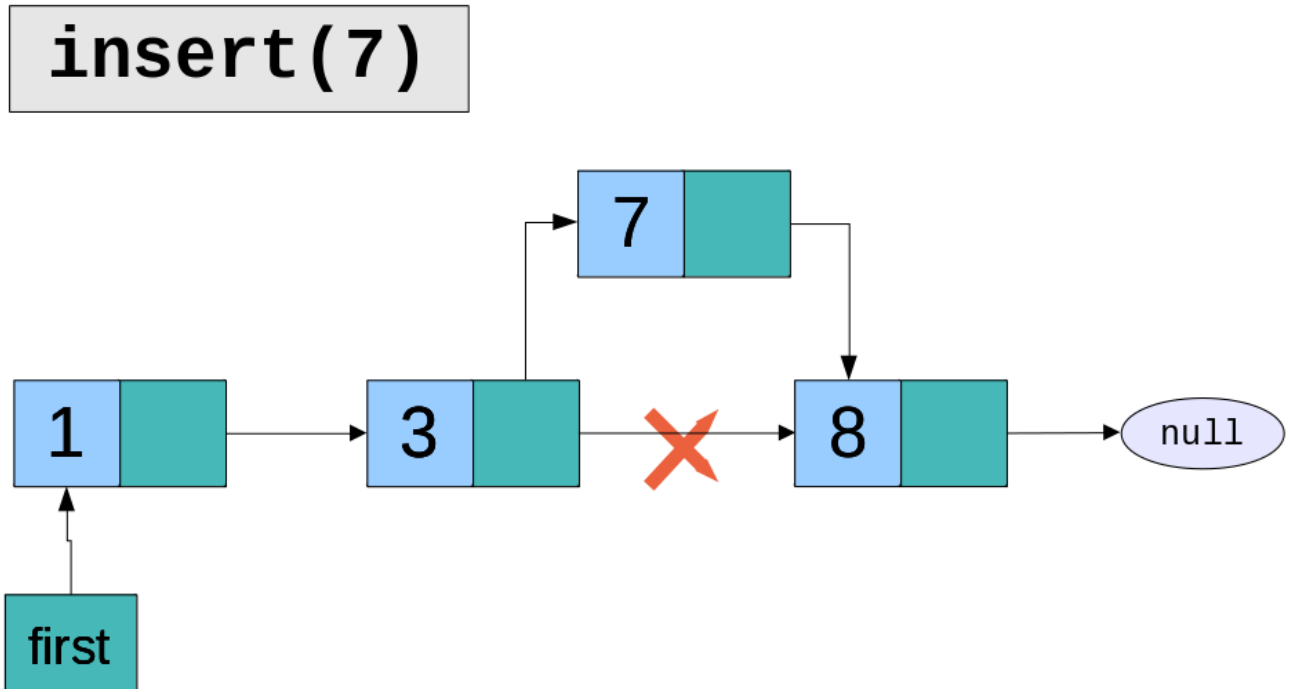
- Inserção no meio da lista:



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

Inserção numa lista ordenada

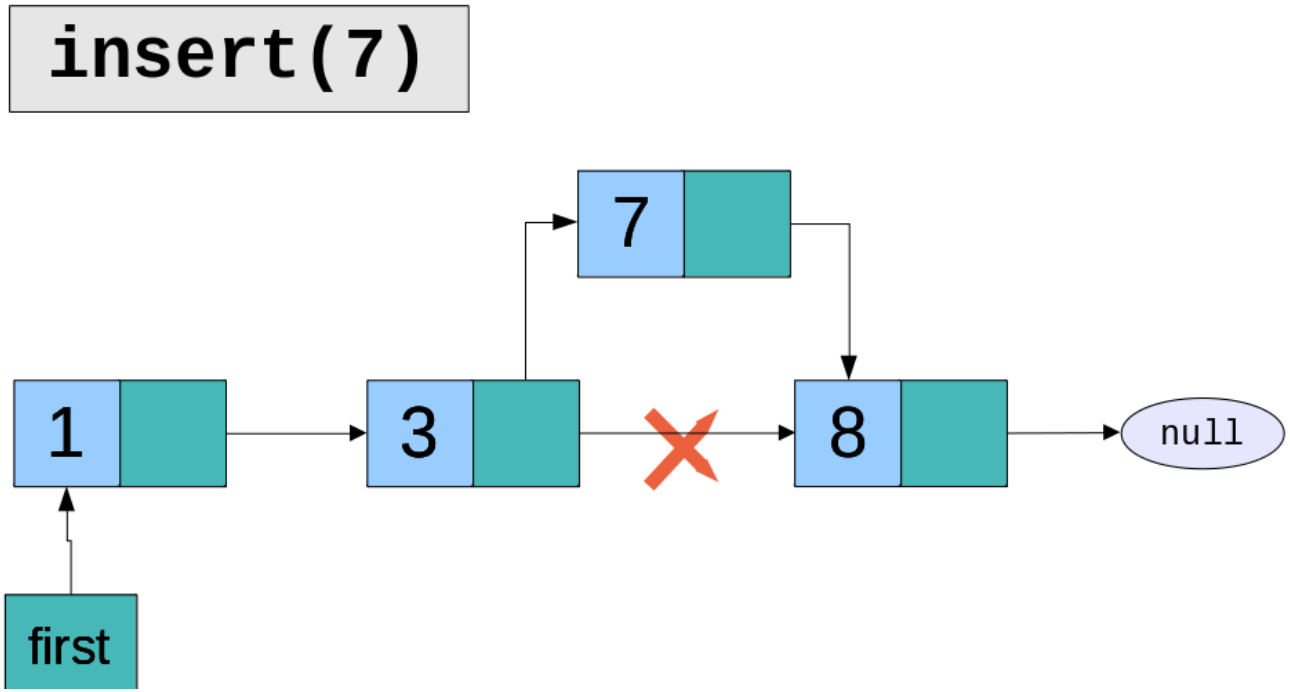
- Inserção no meio da lista:



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

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`

Inserção numa lista ordenada: recursão e iteração

Implementação Iterativa

```
public class SortedListInt {
 ...
 public void insert(int e) {

   if (first==null||e<first.elem)
     first = new NodeInt(e,first);
   else {
     NodeInt p = first;
     NodeInt n = first.next;
     while(n!=null && e>n.elem){
        p = n;
        n = n.next;
     }
     p.next = new NodeInt(e,n);
   }
   size++;
 }
 ...
}
```

Implementação Recursiva

```
public class SortedListInt {
  ...
  public void insert(int e) {
     first = insert(first,e);
     size++;
  }
  private NodeInt insert(NodeInt n,int e){
      if (n==null || e<n.elem)
         return new NodeInt(e,n);
      n.next = insert(n.next,e);
      return n;
  }
  ...
}
```

Inserção numa lista ordenada: recursão e iteração

Implementação Iterativa

```
public class SortedListInt {
  ...
 public void insert(int e) {

    if (first==null||e<first.elem)
      first = new NodeInt(e,first);
    else {
      NodeInt p = first;
      NodeInt n = first.next;
      while(n!=null && e>n.elem){
         p = n;
         n = n.next;
      }
      p.next = new NodeInt(e,n);
    }
    size++;
 }
  ...
}
```

Implementação Recursiva

```
public class SortedListInt {
   ...
  public void insert(int e) {
      first = insert(first,e);
      size++;
  }
  private NodeInt insert(NodeInt n,int e){
       if (n==null || e<n.elem)
          return new NodeInt(e,n);
       n.next = insert(n.next,e);
       return n;
  }
   ...
}
```

Inserção numa lista ordenada: recursão e iteração

Implementação Iterativa

```
public class SortedListInt {
 ...
 public void insert(int e) {

   if (first==null||e<first.elem)
     first = new NodeInt(e,first);
   else {
     NodeInt p = first;
     NodeInt n = first.next;
     while(n!=null && e>n.elem){
        p = n;
        n = n.next;
     }
     p.next = new NodeInt(e,n);
   }
   size++;
 }
 ...
}
```

Implementação Recursiva

```
public class SortedListInt {
   ...
  public void insert(int e) {
     first = insert(first,e);
     size++;
  }
  private NodeInt insert(NodeInt n,int e){
      if (n==null || e<n.elem)
         return new NodeInt(e,n);
      n.next = insert(n.next,e);
      return n;
  }
  ...
}
```

Inserção num vector ordenado

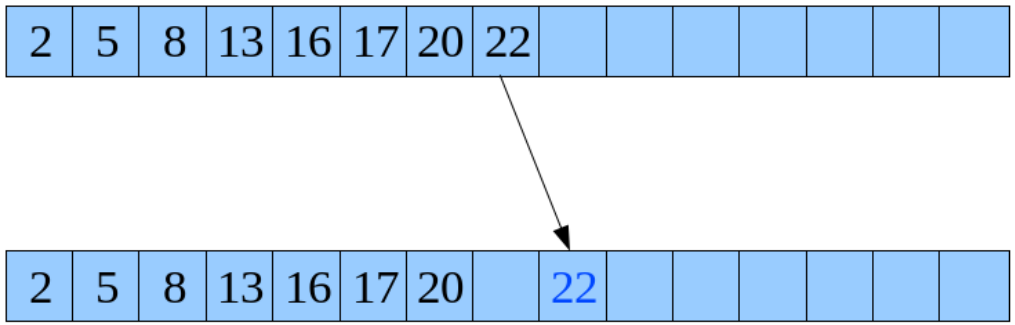
- Inserção no meio do vector:

```
insert(18)
```

Inserção num vector ordenado

- Inserção no meio do vector:

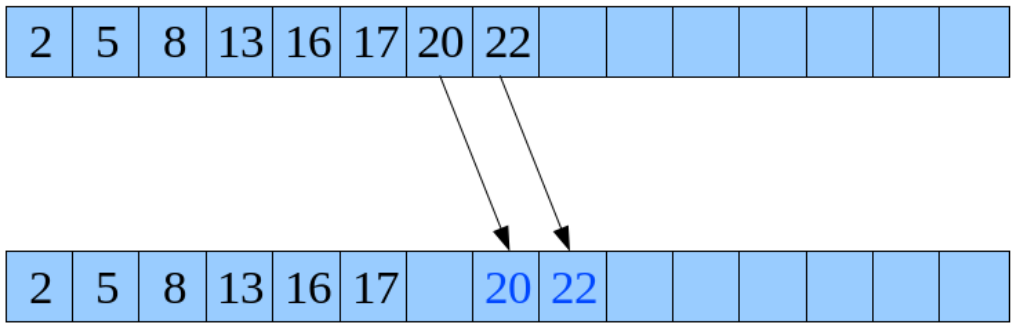
```
insert(18)
```



Inserção num vector ordenado

- Inserção no meio do vector:

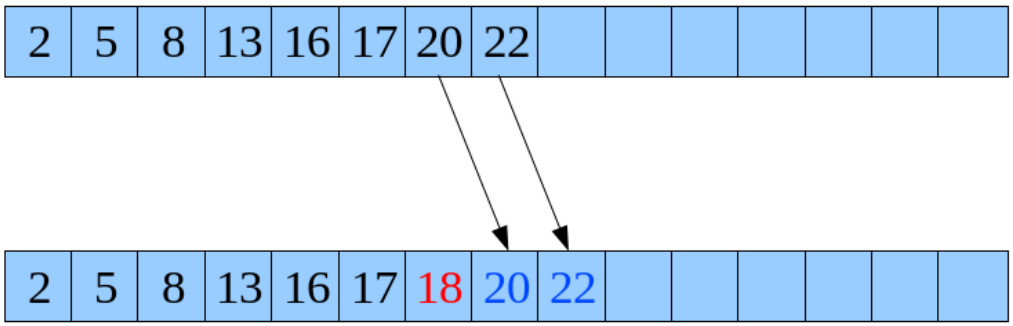
```
insert(18)
```



Inserção num vector ordenado

- Inserção no meio do vector:

```
insert(18)
```



Inserção num vector ordenado: recursão e iteração

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

Implementação Iterativa

```
public static int
    insert(int[] v,int ne,int e){

  int i=ne;
  while (i>0 && e<v[i-1]) {
    v[i] = v[i-1];
    i--;
  }
  v[i] = e;
  return ne+1;
}
```

Implementação Recursiva

```
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>v[i-1]) v[i] = e;
  else {
    v[i] = v[i-1];
    shiftInsert(v,e,i-1);
  }
}
```

Inserção num vector ordenado: recursão e iteração

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

Implementação Iterativa

```
public static int
    insert(int[] v,int ne,int e){

  int i=ne;
  while (i>0 && e<v[i-1]) {
    v[i] = v[i-1];
    i--;
  }
  v[i] = e;
  return ne+1;
}
```

Implementação Recursiva

```
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>v[i-1]) v[i] = e;
  else {
    v[i] = v[i-1];
    shiftInsert(v,e,i-1);
  }
}
```

Inserção num vector ordenado: recursão e iteração

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

Implementação Iterativa

```
public static int
    insert(int[] v,int ne,int e){

  int i=ne;
  while (i>0 && e<v[i-1]) {
    v[i] = v[i-1];
    i--;
  }
  v[i] = e;
  return ne+1;
}
```

Implementação Recursiva

```
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>v[i-1]) v[i] = e;
  else {
    v[i] = v[i-1];
    shiftInsert(v,e,i-1);
  }
}
```

Inserção num vector ordenado: recursão e iteração

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

Implementação Iterativa

```
public static int
     insert(int[] v,int ne,int e){

  int i=ne;
  while (i>0 && e<v[i-1]) {
    v[i] = v[i-1];
    i--;
  }
  v[i] = e;
  return ne+1;
}
```

Implementação Recursiva

```
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>v[i-1]) v[i] = e;
  else {
    v[i] = v[i-1];
    shiftInsert(v,e,i-1);
  }
}
```

Inserção num vector ordenado: recursão e iteração

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

Implementação Iterativa

```
public static int
     insert(int[] v,int ne,int e){

  int i=ne;
  while (i>0 && e<v[i-1]) {
    v[i] = v[i-1];
    i--;
  }
  v[i] = e;
  return ne+1;
}
```

Implementação Recursiva

```
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>v[i-1]) v[i] = e;
  else {
    v[i] = v[i-1];
    shiftInsert(v,e,i-1);
  }
}
```

Implementação de uma lista ordenada genérica

- Qualquer objecto Java tem o método `equals()`
- No entanto, só alguns objectos têm o método `compareTo()` sem o qual não é possível manter uma lista ordenada
- Podemos criar classes genéricas em que o tipo ou tipos não especificados 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>();
   ...
}
```

Implementação de uma lista ordenada genérica

- Qualquer objecto Java tem o método `equals()`
- No entanto, só alguns objectos têm o método `compareTo()` sem o qual não é possível manter uma lista ordenada
- Podemos criar classes genéricas em que o tipo ou tipos não especificados 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>();
   ...
}
```

Implementação de uma lista ordenada genérica

- Qualquer objecto Java tem o método `equals()`
- No entanto, só alguns objectos têm o método `compareTo()` sem o qual não é possível manter uma lista ordenada
- Podemos criar classes genéricas em que o tipo ou tipos não especificados 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>();
   ...
}
```

Implementação de uma lista ordenada genérica

- Qualquer objecto Java tem o método `equals()`
- No entanto, só alguns objectos têm o método `compareTo()` sem o qual não é possível manter uma lista ordenada
- Podemos criar classes genéricas em que o tipo ou tipos não especificados 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>();
    ...
}
```