UFMG
UNIVERSIDADE FEDERAL
DE MINAS GERAIS

DCC004 - Algoritmos e Estruturas de Dados II

Listas duplamente encadeadas e árvores binárias

Renato Martins

Email: renato.martins@dcc.ufmg.br

https://www.dcc.ufmg.br/~renato.martins/courses/DCC004

Material adaptado de PDS2 - Douglas Macharet e Flávio Figueiredo

DCC
DEPARTAMENTO DE
CIÊNCIA DA COMPUTAÇÃO

Revisão: Remoção em uma lista simples

Código

```
void ListaSimplesmenteEncadeada::remove_iesimo(int i) {
  if (i >= this->_num_elementos_inseridos)
    return;
  node_t *atual = this->_inicio;
  node_t *anterior = nullptr;
  for (int j = 0; j < i; j++) {
    anterior = atual;
    atual = atual->proximo;
  }
  if (anterior != nullptr)
    anterior->proximo = atual->proximo;
  if (i == 0)
    this->_inicio = atual->proximo;
  if (i == this->_num_elementos_inseridos - 1)
    this->_fim = anterior;
  this->_num_elementos_inseridos--;
  delete atual;
}
```

Código

```
void ListaSimplesmenteEncadeada::remove_iesimo(int i) {
  if (i >= this->_num_elementos_inseridos)
    return;
  node_t *atual = this->_inicio;
  node_t *anterior = nullptr;
  for (int j = 0; j < i; j++) {
    anterior = atual;
    atual = atual->proximo;
  }
  if (anterior != nullptr)
    anterior->proximo = atual->proximo;
  if (i == 0)
    this->_inicio = atual->proximo;
  if (i == this->_num_elementos_inseridos - 1)
    this->_fim = anterior;
  this->_num_elementos_inseridos--;
  delete atual;
}
```

Verifica se "i" é válido

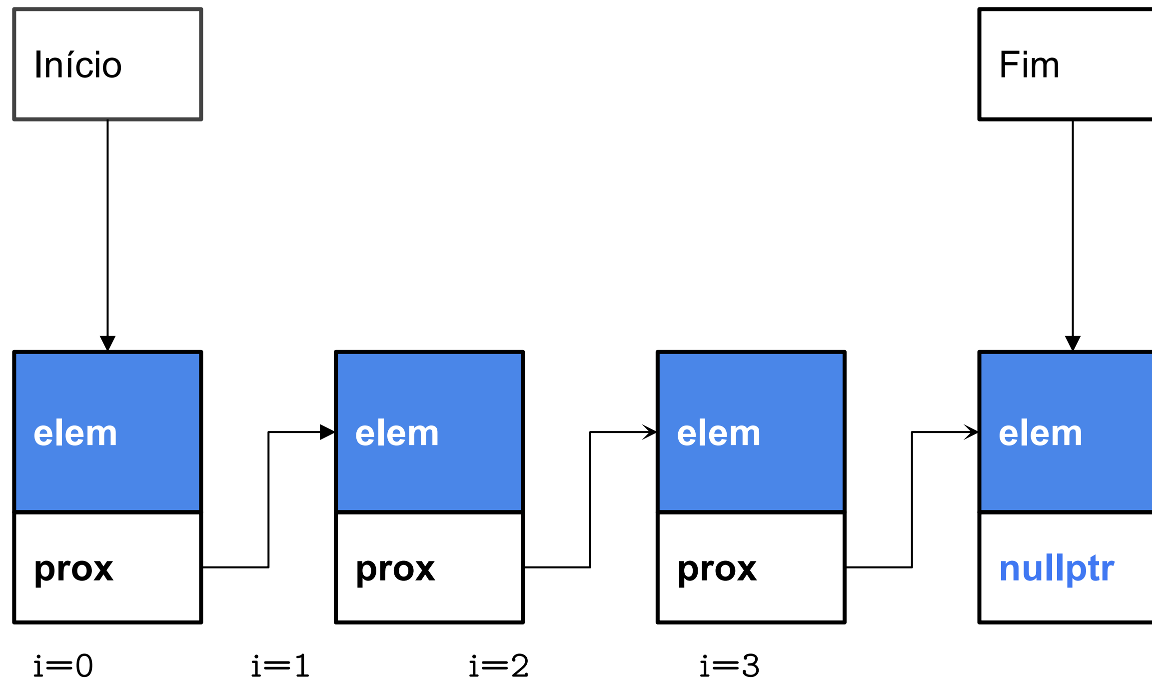
Pula "i" vezes guardando o anterior

Vários "ifs" de casos especiais

Casos especiais

Seu código sempre vai ter tratamento para *corner* cases

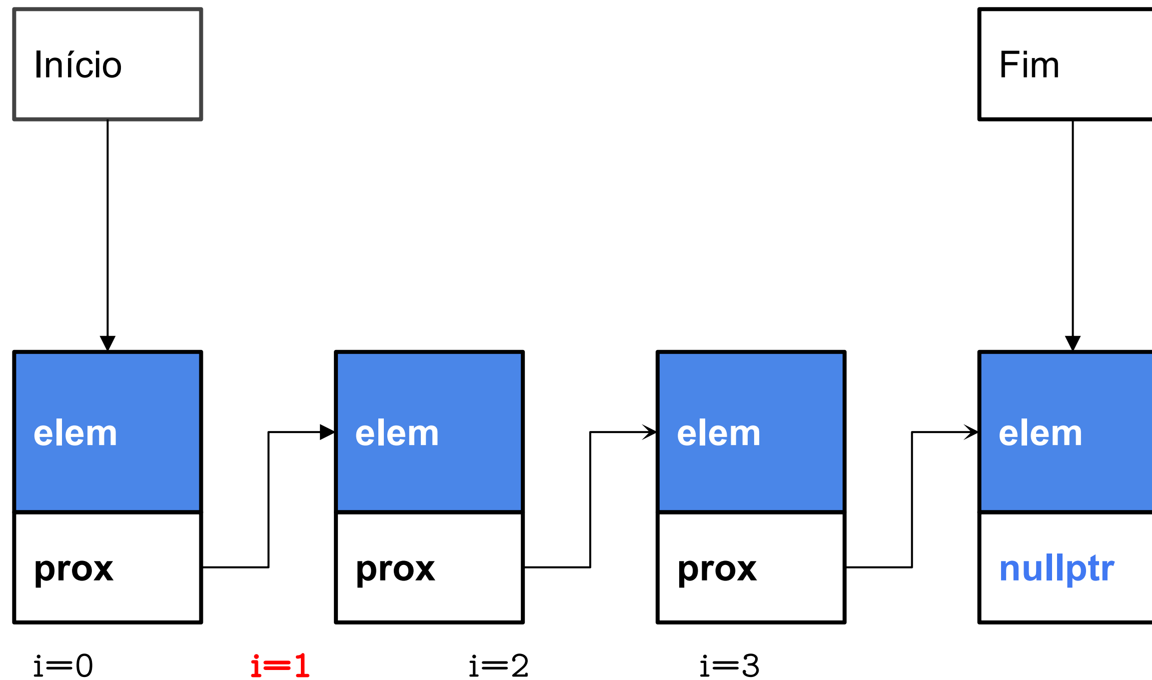
Olhando para o caso geral



Casos especiais

Seu código sempre vai ter tratamento para *corner* cases

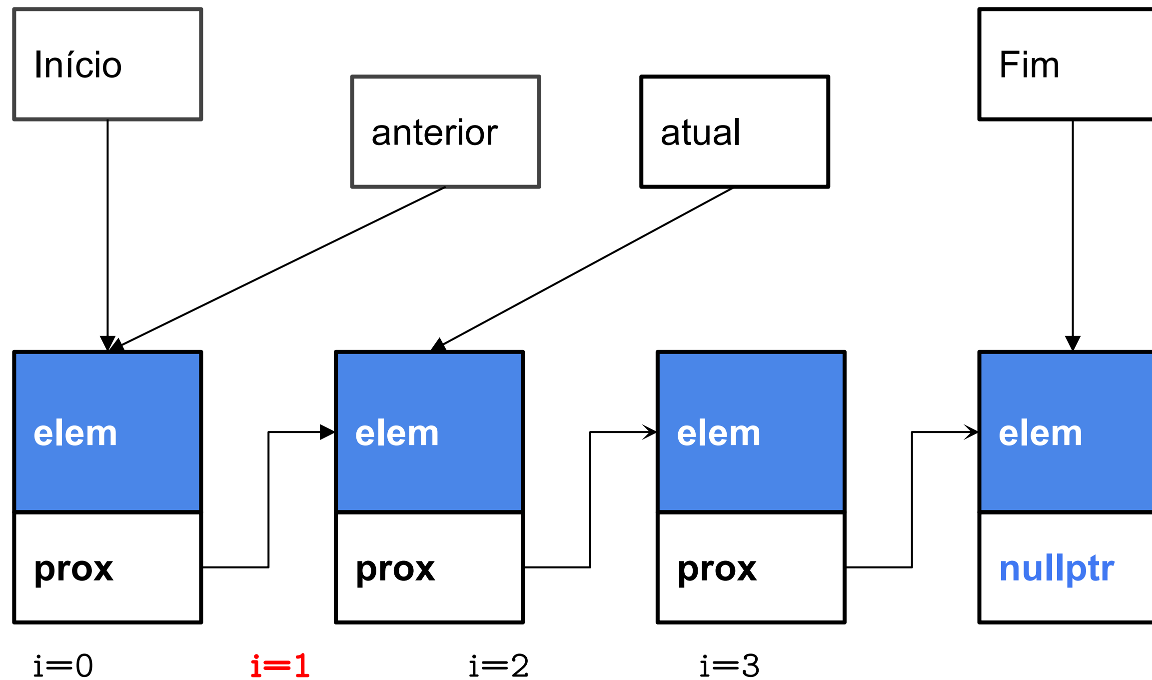
Olhando para o caso geral (**i=1**)



Casos especiais

Seu código sempre vai ter tratamento para *corner* cases

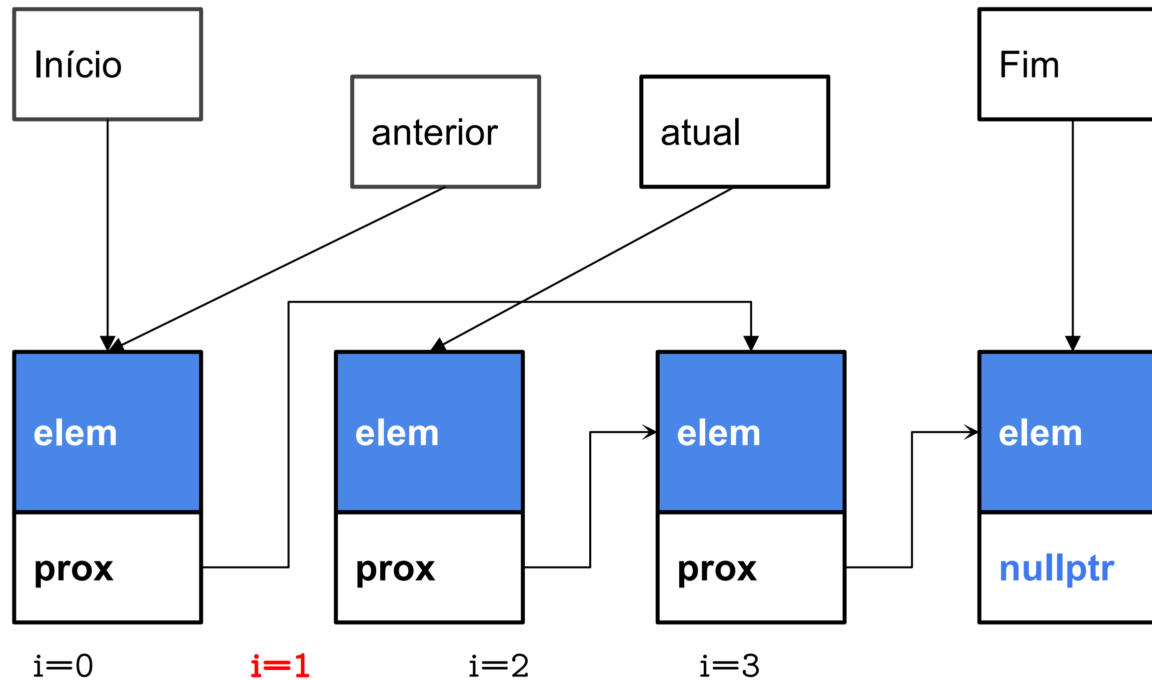
Caminhanhos guardando o anterior



Casos especiais

Seu código sempre vai ter tratamento para *corner* cases

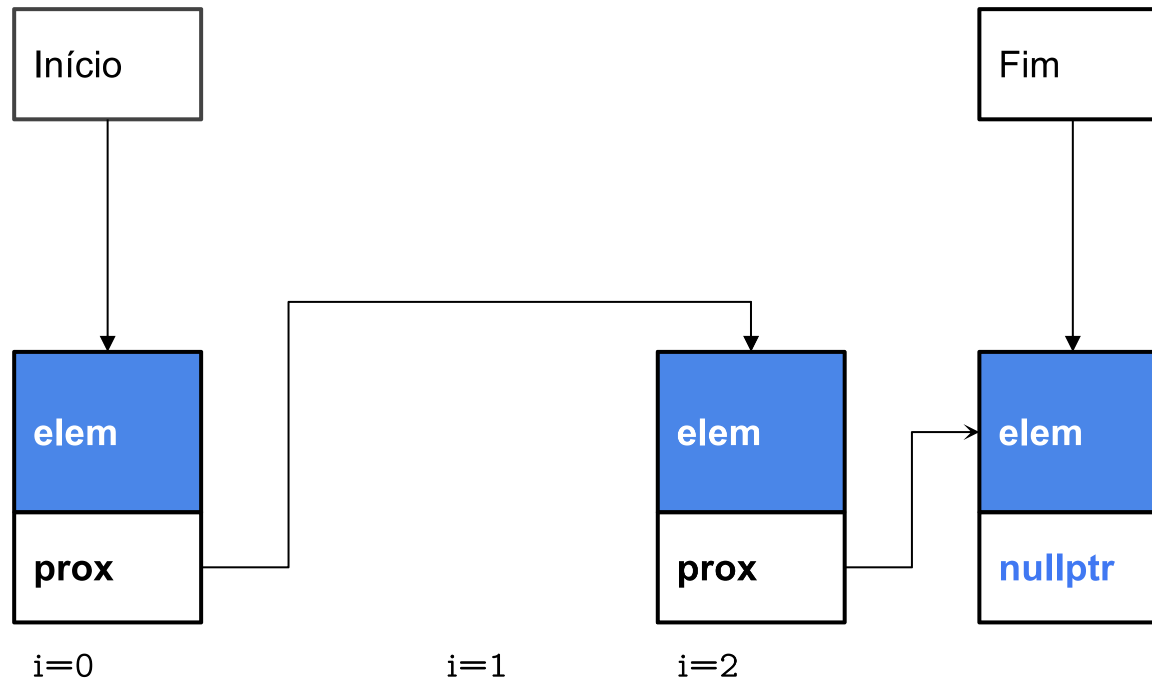
Atualizamos o ponteiro do anterior



Casos especiais

Seu código sempre vai ter tratamento para *corner* cases

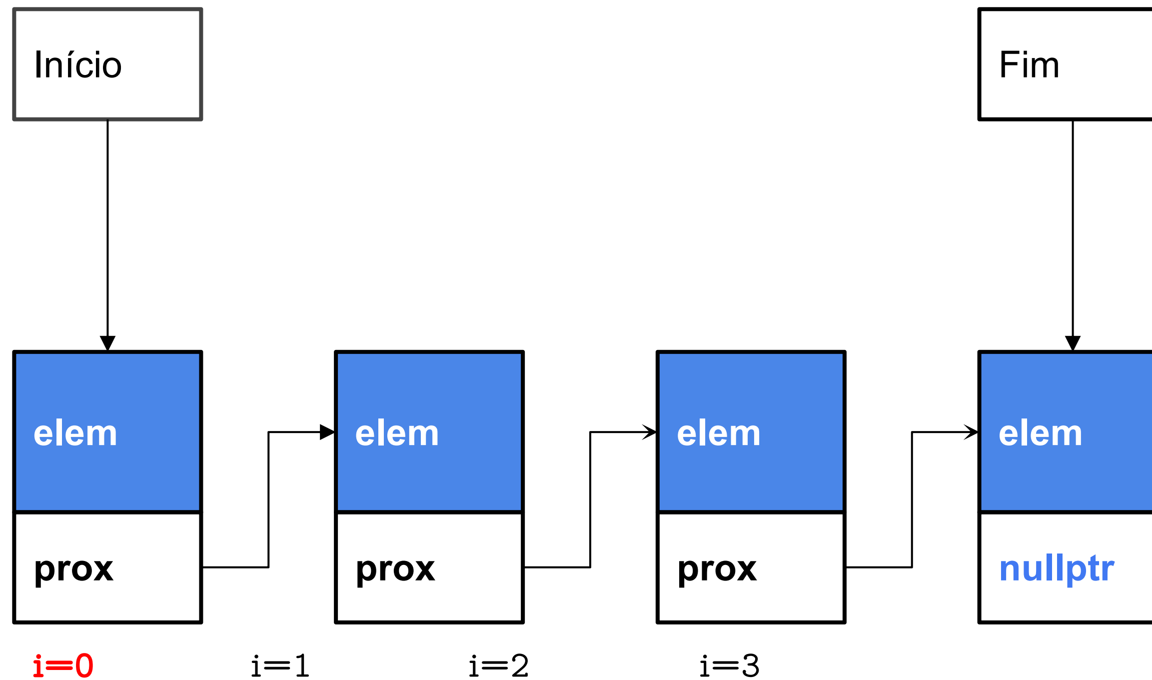
Free!



Casos especiais

Seu código sempre vai ter tratamento para *corner cases*

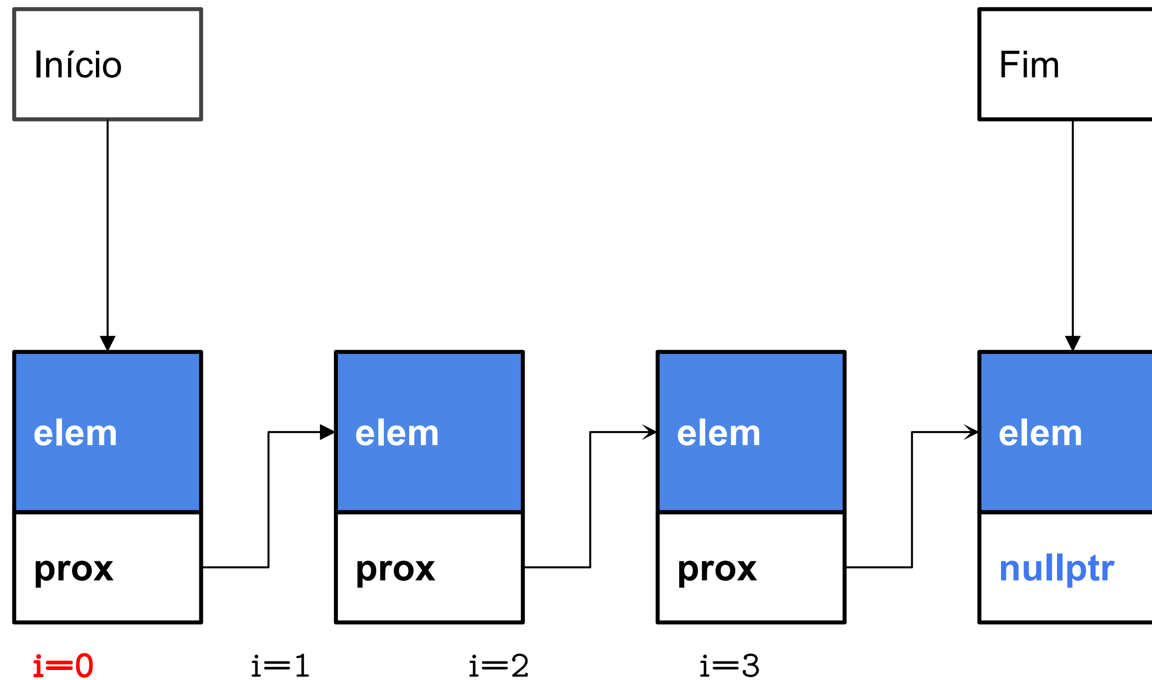
Caso especial (i=0)



Casos especiais

Seu código sempre vai ter tratamento para *corner* cases

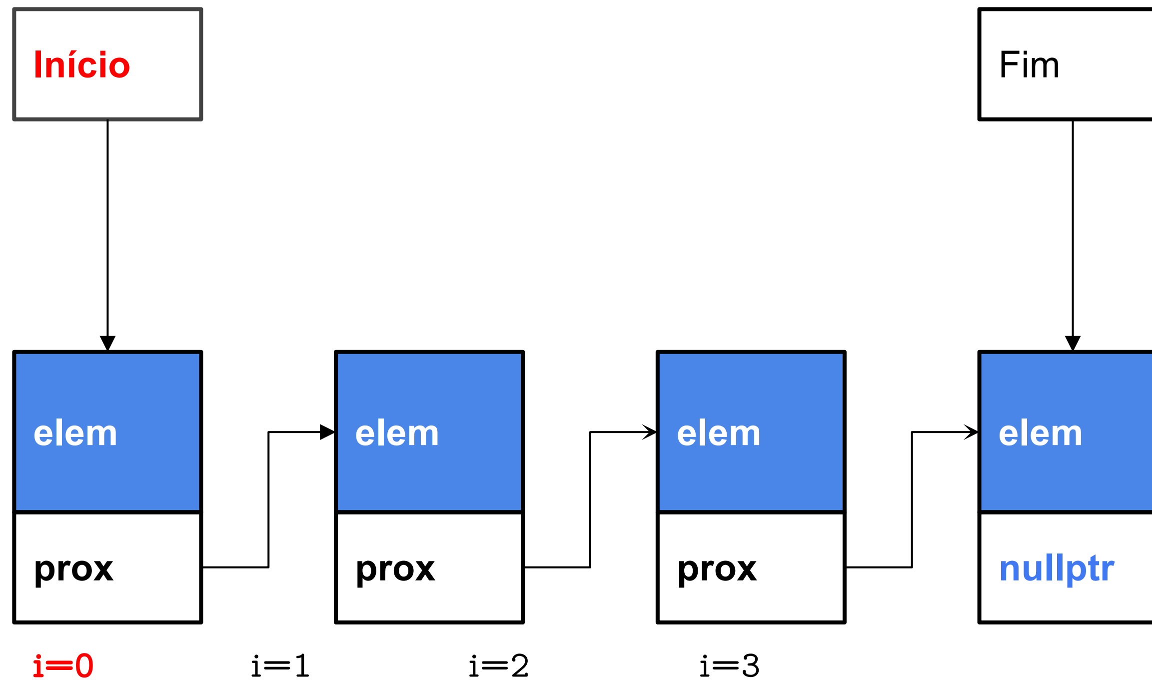
Qual o problema?



Casos especiais

Seu código sempre vai ter tratamento para *corner* cases

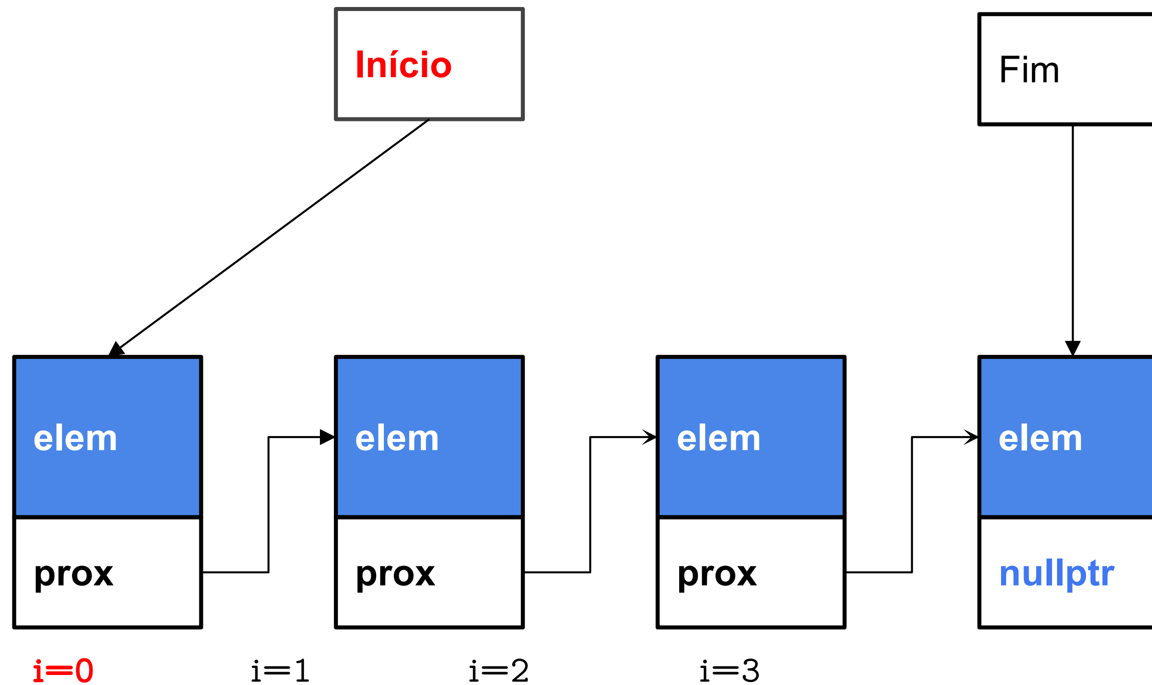
Qual o problema? Ponteiro início!



Casos especiais

Seu código sempre vai ter tratamento para *corner cases*

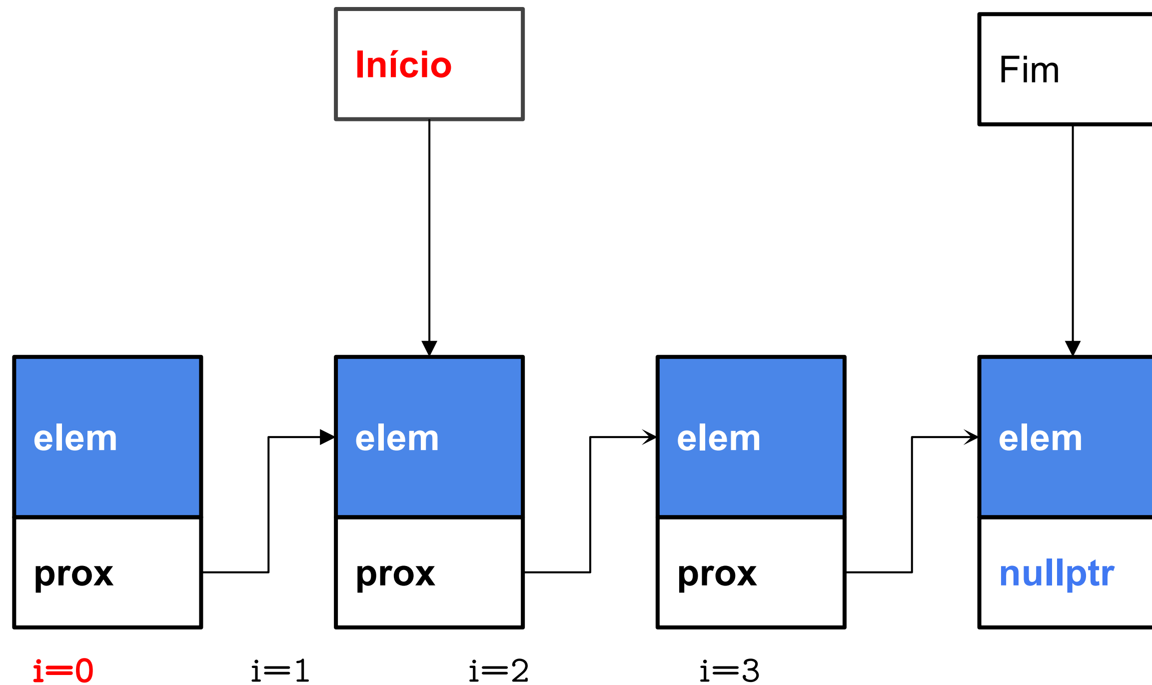
```
if (i == 0)
  this->_inicio = atual->proximo;
```



Casos especiais

Seu código sempre vai ter tratamento para *corner* cases

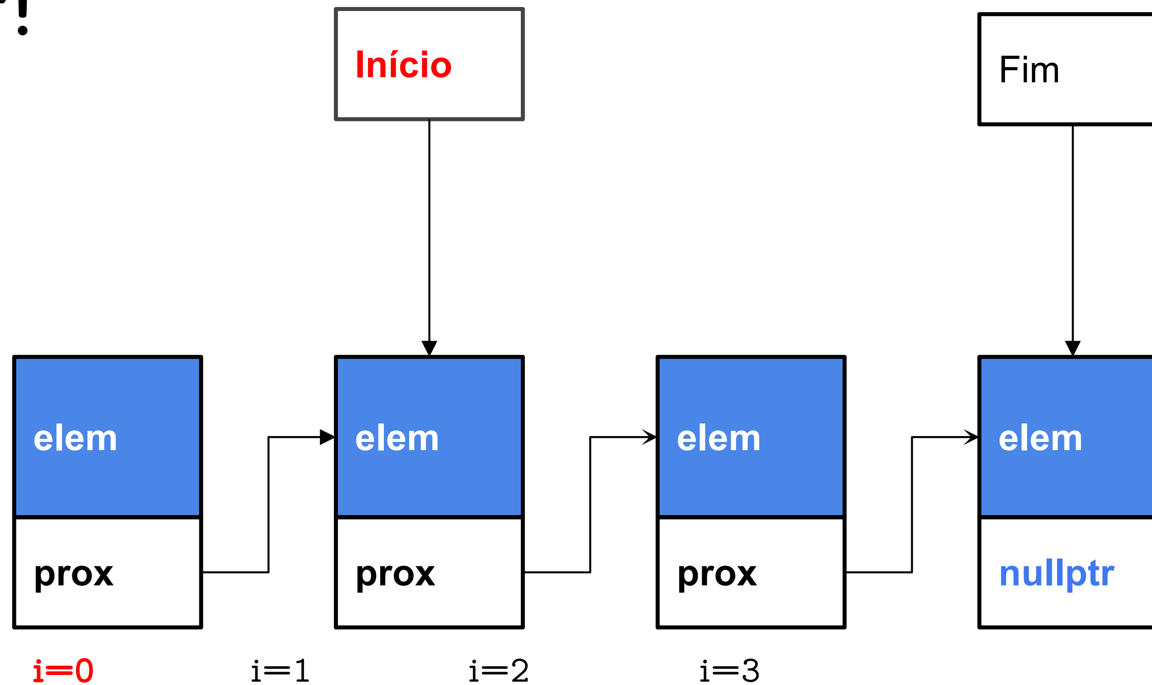
Mais problemas?



Casos especiais

Seu código sempre vai ter tratamento para *corner* cases

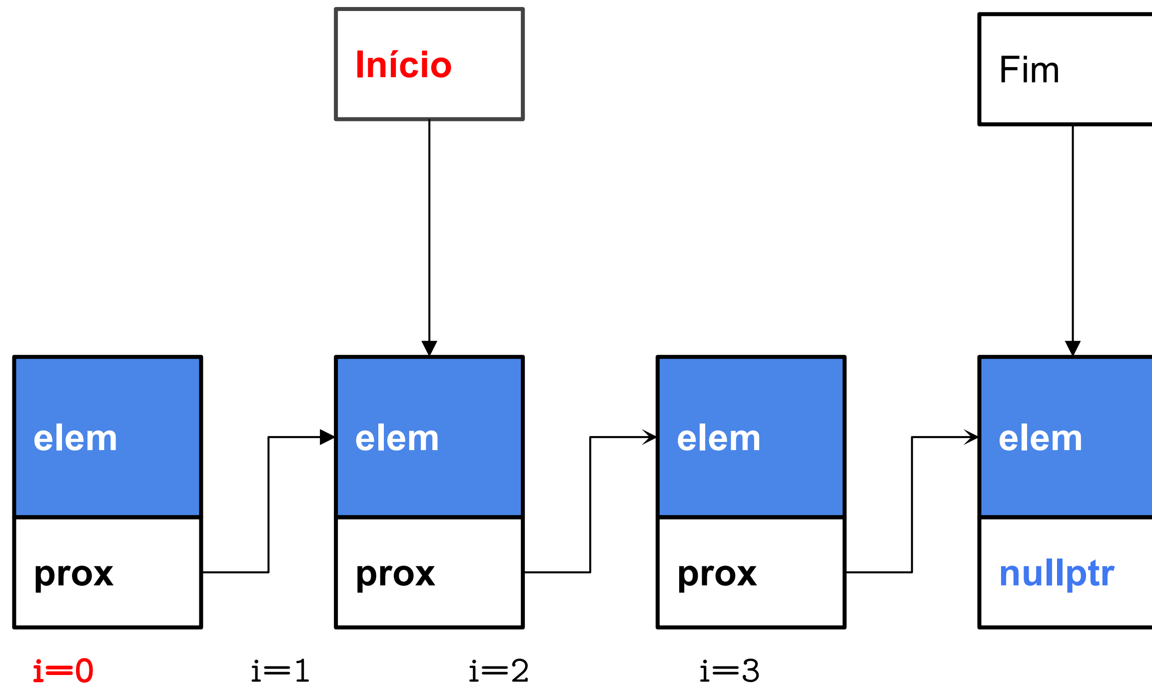
Mais problemas? Valor de anterior é nullptr!



Casos especiais

Seu código sempre vai ter tratamento para *corner cases*

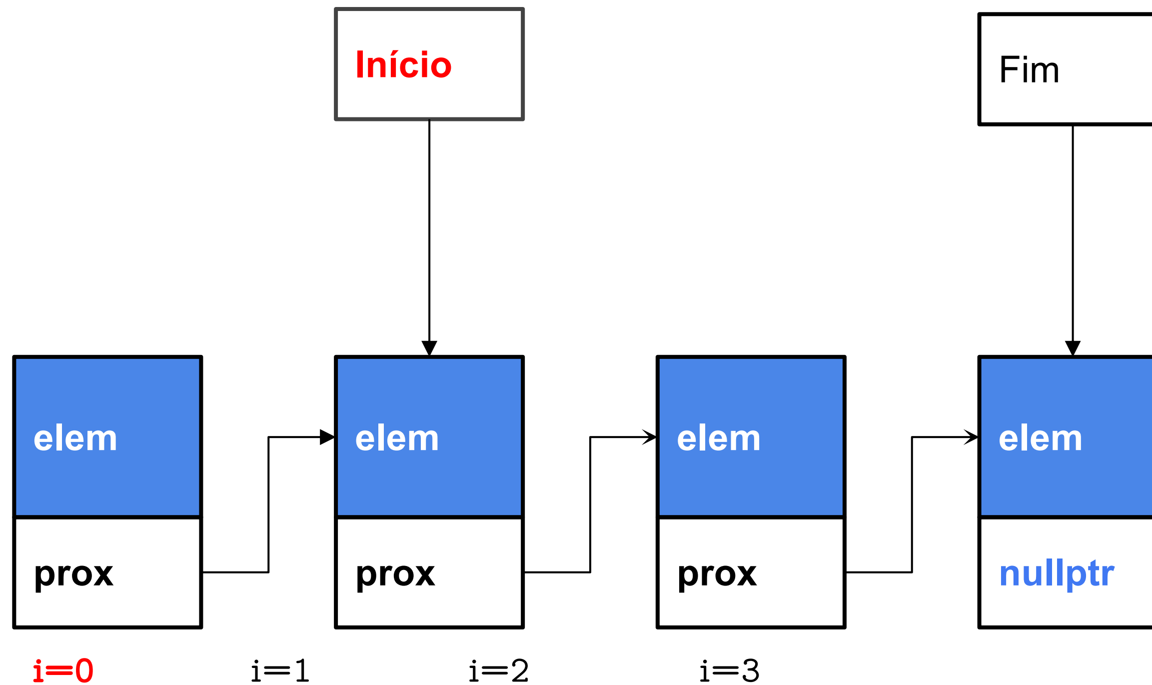
```
if (anterior != nullptr)
  anterior->proximo = atual->proximo;
```



Casos especiais

Seu código sempre vai ter tratamento para *corner cases*

Sem isto? **segfault!**



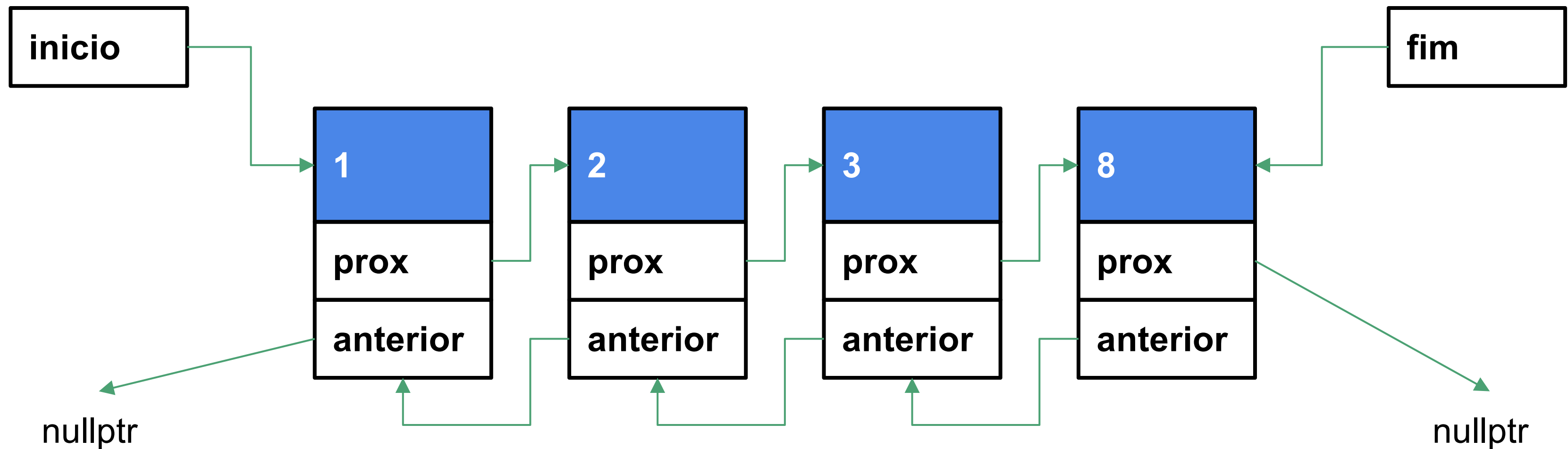
Segfault

- Ao fazer uso de ponteiros é comum ver segmentation faults
- Causas:
  - Você acessou um ponteiro nulo
  - Você acessou um ponteiro lixo!
- No exemplo anterior existem mais casos especiais
  - Quais são?

Lista Duplamente Encadeada

Lista Duplamente Encadeada

Ideia



Olhando para o .h

```
#ifndef PDS2_LISTADUPLA_H
#define PDS2_LISTADUPLA_H
struct node_t {
  int elemento;
  node_t *anterior;
  node_t *proximo;
};

class ListaDuplamenteEncadeada {
private:
  node_t *_inicio;
  node_t *_fim;
  int _num_elementos_inseridos;
public:
  ListaDuplamenteEncadeada();
  ~ListaDuplamenteEncadeada();
  void inserir_elemento(int elemento);
  void remove_iesimo(int i);
  void imprimir();
};
#endif
```

Ponteiros para o anterior e próximo

Essencialmente a mesma coisa de antes

Vantagens
Em relação a lista simples...

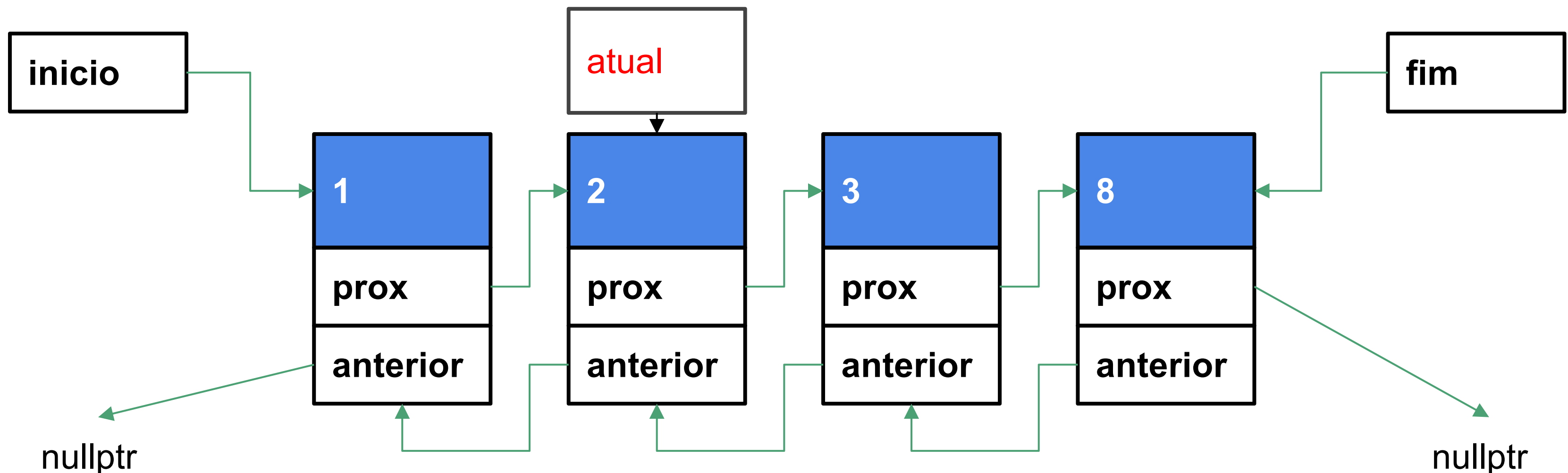
- Como imprimir nós nas duas ordens
  - Complicado na lista simples
  - Podemos iterar nos dois sentidos
- Não precisamos ficar guardando o ponteiro anterior
- Já existe na estrutura

Código de remover i-ésimo
Um pouco mais simples do que antes

```
void ListaDuplamenteEncadeada::remove_iesimo(int i) {
  if (i >= this->_num_elementos_inseridos) {
    return;
  }
  node_t *atual = this->_inicio;
  for (int j = 0; j < i; j++)
    atual = atual->proximo;
  if (atual->proximo != nullptr)
    atual->proximo->anterior = atual->anterior;
  if (atual->anterior != nullptr)
    atual->anterior->proximo = atual->proximo;
  if (i == 0)
    this->_inicio = atual->proximo;
  if (i == this->_num_elementos_inseridos - 1)
    this->_fim = atual->anterior;
  this->_num_elementos_inseridos--;
  delete atual;
}
```

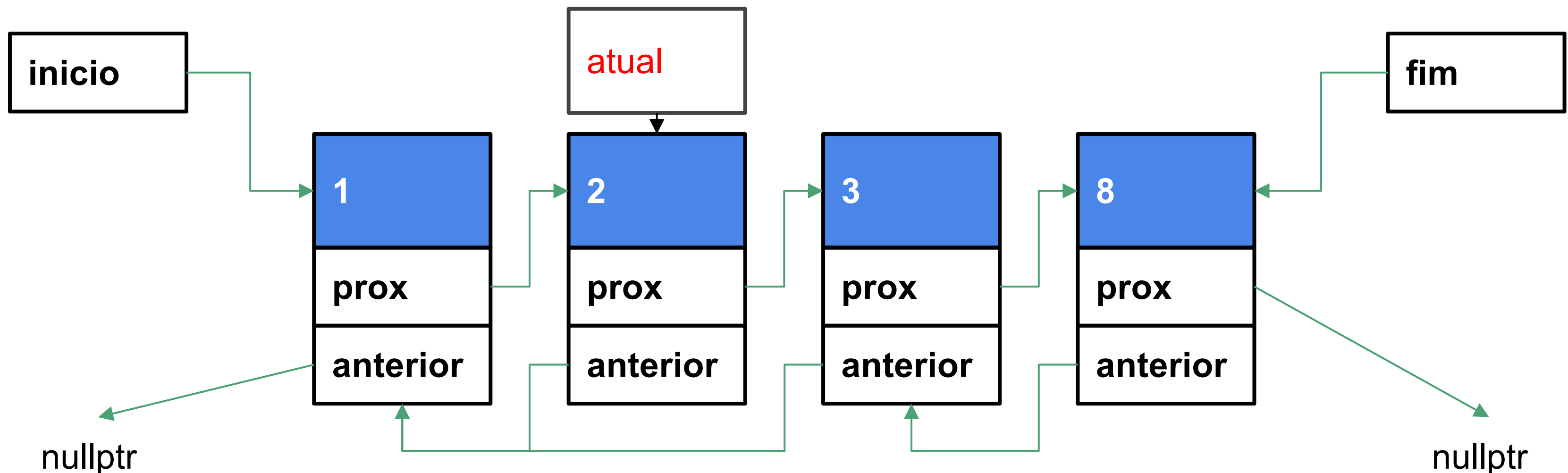
Código de remover i-ésimo
Um pouco mais simples do que antes

```
void ListaDuplamenteEncadeada::remove_iesimo(int i) {
  // . . .
  if (atual->proximo != nullptr)
    atual->proximo->anterior = atual->anterior;
  if (atual->anterior != nullptr)
    atual->anterior->proximo = atual->proximo;
  // . . .
```



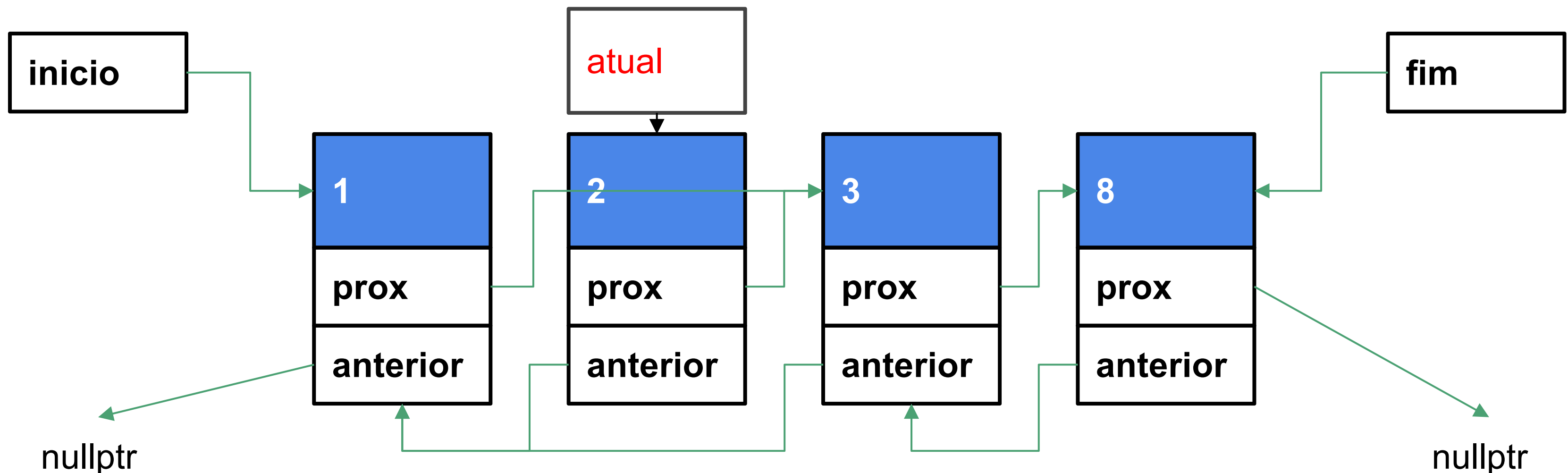
Código de remover i-ésimo
Um pouco mais simples do que antes

```
void ListaDuplamenteEncadeada::remove_iesimo(int i) {
  // . . .
  if (atual->proximo != nullptr)
    atual->proximo->anterior = atual->anterior;
  if (atual->anterior != nullptr)
    atual->anterior->proximo = atual->proximo;
  // . . .
```



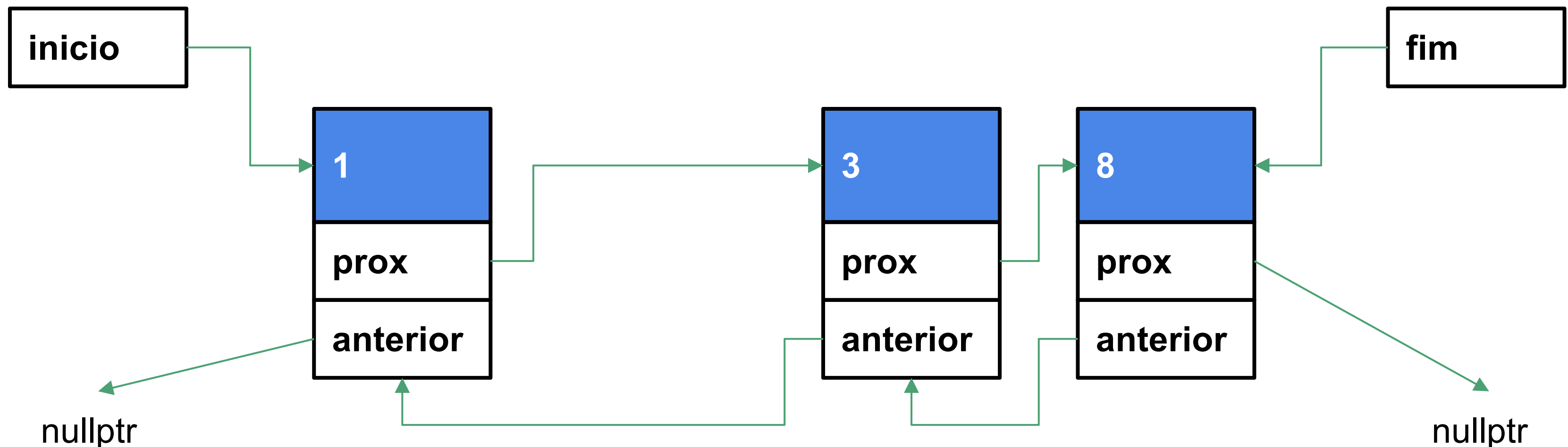
Código de remover i-ésimo
Um pouco mais simples do que antes

```
void ListaDuplamenteEncadeada::remove_iesimo(int i) {
  // . . .
  if (atual->proximo != nullptr)
    atual->proximo->anterior = atual->anterior;
  if (atual->anterior != nullptr)
    atual->anterior->proximo = atual->proximo;
  // . . .
```



Código de remover i-ésimo
Um pouco mais simples do que antes

```
void ListaDuplamenteEncadeada::remove_iesimo(int i) {
  // . . .
  if (atual->proximo != nullptr)
    atual->proximo->anterior = atual->anterior;
  if (atual->anterior != nullptr)
    atual->anterior->proximo = atual->proximo;
  // . . .
```



Problemas que precisam de listas
Ou de conceitos similares

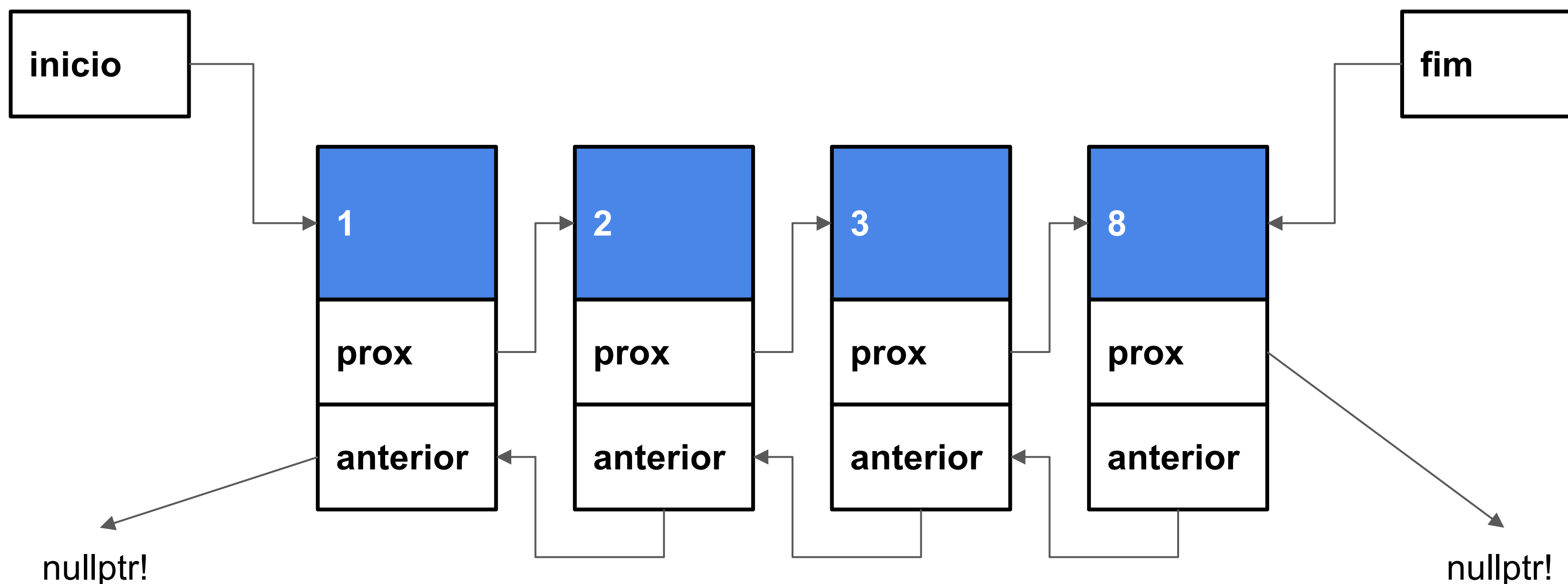
- Tipos simples de dados têm limites
  - int, float, double, long
- Como representar números gigantescos?

Problemas que precisam de listas
Ou de conceitos similares

- Tipos simples de dados têm limites
  - int, float, double, long
- Como representar números gigantescos?
  - Sequência de dígitos

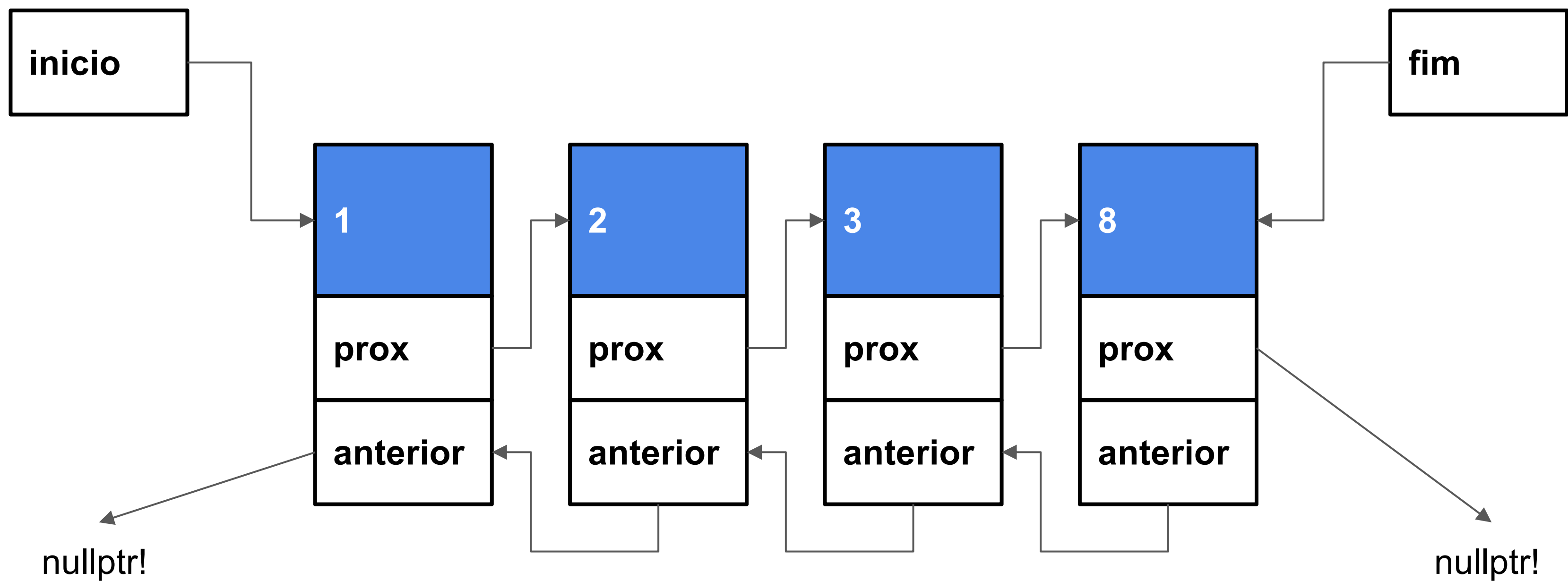
Ideia

Lista duplamente encadeada: 1238 abaixo



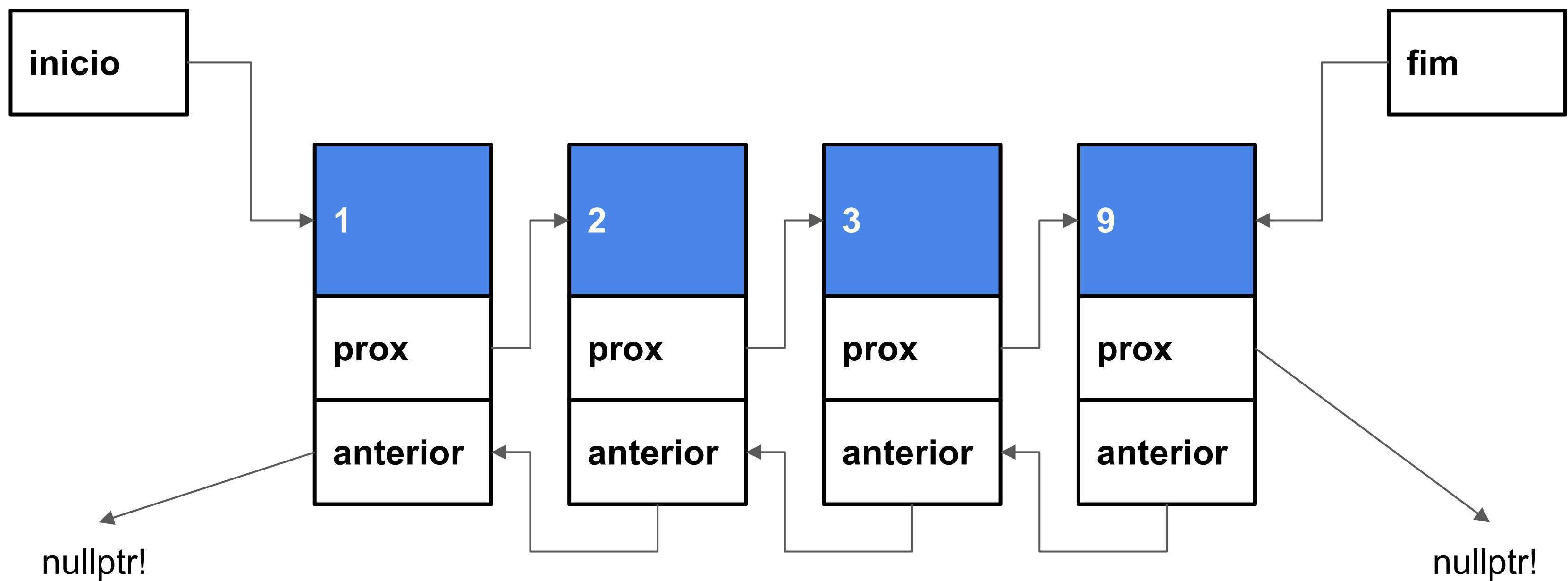
Ideia

$$1238 + 1 = 1239$$



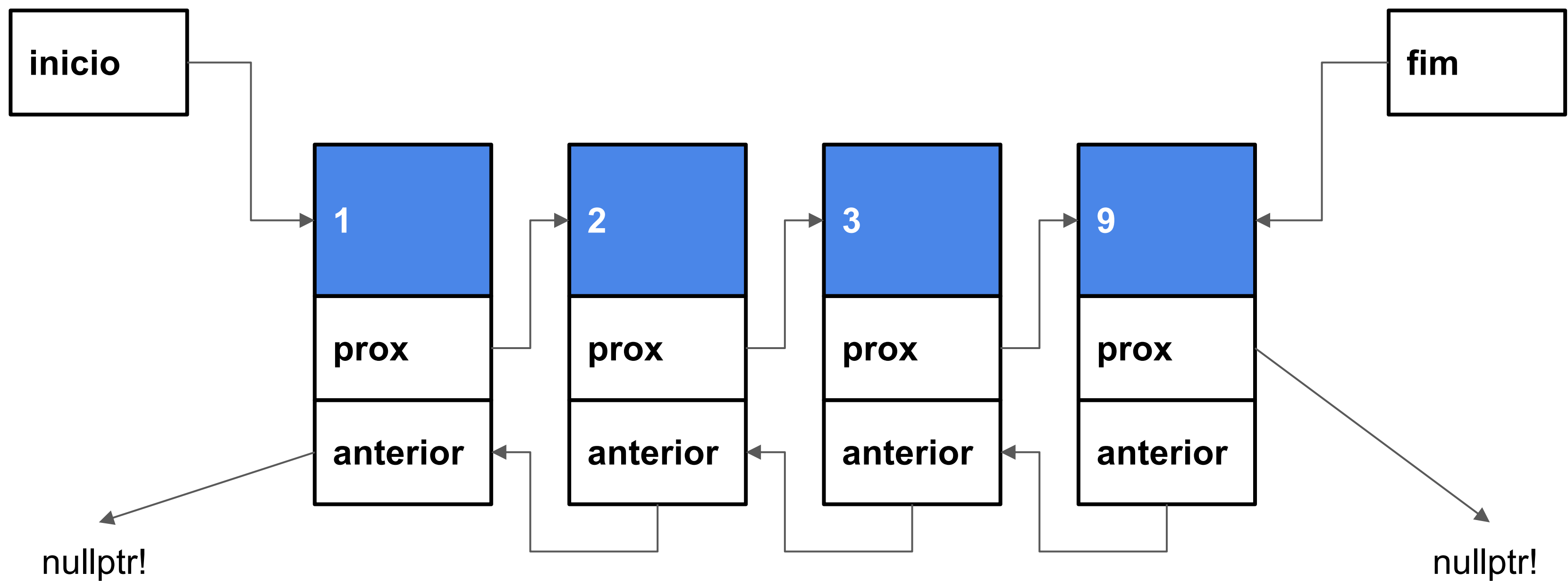
Ideia

$$1238 + 1 = 1239$$



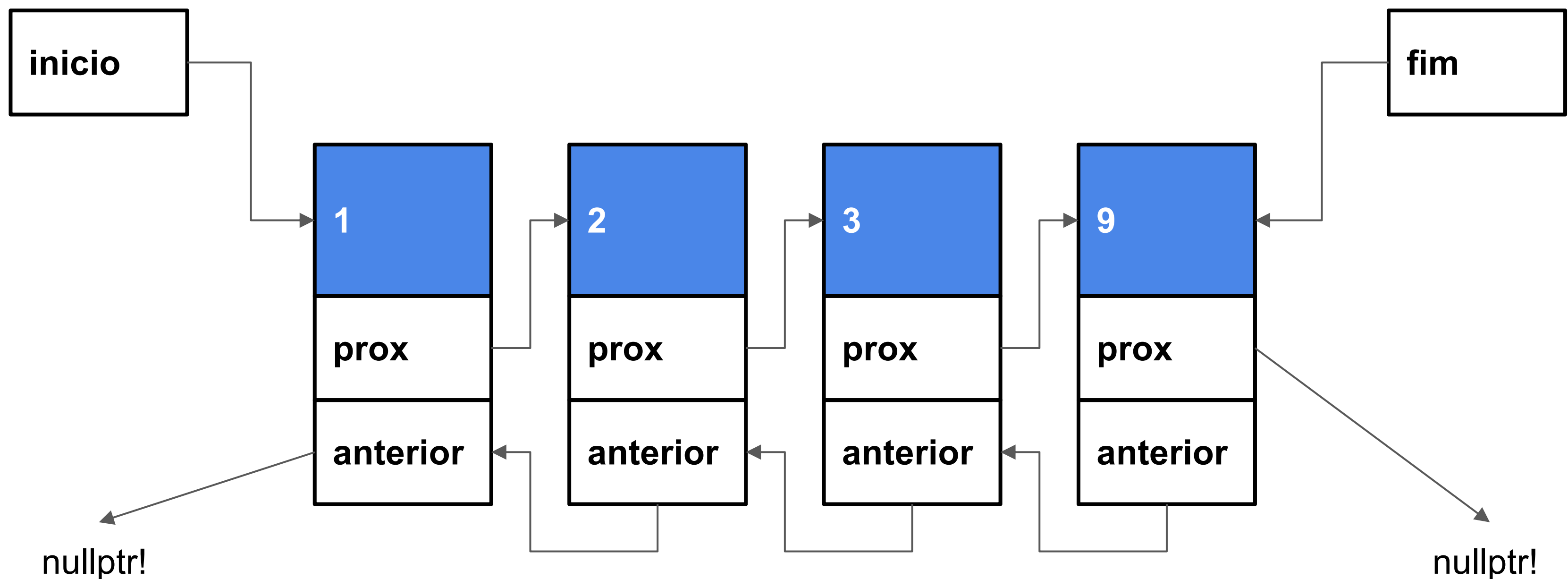
Ideia

$$1239 + 1 = 1240$$



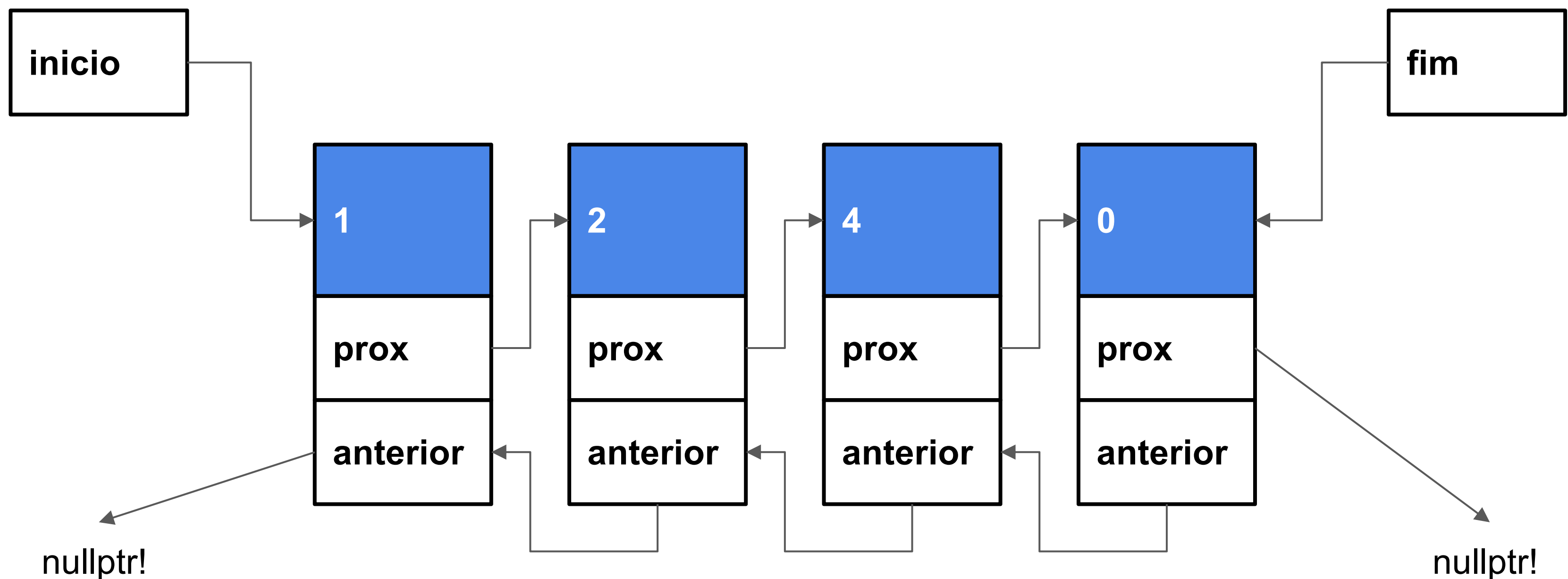
Ideia

vai 1 para frente: zeramos o menor



Ideia

vai 1 para frente: zeramos o menor



TAD

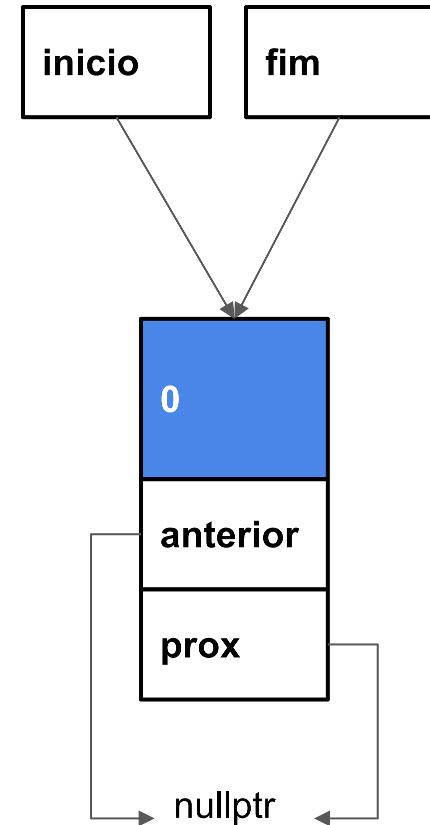
- Note que ainda é uma lista
- Mudamos as operações
- Novo TAD
  - Memória similar
  - Operações diferentes
- Já fizemos o oposto
  - Mesmas operações
  - Mesmo TAD

```
#ifndef PDS2_BIGNUM_H
#define PDS2_BIGNUM_H
struct node_t {
  int valor;
  node_t *anterior;
  node_t *proximo;
};

class BigNum {
private:
  node_t *_inicio;
  node_t *_fim;
public:
  BigNum();
  ~BigNum();
  void incrementa();
  void decrementa();
  void imprimir();
};
#endif
```

Iniciando com 1 dígito zero

```
void BigNum::incrementa() {
  node_t *digito_atual = this->_fim;
  while (1) {
            if (digito_atual->valor < 9) {
              digito_atual->valor += 1;
              break;
            }
            if (digito_atual->anterior == nullptr) {
              digito_atual->anterior = new node_t();
              digito_atual->anterior->proximo = digito_atual;
              this->_inicio = digito_atual->anterior;
      this->_inicio->proximo = digito_atual;
            }
            digito_atual->valor = 0;
    digito_atual->anterior = nullptr;
            digito_atual = digito_atual->anterior;
  }
}
```

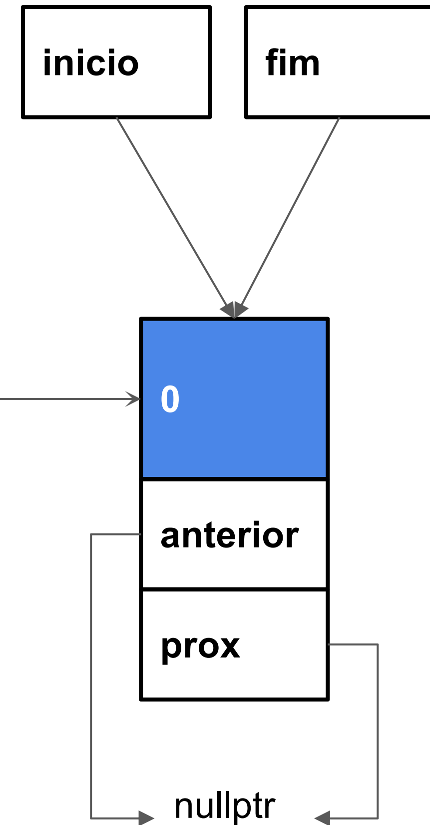
*digito_atual



Iniciando com 1 dígito zero

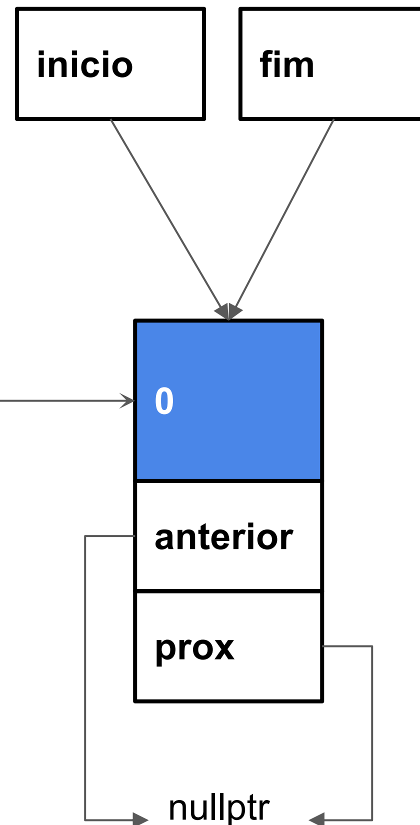
```
void BigNum::incrementa() {
  node_t *digito_atual = this->_fim;
  while (1) {
        if (digito_atual->valor < 9) {
          digito_atual->valor += 1;
          break;
        }
        if (digito_atual->anterior == nullptr) {
          digito_atual->anterior = new node_t();
          digito_atual->anterior->proximo = digito_atual;
          this->_inicio = digito_atual->anterior;
    this->_inicio->proximo = digito_atual;
        }
        digito_atual->valor = 0;
   digito_atual->anterior = nullptr;
        digito_atual = digito_atual->anterior;
  }
}
```

*digito_atual



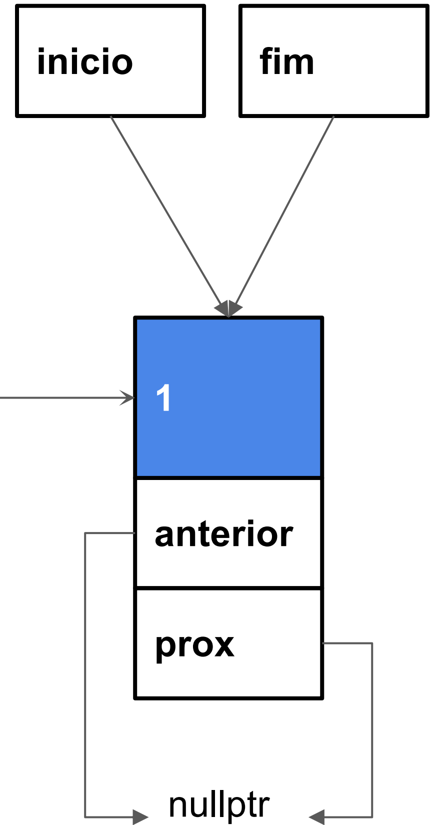
valor < 9?

```
void BigNum::incrementa() {
  node_t *digito_atual = this->_fim;
  while (1) {
        if (digito_atual->valor < 9) {
          digito_atual->valor += 1;
          break;
        }
        if (digito_atual->anterior == nullptr) {
          digito_atual->anterior = new node_t();
          digito_atual->anterior->proximo = digito_atual;
          this->_inicio = digito_atual->anterior;
     this->_inicio->proximo = digito_atual;
        }
        digito_atual->valor = 0;
    digito_atual->anterior = nullptr;
        digito_atual = digito_atual->anterior;
  }
}
```



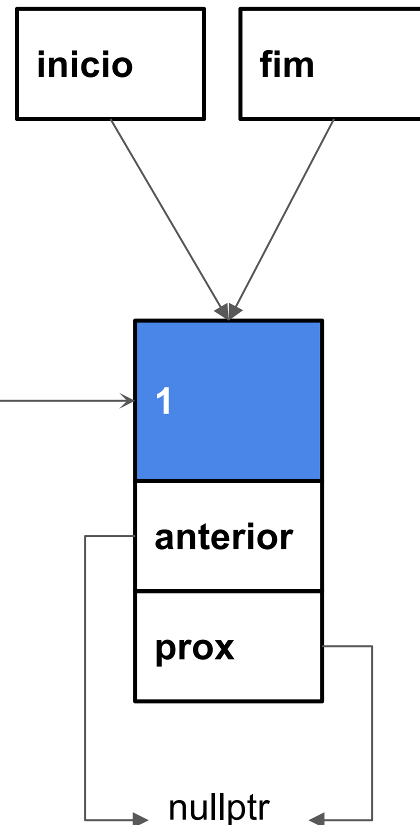
ok → incrementa

```
void BigNum::incrementa() {
  node_t *digito_atual = this->_fim;
  while (1) {
          if (digito_atual->valor < 9) {
            digito_atual->valor += 1;
            break;
          }
          if (digito_atual->anterior == nullptr) {
            digito_atual->anterior = new node_t();
            digito_atual->anterior->proximo = digito_atual;
            this->_inicio = digito_atual->anterior;
     this->_inicio->proximo = digito_atual;
          }
          digito_atual->valor = 0;
    digito_atual->anterior = nullptr;
          digito_atual = digito_atual->anterior;
  }
}
```



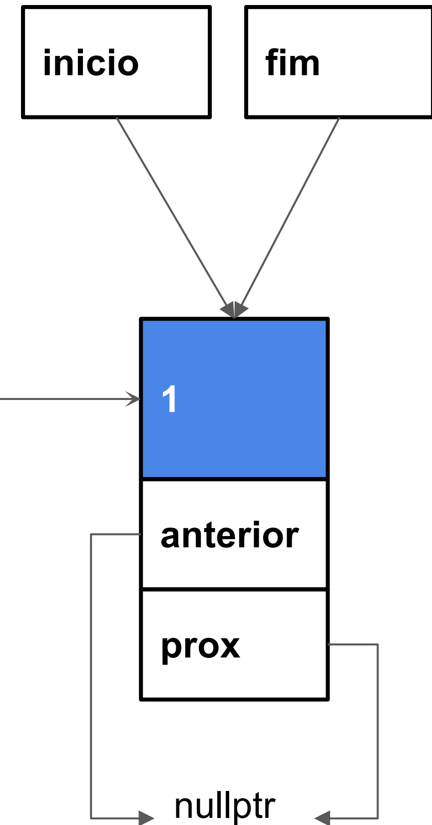
vamos embora!

```
void BigNum::incrementa() {
  node_t *digito_atual = this->_fim;
  while (1) {
            if (digito_atual->valor < 9) {
              digito_atual->valor += 1;
              break;
            }
            if (digito_atual->anterior == nullptr) {
              digito_atual->anterior = new node_t();
              digito_atual->anterior->proximo = digito_atual;
              this->_inicio = digito_atual->anterior;
      this->_inicio->proximo = digito_atual;
            }
            digito_atual->valor = 0;
    digito_atual->anterior = nullptr;
            digito_atual = digito_atual->anterior;
  }
}
```



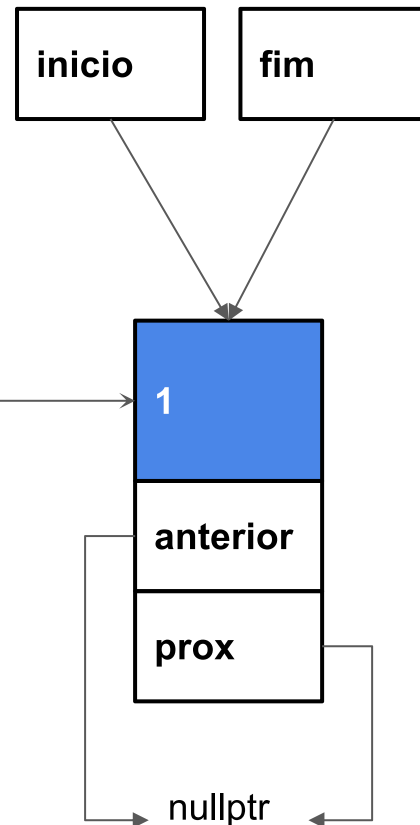
chamando +1 vez a função incrementa

```
void BigNum::incrementa() {
  node_t *digito_atual = this->_fim;
  while (1) {
            if (digito_atual->valor < 9) {
              digito_atual->valor += 1;
              break;
            }
            if (digito_atual->anterior == nullptr) {
              digito_atual->anterior = new node_t();
              digito_atual->anterior->proximo = digito_atual;
              this->_inicio = digito_atual->anterior;
      this->_inicio->proximo = digito_atual;
            }
            digito_atual->valor = 0;
    digito_atual->anterior = nullptr;
            digito_atual = digito_atual->anterior;
  }
}
```



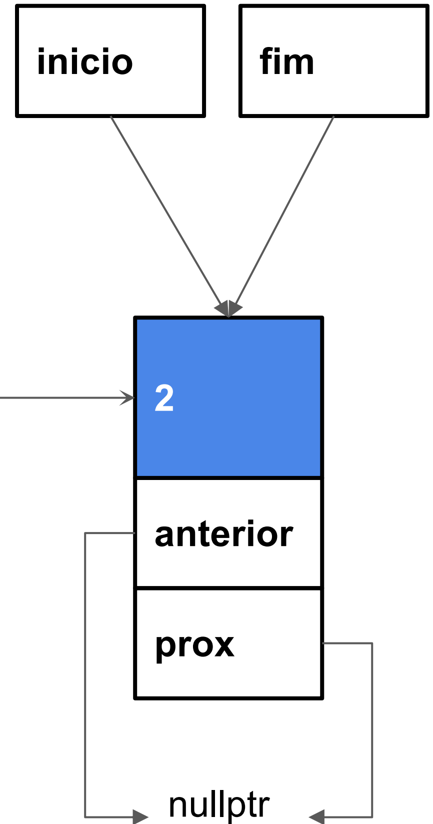
valor < 9?

```
void BigNum::incrementa() {
  node_t *digito_atual = this->_fim;
  while (1) {
        if (digito_atual->valor < 9) {
          digito_atual->valor += 1;
          break;
        }
        if (digito_atual->anterior == nullptr) {
          digito_atual->anterior = new node_t();
          digito_atual->anterior->proximo = digito_atual;
          this->_inicio = digito_atual->anterior;
     this->_inicio->proximo = digito_atual;
        }
        digito_atual->valor = 0;
    digito_atual->anterior = nullptr;
        digito_atual = digito_atual->anterior;
  }
}
```



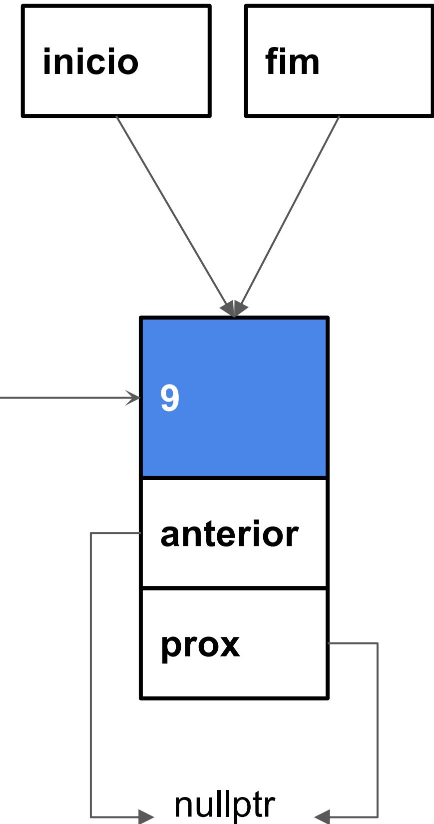
ok → incrementa

```
void BigNum::incrementa() {
  node_t *digito_atual = this->_fim;
  while (1) {
        if (digito_atual->valor < 9) {
          digito_atual->valor += 1;
          break;
        }
        if (digito_atual->anterior == nullptr) {
          digito_atual->anterior = new node_t();
          digito_atual->anterior->proximo = digito_atual;
          this->_inicio = digito_atual->anterior;
    this->_inicio->proximo = digito_atual;
        }
        digito_atual->valor = 0;
    digito_atual->anterior = nullptr;
        digito_atual = digito_atual->anterior;
  }
}
```



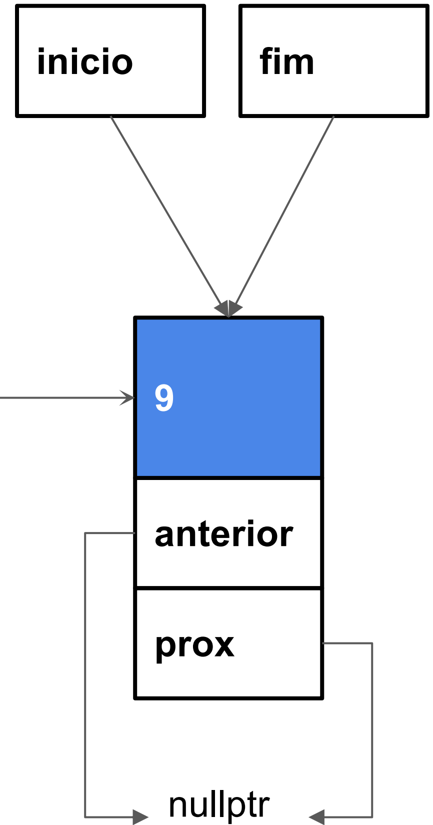
depois de 9 chamadas!

```
void BigNum::incrementa() {
  node_t *digito_atual = this->_fim;
  while (1) {
          if (digito_atual->valor < 9) {
            digito_atual->valor += 1;
            break;
          }
          if (digito_atual->anterior == nullptr) {
            digito_atual->anterior = new node_t();
            digito_atual->anterior->proximo = digito_atual;
            this->_inicio = digito_atual->anterior;
      this->_inicio->proximo = digito_atual;
          }
          digito_atual->valor = 0;
    digito_atual->anterior = nullptr;
          digito_atual = digito_atual->anterior;
  }
}
```



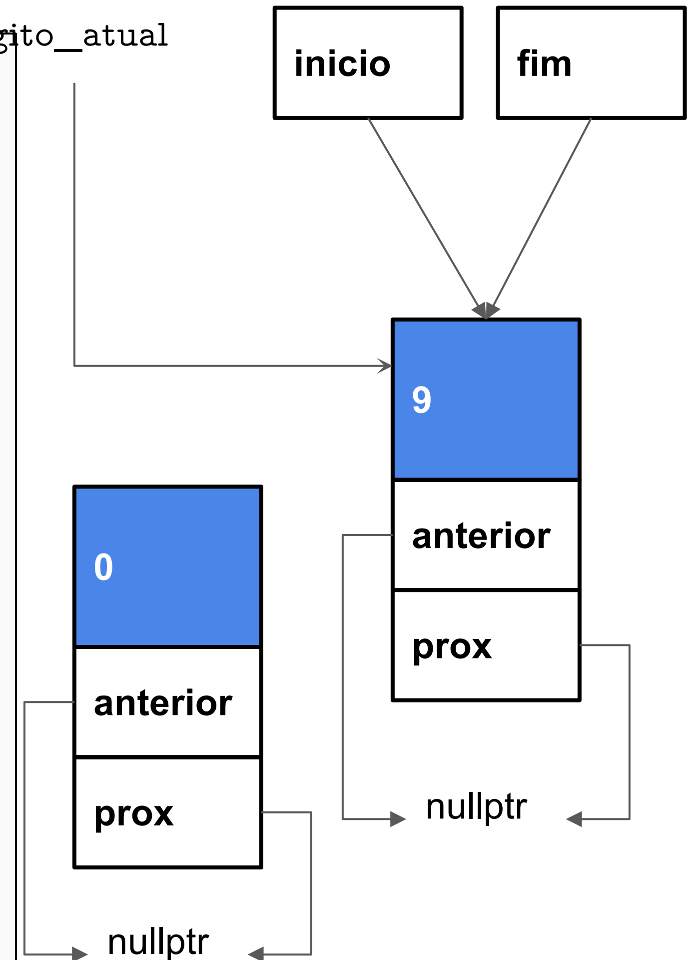
< 9 → false

```
void BigNum::incrementa() {
  node_t *digito_atual = this->_fim;
  while (1) {
            if (digito_atual->valor < 9) {
              digito_atual->valor += 1;
              break;
            }
            if (digito_atual->anterior == nullptr) {
              digito_atual->anterior = new node_t();
              digito_atual->anterior->proximo = digito_atual;
              this->_inicio = digito_atual->anterior;
      this->_inicio->proximo = digito_atual;
            }
            digito_atual->valor = 0;
    digito_atual->anterior = nullptr;
            digito_atual = digito_atual->anterior;
  }
}
```



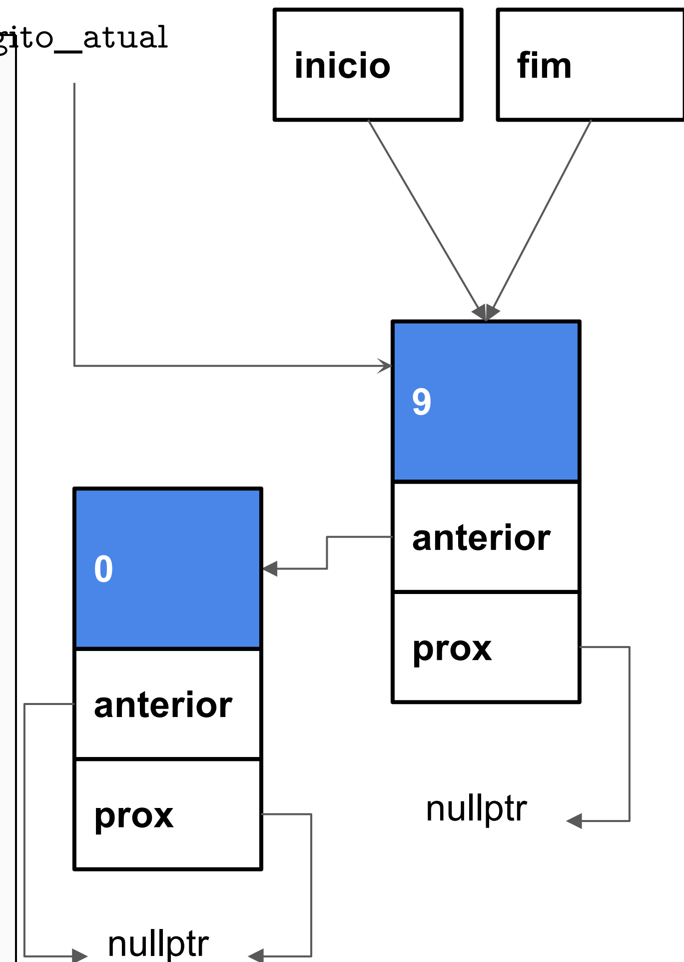
novo dígito

```
void BigNum::incrementa() {
  node_t *digito_atual = this->_fim;
  while (1) {
            if (digito_atual->valor < 9) {
              digito_atual->valor += 1;
              break;
            }
            if (digito_atual->anterior == nullptr) {
              digito_atual->anterior = new node_t();
              digito_atual->anterior->proximo = digito_atual;
              this->_inicio = digito_atual->anterior;
      this->_inicio->proximo = digito_atual;
            }
            digito_atual->valor = 0;
    digito_atual->anterior = nullptr;
            digito_atual = digito_atual->anterior;
  }
}
```



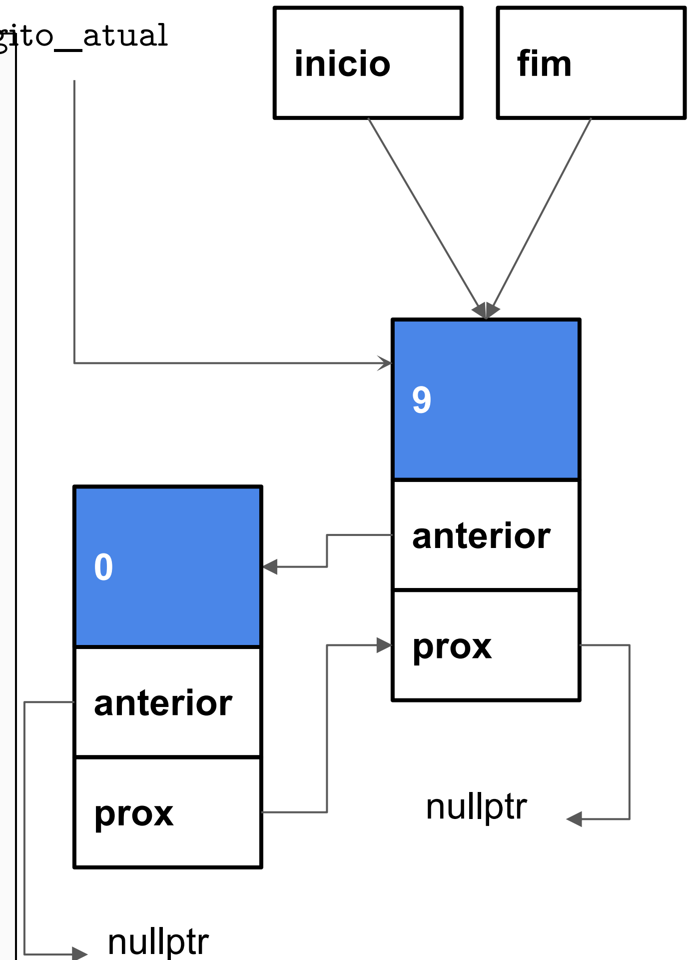
próximo do atual

```
void BigNum::incrementa() {
  node_t *digito_atual = this->_fim;
  while (1) {
          if (digito_atual->valor < 9) {
            digito_atual->valor += 1;
            break;
          }
          if (digito_atual->anterior == nullptr) {
            digito_atual->anterior = new node_t();
            digito_atual->anterior->proximo = digito_atual;
            this->_inicio = digito_atual->anterior;
    this->_inicio->proximo = digito_atual;
          }
          digito_atual->valor = 0;
    digito_atual->anterior = nullptr;
          digito_atual = digito_atual->anterior;
  }
}
```



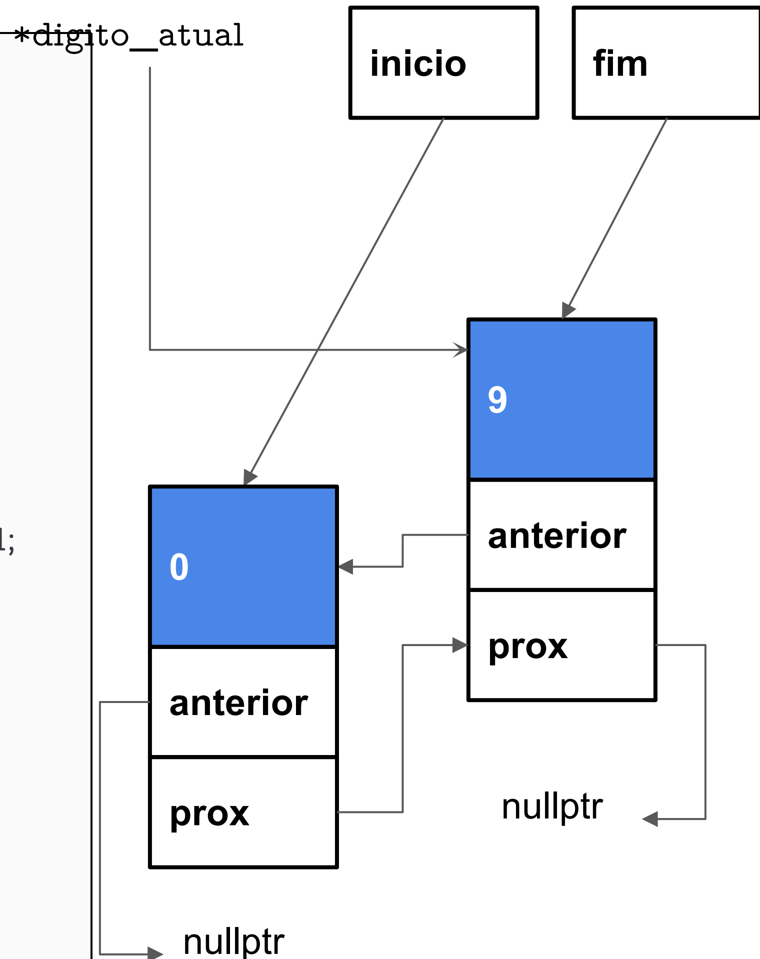
tem como anterior o atual

```
void BigNum::incrementa() {
  node_t *digito_atual = this->_fim;
  while (1) {
            if (digito_atual->valor < 9) {
              digito_atual->valor += 1;
              break;
            }
            if (digito_atual->anterior == nullptr) {
              digito_atual->anterior = new node_t();
              digito_atual->anterior->proximo = digito_atual;
              this->_inicio = digito_atual->anterior;
       this->_inicio->proximo = digito_atual;
            }
            digito_atual->valor = 0;
     digito_atual->anterior = nullptr;
            digito_atual = digito_atual->anterior;
  }
}
```



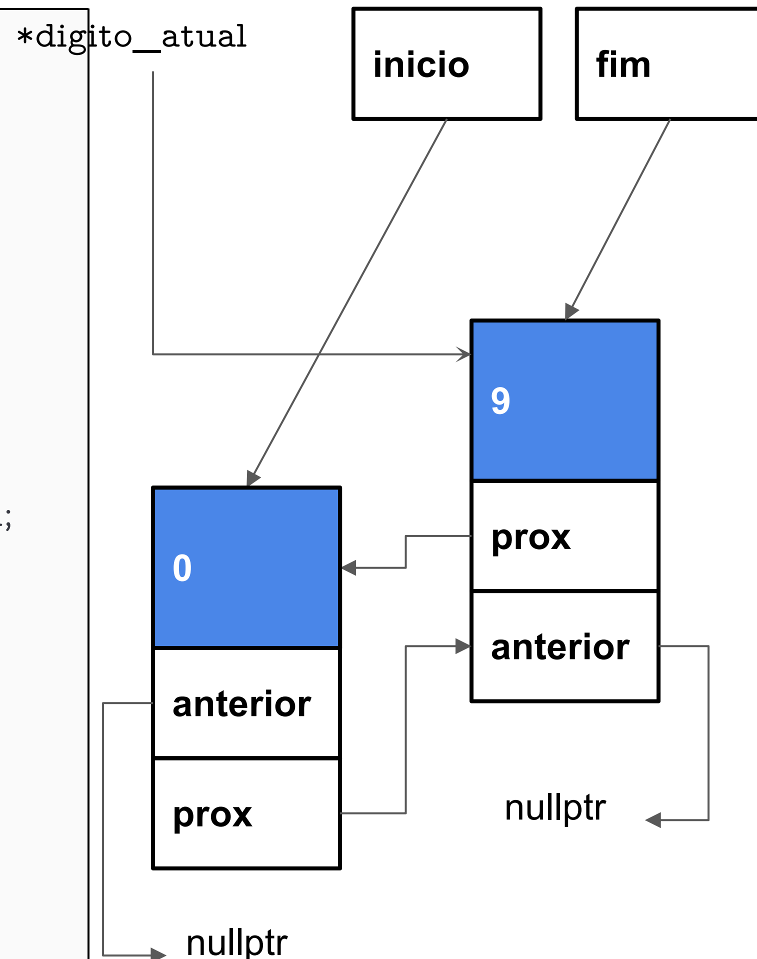
atualizamos o início

```
void BigNum::incrementa() {
  node_t *digito_atual = this->_fim;
  while (1) {
          if (digito_atual->valor < 9) {
            digito_atual->valor += 1;
            break;
          }
          if (digito_atual->anterior == nullptr) {
            digito_atual->anterior = new node_t();
            digito_atual->anterior->proximo = digito_atual;
            this->_inicio = digito_atual->anterior;
      this->_inicio->proximo = digito_atual;
          }
          digito_atual->valor = 0;
    digito_atual->anterior = nullptr;
          digito_atual = digito_atual->anterior;
  }
}
```



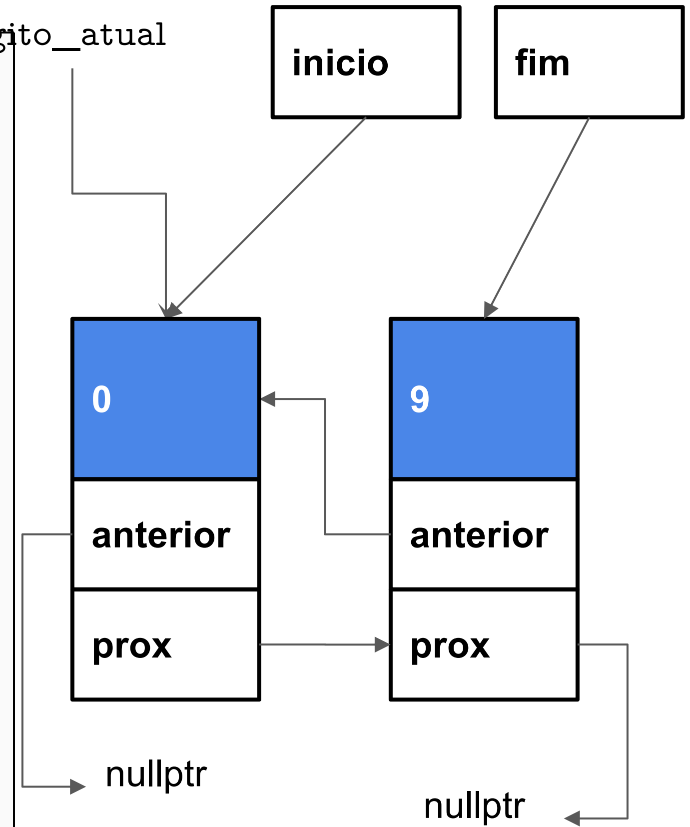
zeramos o novo atual

```
void BigNum::incrementa() {
  node_t *digito_atual = this->_fim;
  while (1) {
        if (digito_atual->valor < 9) {
          digito_atual->valor += 1;
          break;
        }
        if (digito_atual->anterior == nullptr) {
          digito_atual->anterior = new node_t();
          digito_atual->anterior->proximo = digito_atual;
          this->_inicio = digito_atual->anterior;
    this->_inicio->proximo = digito_atual;
        }
        digito_atual->valor = 0;
    digito_atual->anterior = nullptr;
        digito_atual = digito_atual->anterior;
  }
}
```



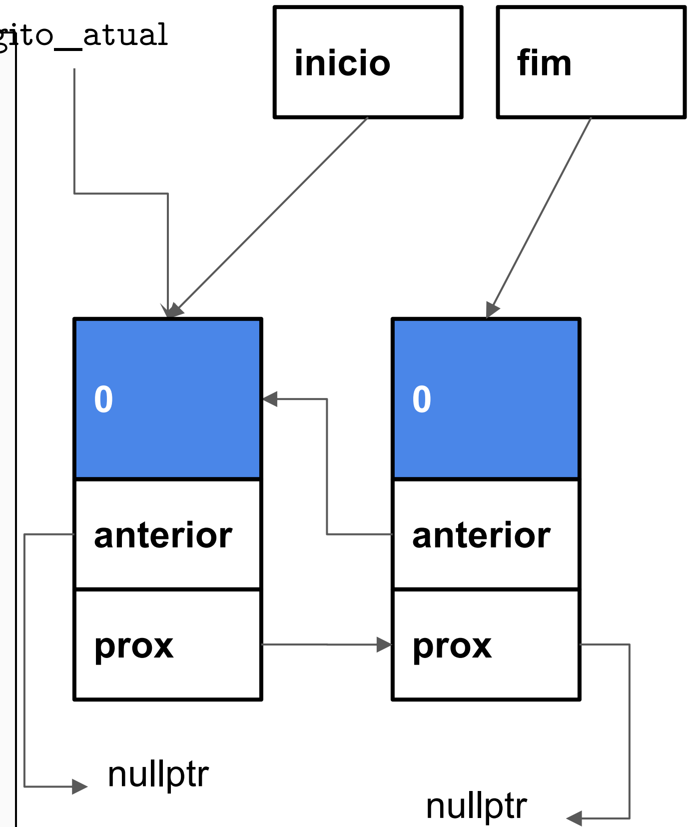
caminha pra frente

```
void BigNum::incrementa() {
  node_t *digito_atual = this->_fim;
  while (1) {
        if (digito_atual->valor < 9) {
          digito_atual->valor += 1;
          break;
        }
        if (digito_atual->anterior == nullptr) {
          digito_atual->anterior = new node_t();
          digito_atual->anterior->proximo = digito_atual;
          this->_inicio = digito_atual->anterior;
    this->_inicio->proximo = digito_atual;
        }
        digito_atual->valor = 0;
    digito_atual->anterior = nullptr;
        digito_atual = digito_atual->anterior;
  }
}
```



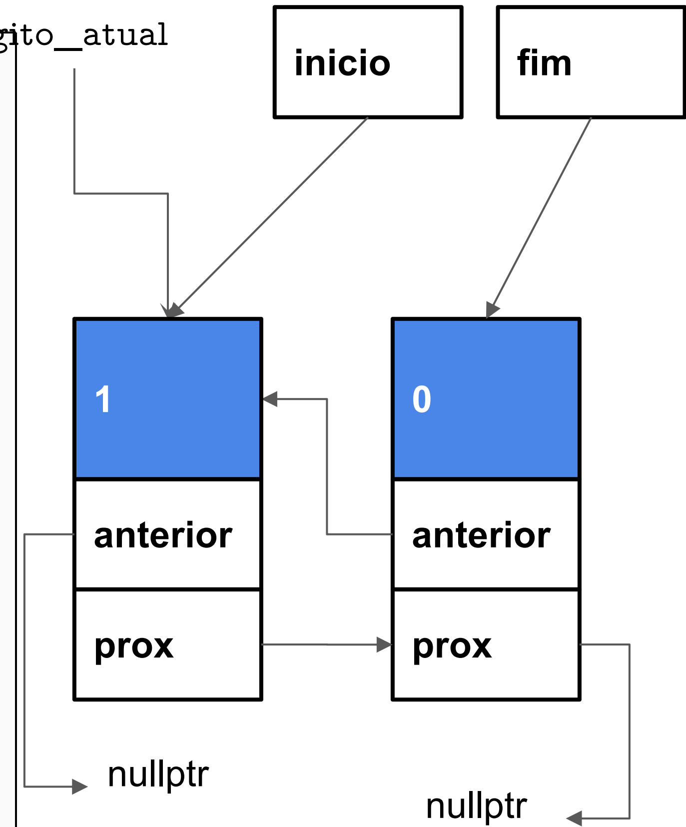
e agora?

```
void BigNum::incrementa() {
  node_t *digito_atual = this->_fim;
  while (1) {
          if (digito_atual->valor < 9) {
            digito_atual->valor += 1;
            break;
          }
          if (digito_atual->anterior == nullptr) {
            digito_atual->anterior = new node_t();
            digito_atual->anterior->proximo = digito_atual;
            this->_inicio = digito_atual->anterior;
      this->_inicio->proximo = digito_atual;
          }
          digito_atual->valor = 0;
    digito_atual->anterior = nullptr;
          digito_atual = digito_atual->anterior;
  }
}
```



chegamos no 10! note o while(1)

```
void BigNum::incrementa() {
  node_t *digito_atual = this->_fim;
  while (1) {
          if (digito_atual->valor < 9) {
            digito_atual->valor += 1;
            break;
          }
          if (digito_atual->anterior == nullptr) {
            digito_atual->anterior = new node_t();
            digito_atual->anterior->proximo = digito_atual;
            this->_inicio = digito_atual->anterior;
      this->_inicio->proximo = digito_atual;
          }
          digito_atual->valor = 0;
    digito_atual->anterior = nullptr;
          digito_atual = digito_atual->anterior;
  }
}
```

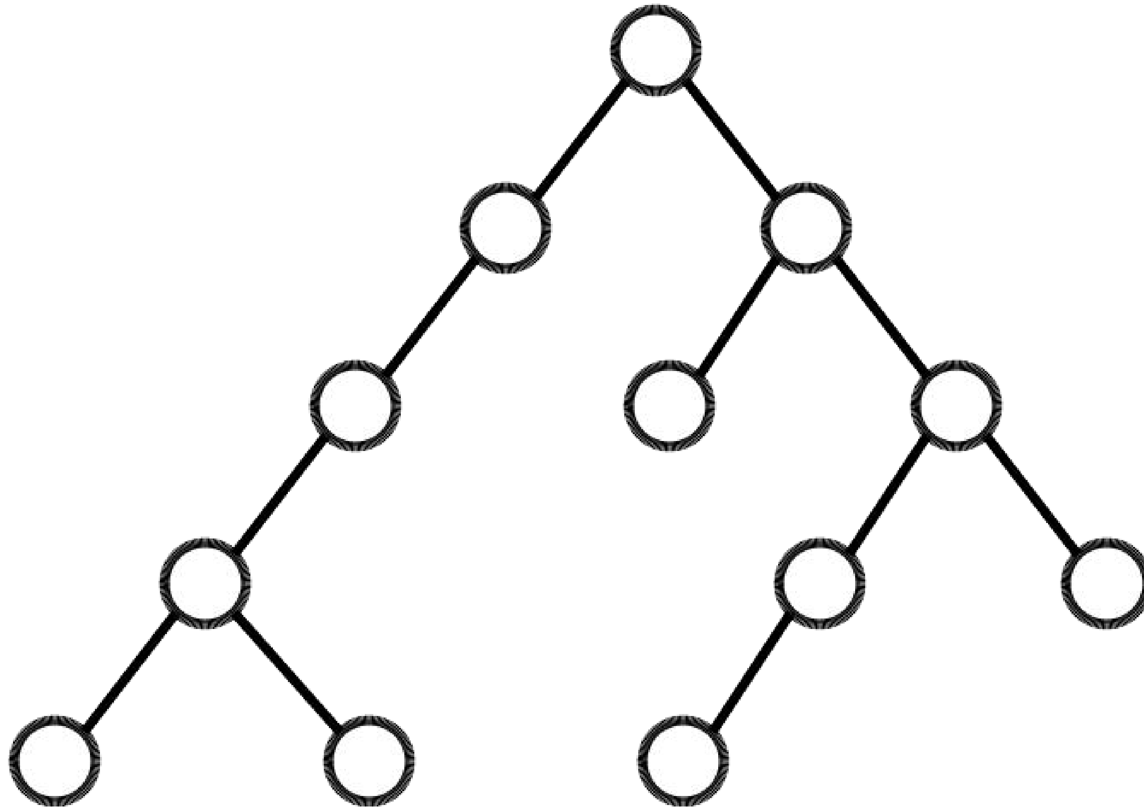


BigNum com lista dupla

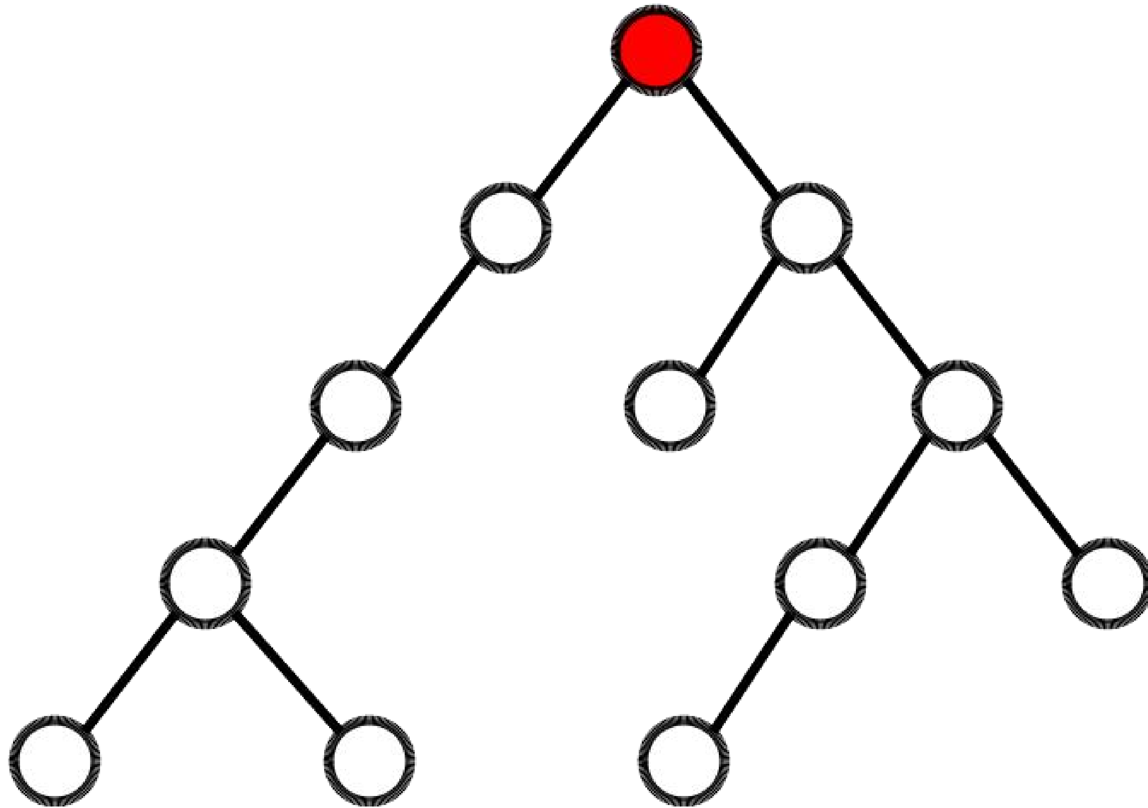
- Para imprimir $\rightarrow$
- Para realizar operações $\leftarrow$

Árvores

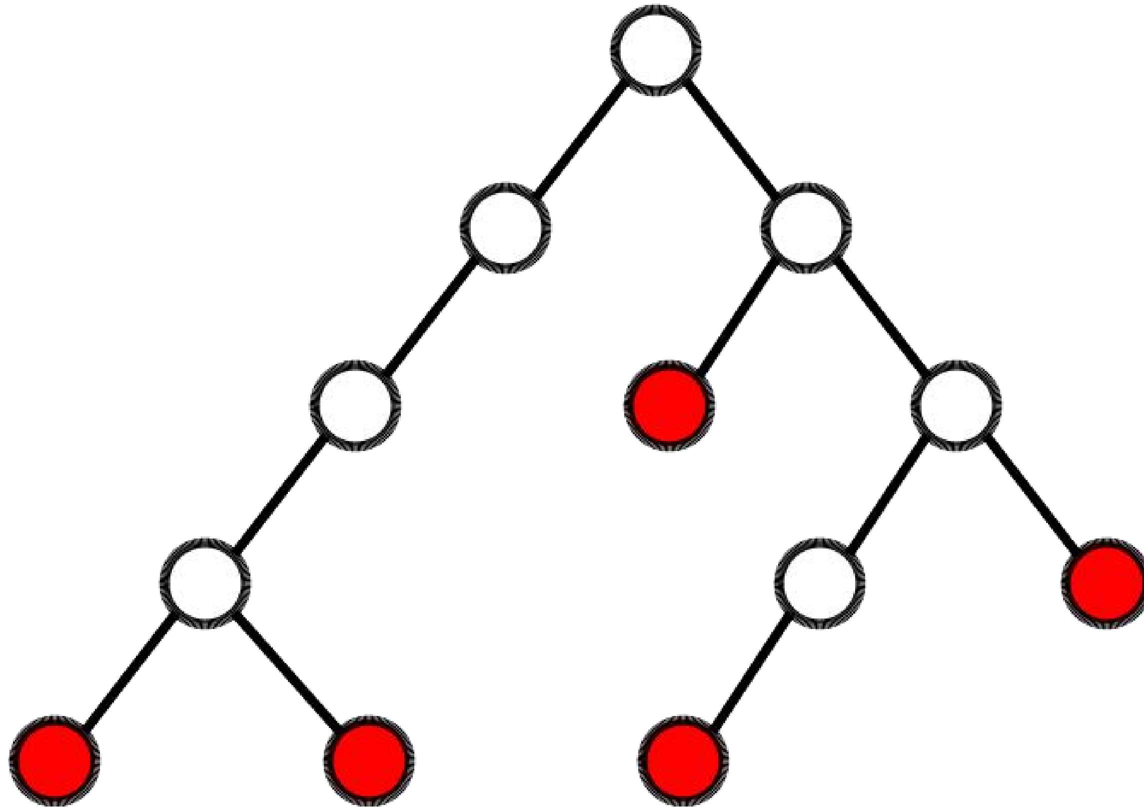
Árvores



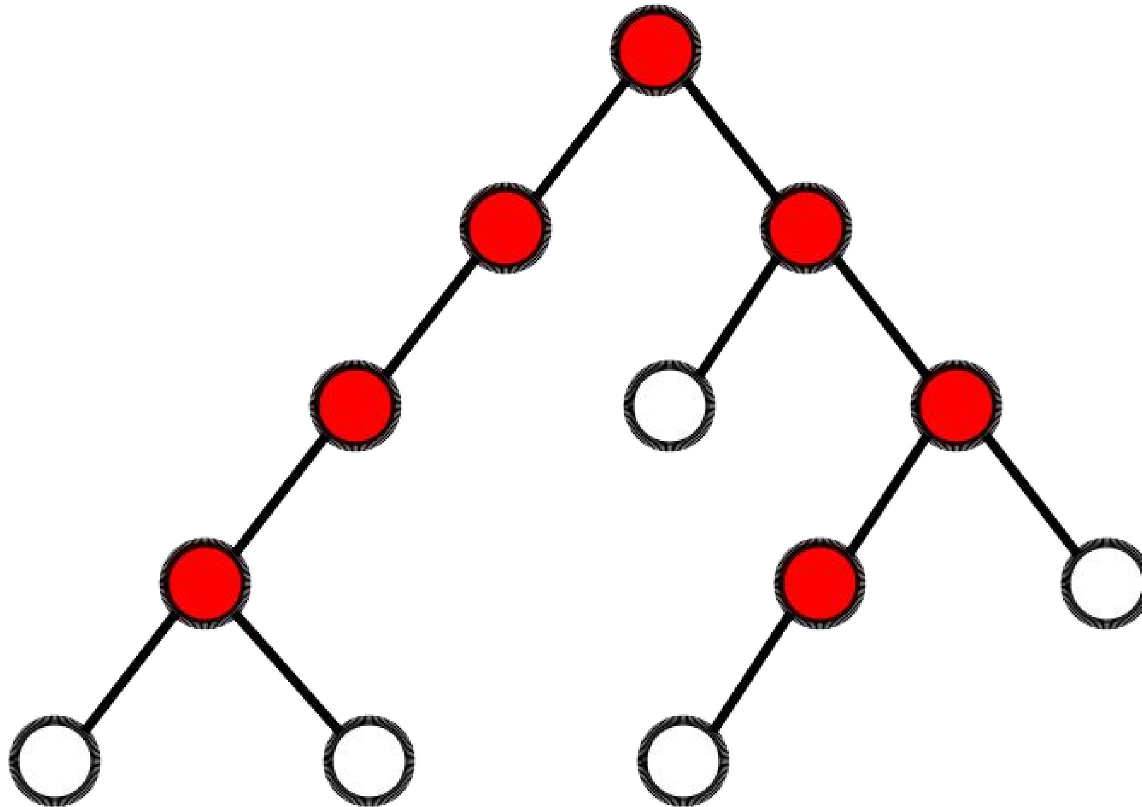
Raíz



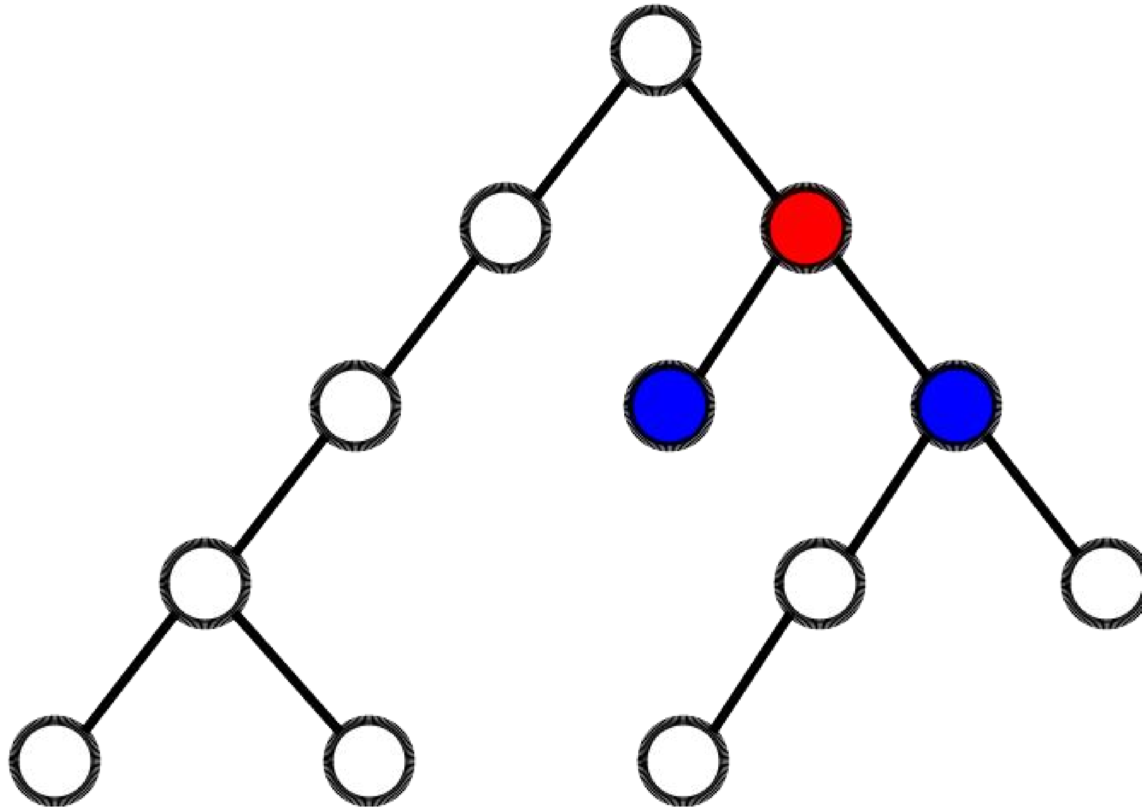
Folhas



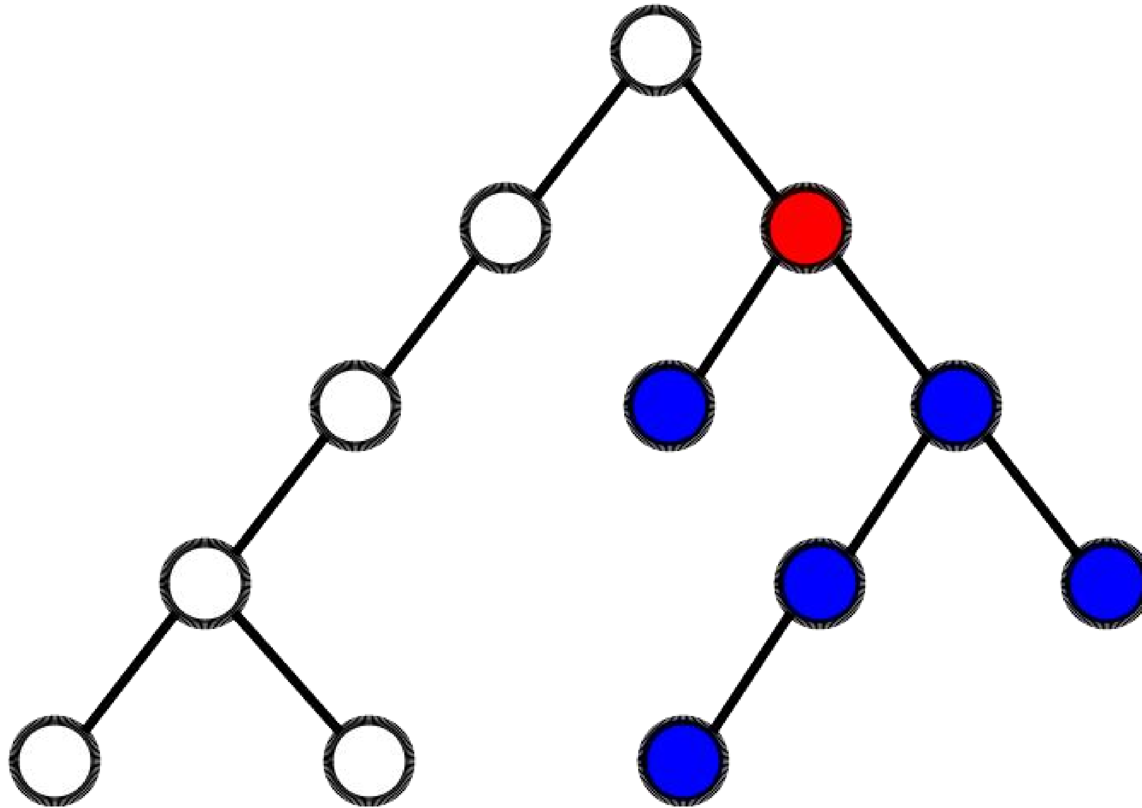
Nós Internos



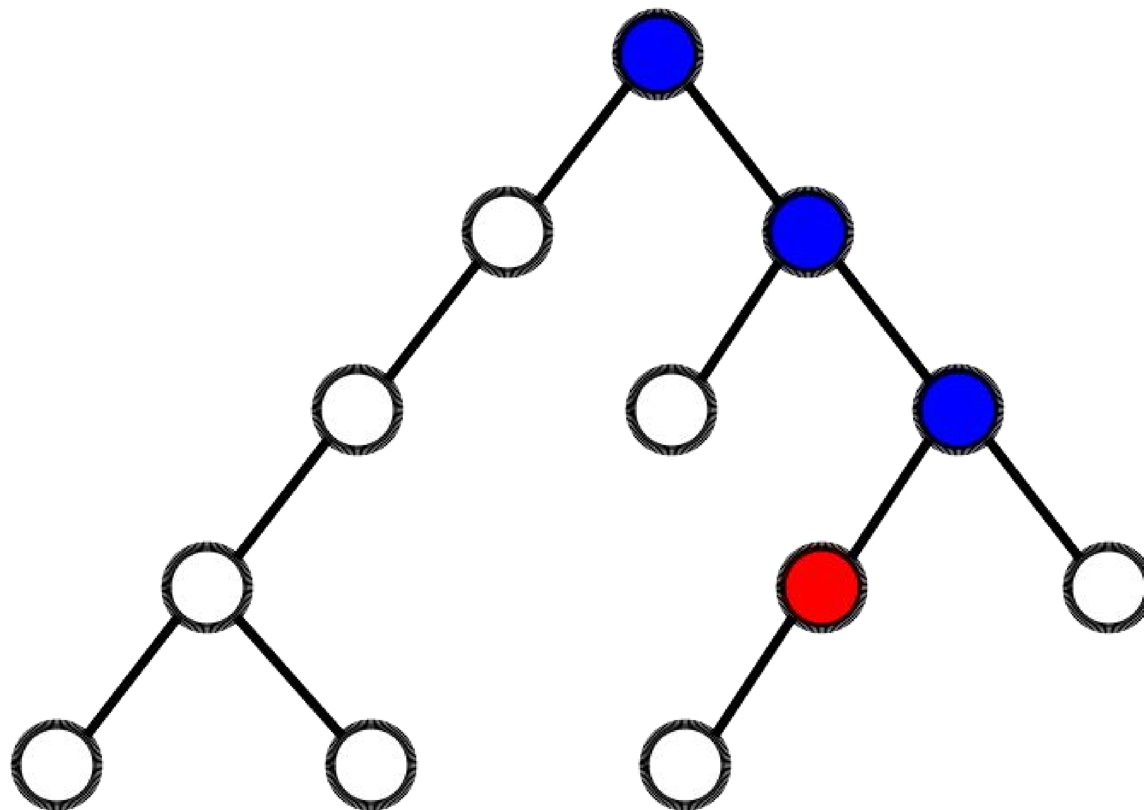
Pais e Filhos



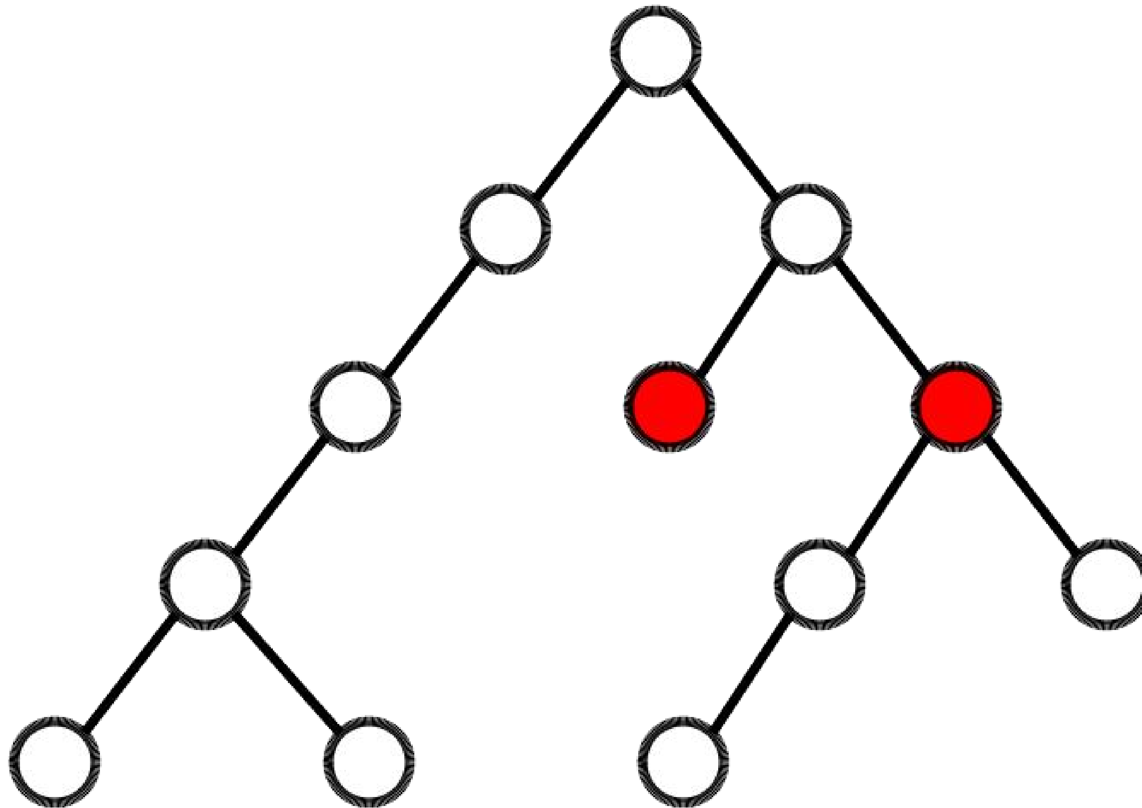
Descendentes



Ancestrais

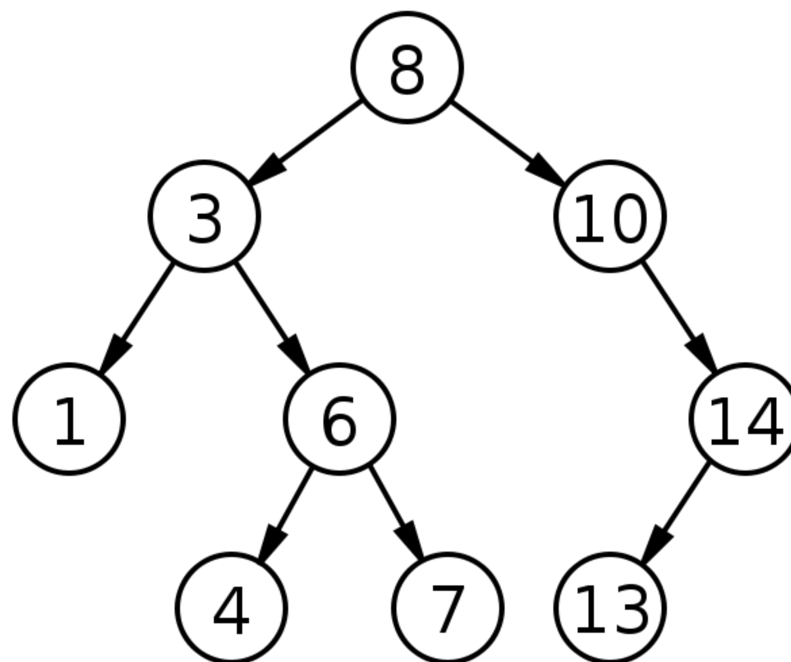


Irmãos



Árvore Binária

Apenas 2 descendentes imediatos por nó

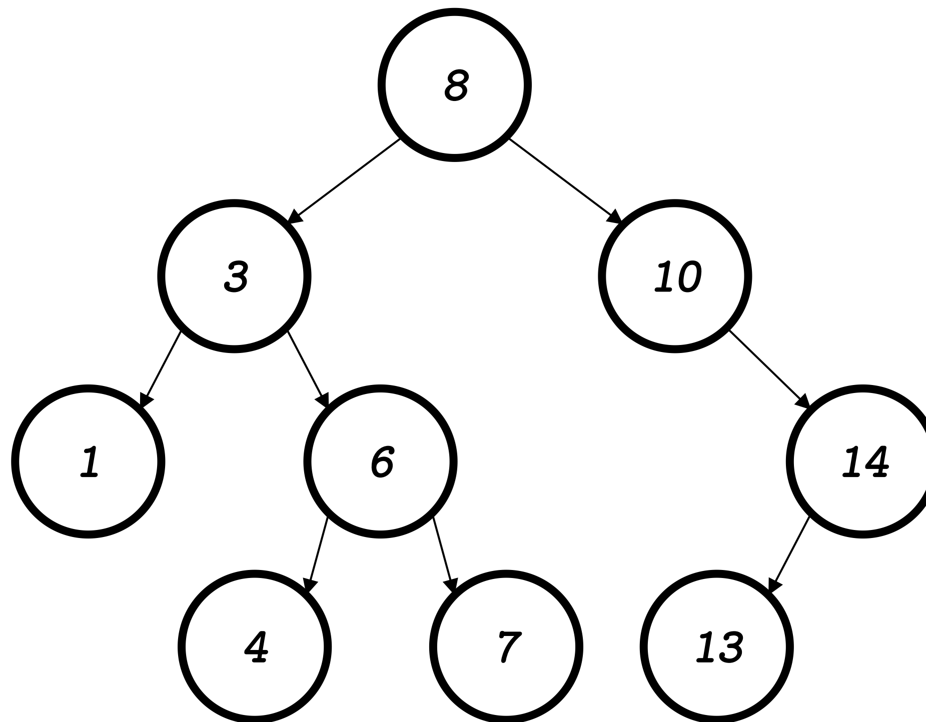


Árvore Binária de Pesquisa/Busca (Binary Search Tree - BST)

- Invariantes:
  - O filho da esquerda é menor ou igual ao nó
  - O filho da direita é maior do que o nó
- Consequências:
  - Elementos na esquerda são menores
  - Elementos na direita são maiores

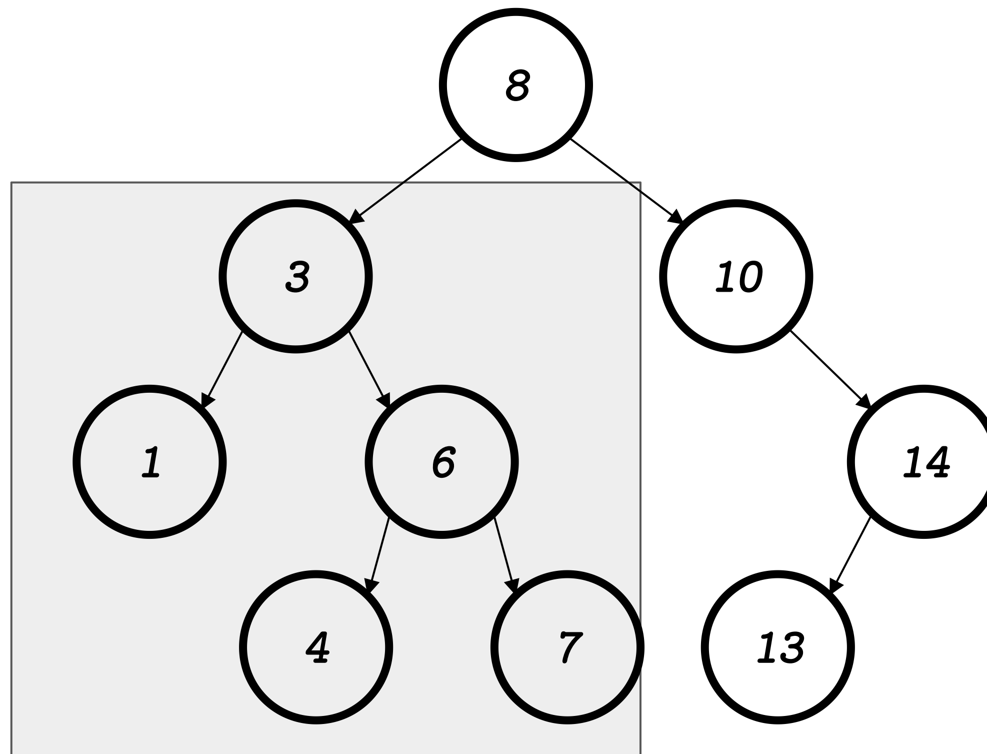
Implementando árvores

Qual o segredo?



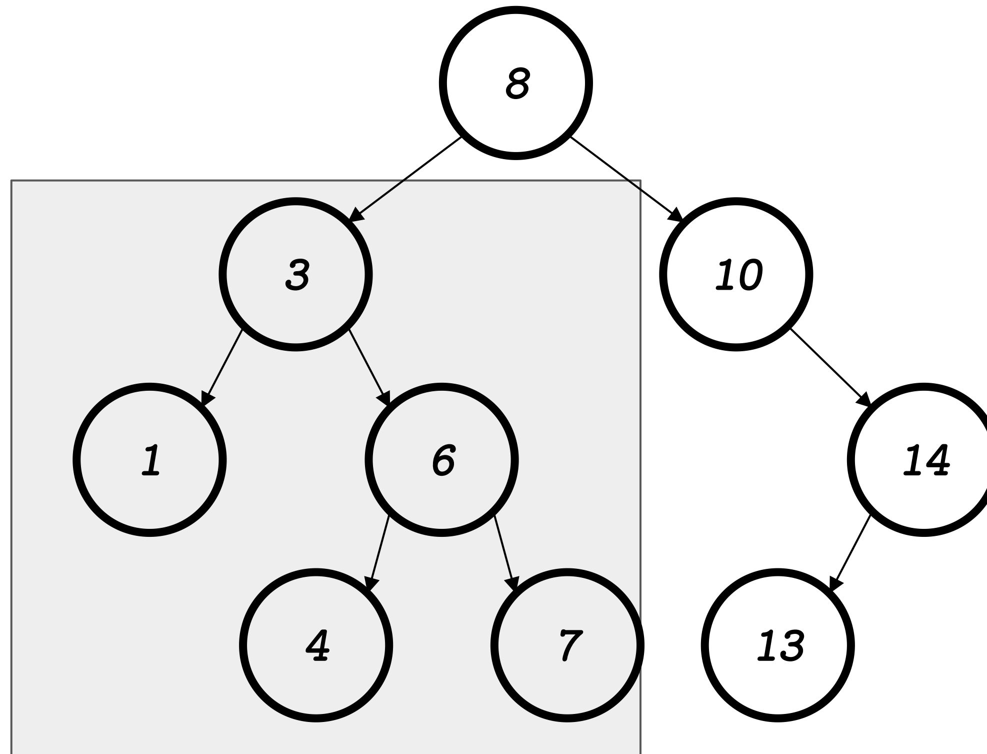
Implementando árvores

Todo nó representa uma árvore!



Implementando árvores

Vamos implementar no nó



.h da classe Node

```
#ifndef PDS2_NODE_H
#define PDS2_NODE_H
class Node {
private:
  Node *_esquerda;
  Node *_direita;
  int _elemento;
public:
  Node(int valor);
  ~Node();
  // insere elemento abaixo do nó atual
  void inserir_elemento(int elemento);
  // verifica se o inteiro existe na sub-árvore
  bool tem_elemento(int elemento);
  // imprime a árvore deste nó para baixo
  void imprimir();
};
#endif
```

Diferente do struct_t, contém métodos

```
#ifndef PDS2_NODE_H
#define PDS2_NODE_H
class Node {
private:
  Node *_esquerda;
  Node *_direita;
  int _elemento;
public:
  Node(int valor);
  ~Node();
  // insere elemento abaixo do nó atual
  void inserir_elemento(int elemento);
  // verifica se o inteiro existe na sub-árvore
  bool tem_elemento(int elemento);
  // imprime a árvore deste nó para baixo
  void imprimir();
};
#endif
```

Ponteiros para esquera e direita

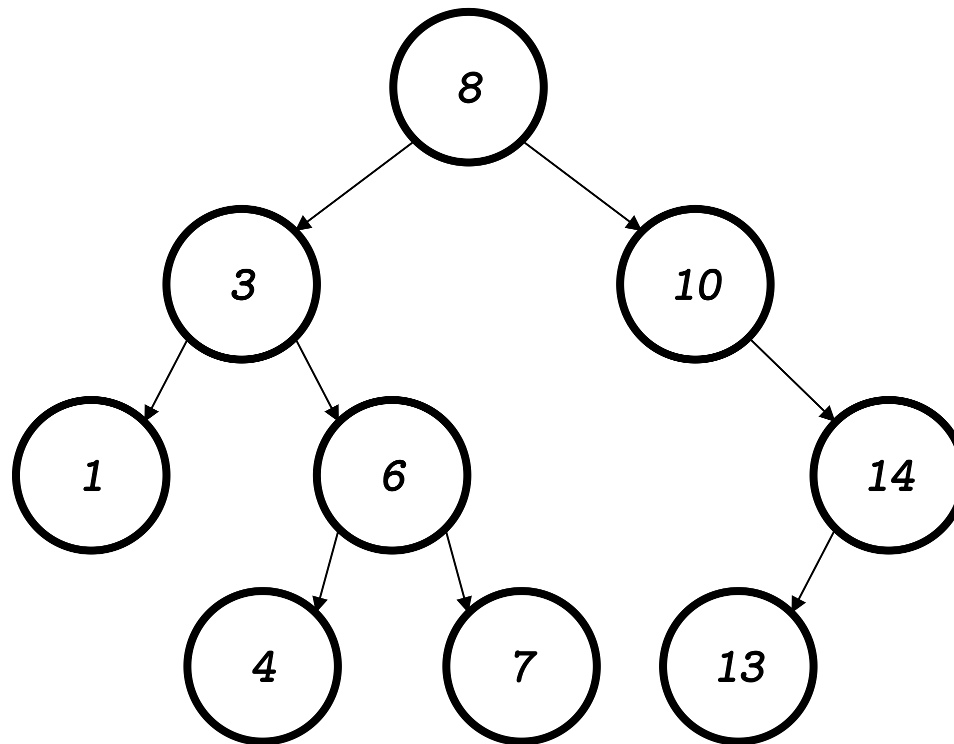
Métodos públicos

Construtor e Destrutor

- Note que o destrutor vai destruir os filhos.
  Os mesmos destroem os netos...

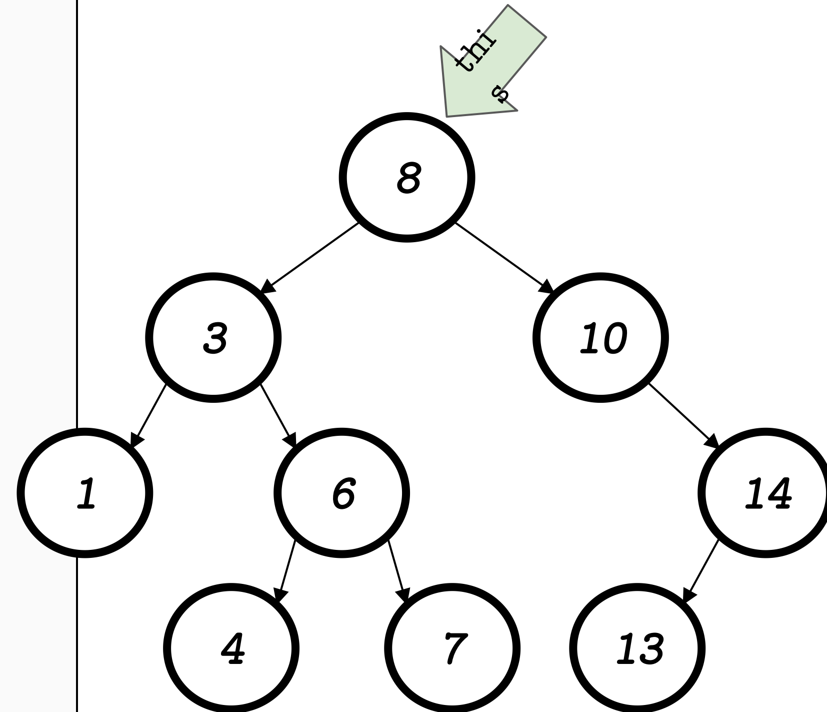
```
Node::Node(int elemento) {
  this->_elemento = elemento;
  this->_esquerda = nullptr;
  this->_direita = nullptr;
}

Node::~Node() {
  if (this->_esquerda != nullptr)
    delete this->_esquerda;
  if (this->_direita != nullptr)
    delete this->_direita;
}
```

Como achar um valor?



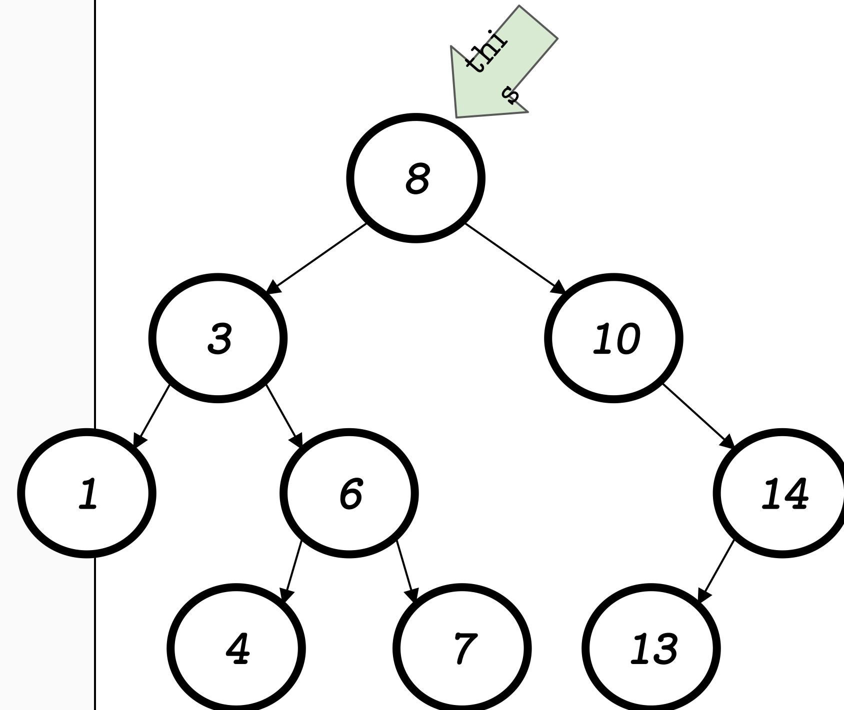
x = 4

```
bool Node::tem_elemento(int elemento) {
  if (this->_elemento == elemento) {
    return true;
  } else if (elemento < this->_elemento) {
    if (this->_esquerda == nullptr) {
      return false;
    } else {
      return this->_esquerda->tem_elemento(elemento);
    }
  } else {
    if (this->_direita == nullptr) {
      return false;
    } else {
      return this->_direita->tem_elemento(elemento);
    }
  }
}
```



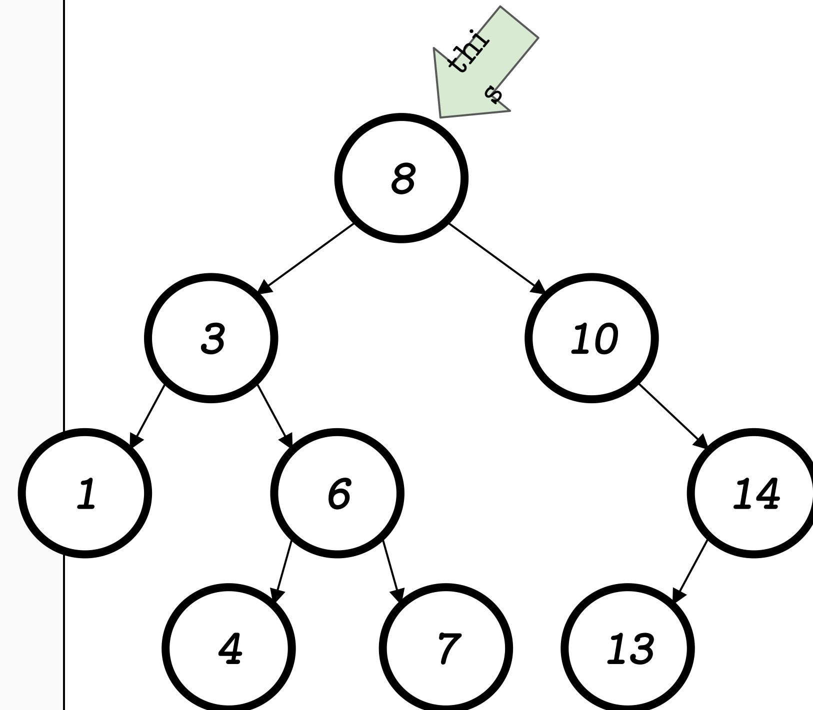
não achamos

```
bool Node::tem_elemento(int elemento) {
  if (this->_elemento == elemento) {
    return true;
  } else if (elemento < this->_elemento) {
    if (this->_esquerda == nullptr) {
      return false;
    } else {
      return this->_esquerda->tem_elemento(elemento);
    }
  } else {
    if (this->_direita == nullptr) {
      return false;
    } else {
      return this->_direita->tem_elemento(elemento);
    }
  }
}
```



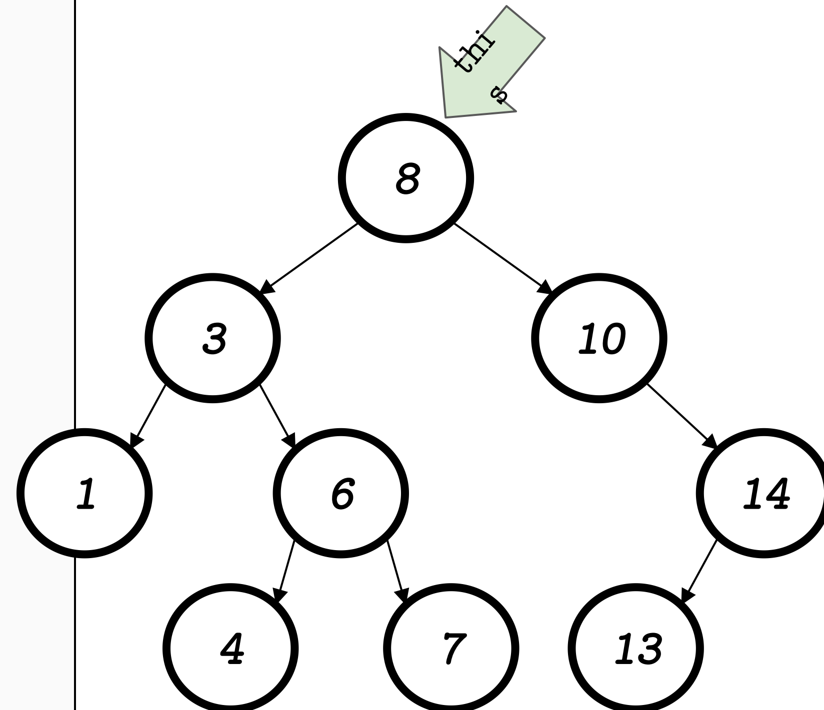
verdade! 4 < 8

```
bool Node::tem_elemento(int elemento) {
  if (this->_elemento == elemento) {
    return true;
  } else if (elemento < this->_elemento) {
    if (this->_esquerda == nullptr) {
      return false;
    } else {
      return this->_esquerda->tem_elemento(elemento);
    }
  } else {
    if (this->_direita == nullptr) {
      return false;
    } else {
      return this->_direita->tem_elemento(elemento);
    }
  }
}
```



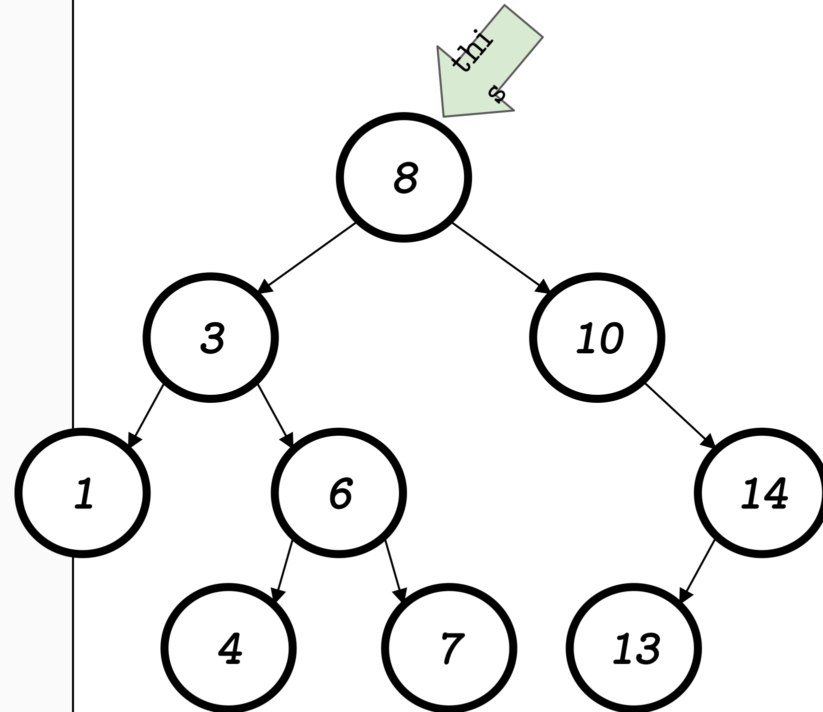
esquerda não é nullptr

```
bool Node::tem_elemento(int elemento) {
  if (this->_elemento == elemento) {
    return true;
  } else if (elemento < this->_elemento) {
    if (this->_esquerda == nullptr) {
      return false;
    } else {
      return this->_esquerda->tem_elemento(elemento);
    }
  } else {
    if (this->_direita == nullptr) {
      return false;
    } else {
      return this->_direita->tem_elemento(elemento);
    }
  }
}
```



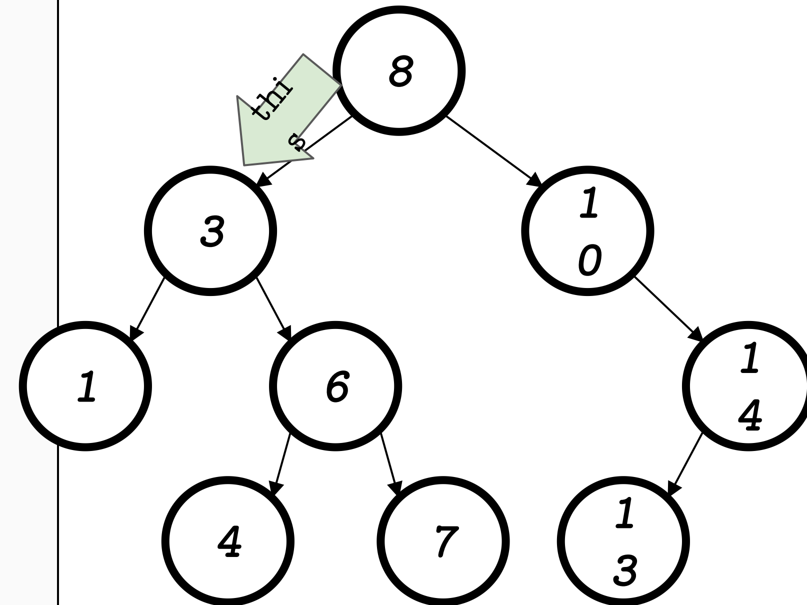
chamamos para o próximo

```
bool Node::tem_elemento(int elemento) {
  if (this->_elemento == elemento) {
    return true;
  } else if (elemento < this->_elemento) {
    if (this->_esquerda == nullptr) {
      return false;
    } else {
      return this->_esquerda->tem_elemento(elemento);
    }
  } else {
    if (this->_direita == nullptr) {
      return false;
    } else {
      return this->_direita->tem_elemento(elemento);
    }
  }
}
```



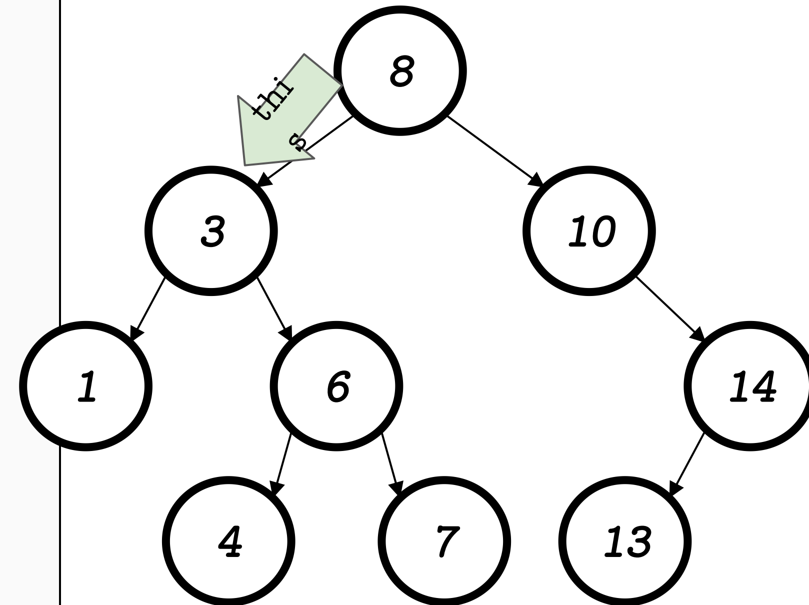
novamente, false: 3 != 4

```
bool Node::tem_elemento(int elemento) {
  if (this->_elemento == elemento) {
    return true;
  } else if (elemento < this->_elemento) {
    if (this->_esquerda == nullptr) {
      return false;
    } else {
      return this->_esquerda->tem_elemento(elemento);
    }
  } else {
    if (this->_direita == nullptr) {
      return false;
    } else {
      return this->_direita->tem_elemento(elemento);
    }
  }
}
```



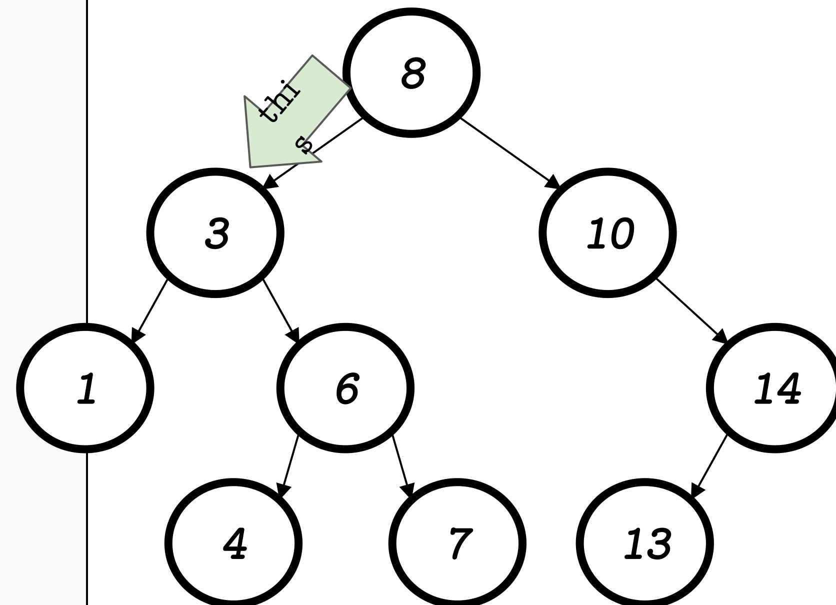
como 4 > 3, vamos para direita

```
bool Node::tem_elemento(int elemento) {
  if (this->_elemento == elemento) {
    return true;
  } else if (elemento < this->_elemento) {
    if (this->_esquerda == nullptr) {
      return false;
    } else {
      return this->_esquerda->tem_elemento(elemento);
    }
  } else {
    if (this->_direita == nullptr) {
      return false;
    } else {
      return this->_direita->tem_elemento(elemento);
    }
  }
}
```



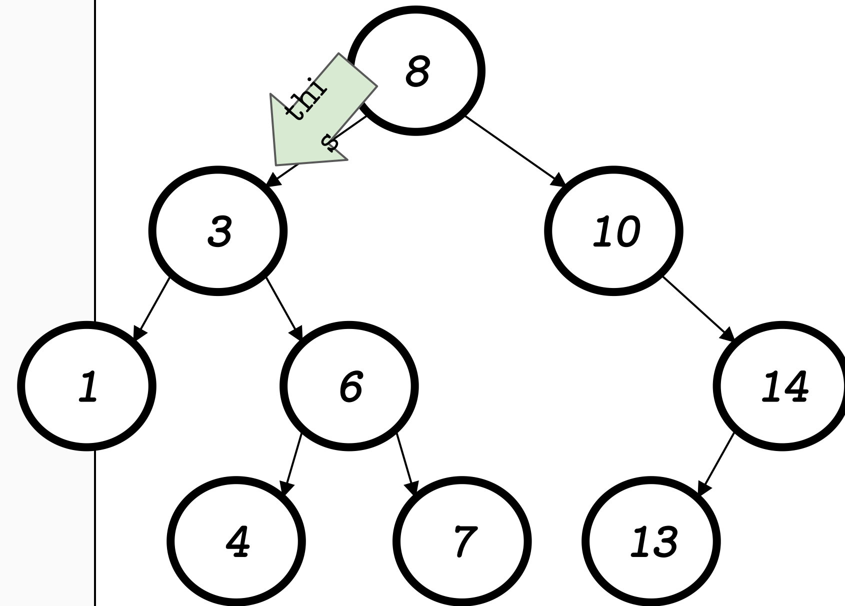
não é nullptr

```
bool Node::tem_elemento(int elemento) {
  if (this->_elemento == elemento) {
    return true;
  } else if (elemento < this->_elemento) {
    if (this->_esquerda == nullptr) {
      return false;
    } else {
      return this->_esquerda->tem_elemento(elemento);
    }
  } else {
    if (this->_direita == nullptr) {
      return false;
    } else {
      return this->_direita->tem_elemento(elemento);
    }
  }
}
```



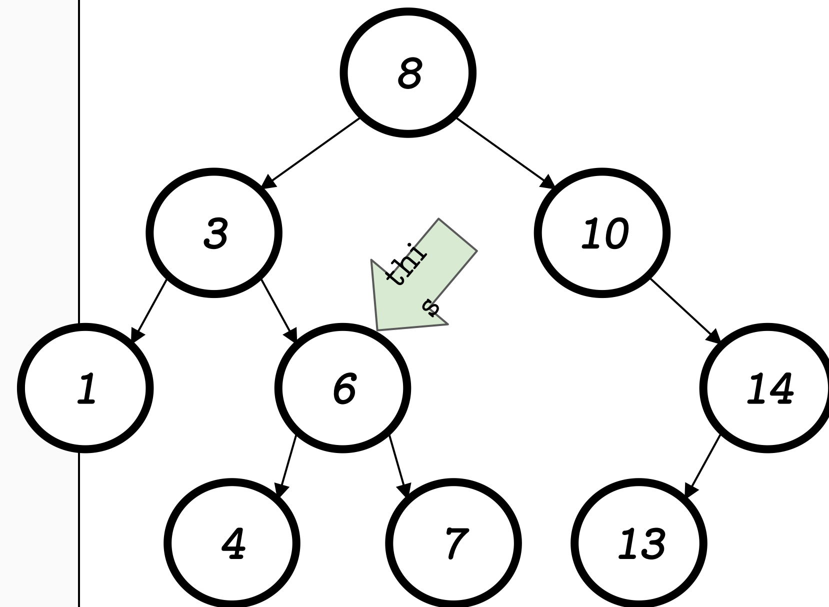
não é nullptr

```
bool Node::tem_elemento(int elemento) {
  if (this->_elemento == elemento) {
    return true;
  } else if (elemento < this->_elemento) {
    if (this->_esquerda == nullptr) {
      return false;
    } else {
      return this->_esquerda->tem_elemento(elemento);
    }
  } else {
    if (this->_direita == nullptr) {
      return false;
    } else {
      return this->_direita->tem_elemento(elemento);
    }
  }
}
```



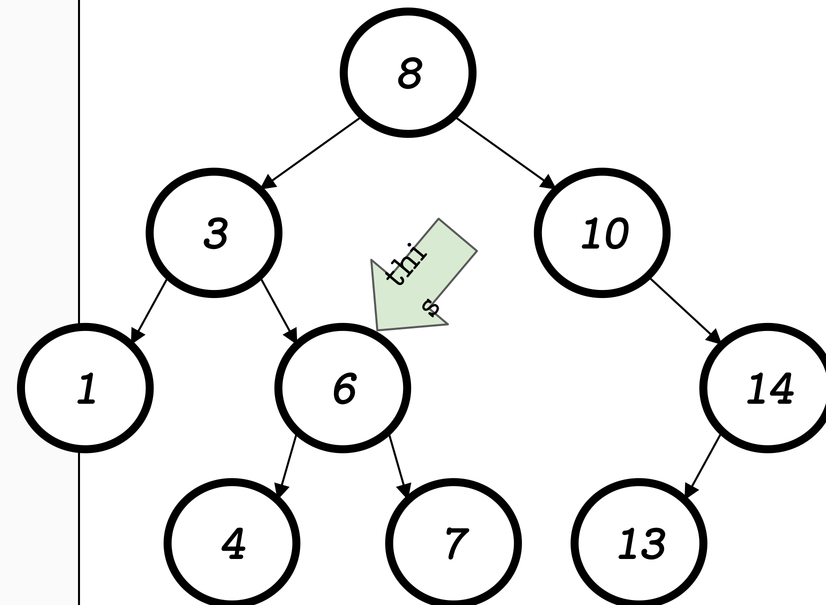
estamos no 6, vamos para esq.

```
bool Node::tem_elemento(int elemento) {
  if (this->_elemento == elemento) {
    return true;
  } else if (elemento < this->_elemento) {
    if (this->_esquerda == nullptr) {
      return false;
    } else {
      return this->_esquerda->tem_elemento(elemento);
    }
  } else {
    if (this->_direita == nullptr) {
      return false;
    } else {
      return this->_direita->tem_elemento(elemento);
    }
  }
}
```



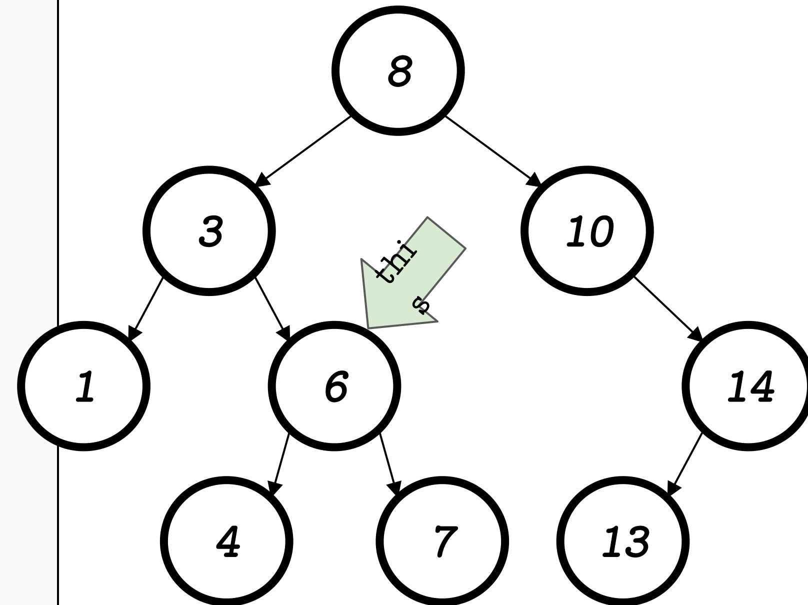
estamos no 6, vamos para esq.

```
bool Node::tem_elemento(int elemento) {
  if (this->_elemento == elemento) {
    return true;
  } else if (elemento < this->_elemento) {
    if (this->_esquerda == nullptr) {
      return false;
    } else {
      return this->_esquerda->tem_elemento(elemento);
    }
  } else {
    if (this->_direita == nullptr) {
      return false;
    } else {
      return this->_direita->tem_elemento(elemento);
    }
  }
}
```



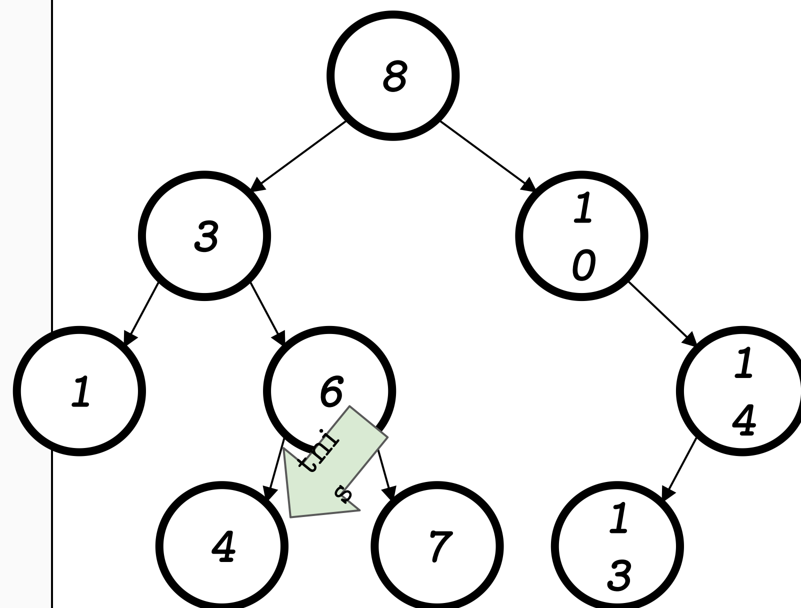
estamos no 6, vamos para esq.

```
bool Node::tem_elemento(int elemento) {
  if (this->_elemento == elemento) {
    return true;
  } else if (elemento < this->_elemento) {
    if (this->_esquerda == nullptr) {
      return false;
    } else {
      return this->_esquerda->tem_elemento(elemento);
    }
  } else {
    if (this->_direita == nullptr) {
      return false;
    } else {
      return this->_direita->tem_elemento(elemento);
    }
  }
}
```



achamos!!

```
bool Node::tem_elemento(int elemento) {
  if (this->_elemento == elemento) {
    return true;
  } else if (elemento < this->_elemento) {
    if (this->_esquerda == nullptr) {
      return false;
    } else {
      return this->_esquerda->tem_elemento(elemento);
    }
  } else {
    if (this->_direita == nullptr) {
      return false;
    } else {
      return this->_direita->tem_elemento(elemento);
    }
  }
}
```



Em cada passo

- this->elemento == x?
  - Achamos o nČ!
- this->elemento < x?
  - Passo para a esquerda
- this->elemento > x?
  - Passo para a direita

- Temos uma chamada para o outro nó
  - Uma forma de recursão
  - Mas OO deixa isso um pouco mais oculto

Inserindo um elemento

- Similar ao código anterior
- Só que em algum momento vamos chegar em nullptr
- Local correto da inserção

Inserção

```
void Node::inserir_elemento(int elemento) {
  if (elemento < this->_elemento) {
    if (this->_esquerda == nullptr) {
      this->_esquerda = new Node(elemento);
    } else {
      this->_esquerda->inserir_elemento(elemento);
    }
  } else if (elemento > this->_elemento) {
    if (this->_direita == nullptr) {
      this->_direita = new Node(elemento);
    } else {
      this->_direita->inserir_elemento(elemento);
    }
  }
}
```

Ao achar nulo, criamos novo nó!

Ao achar nulo, criamos novo nó!

Inserção

```
void Node::inserir_elemento(int elemento) {
  if (elemento < this->_elemento) {
    if (this->_esquerda == nullptr) {
      this->_esquerda = new Node(elemento);
    } else {
      this->_esquerda->inserir_elemento(elemento);
    }
  } else if (elemento > this->_elemento) {
    if (this->_direita == nullptr) {
      this->_direita = new Node(elemento);
    } else {
      this->_direita->inserir_elemento(elemento);
    }
  }
}
```

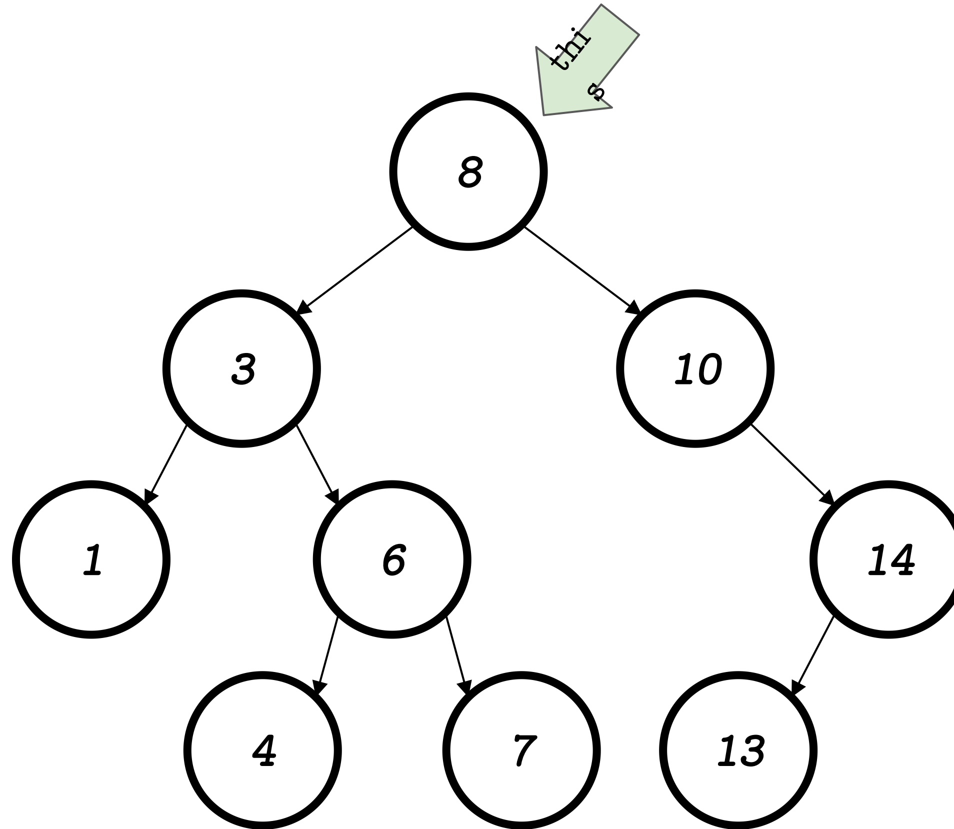
Ao achar nulo, criamos novo nó!

Se não for nulo, caminha

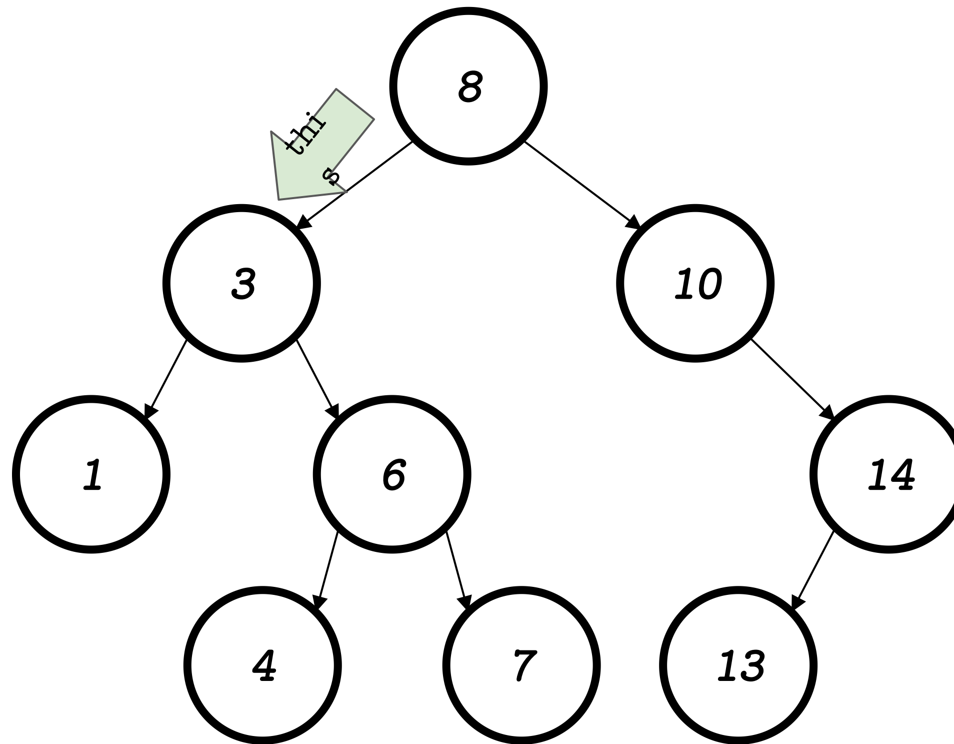
Ao achar nulo, criamos novo nó!

Se não for nulo, caminha

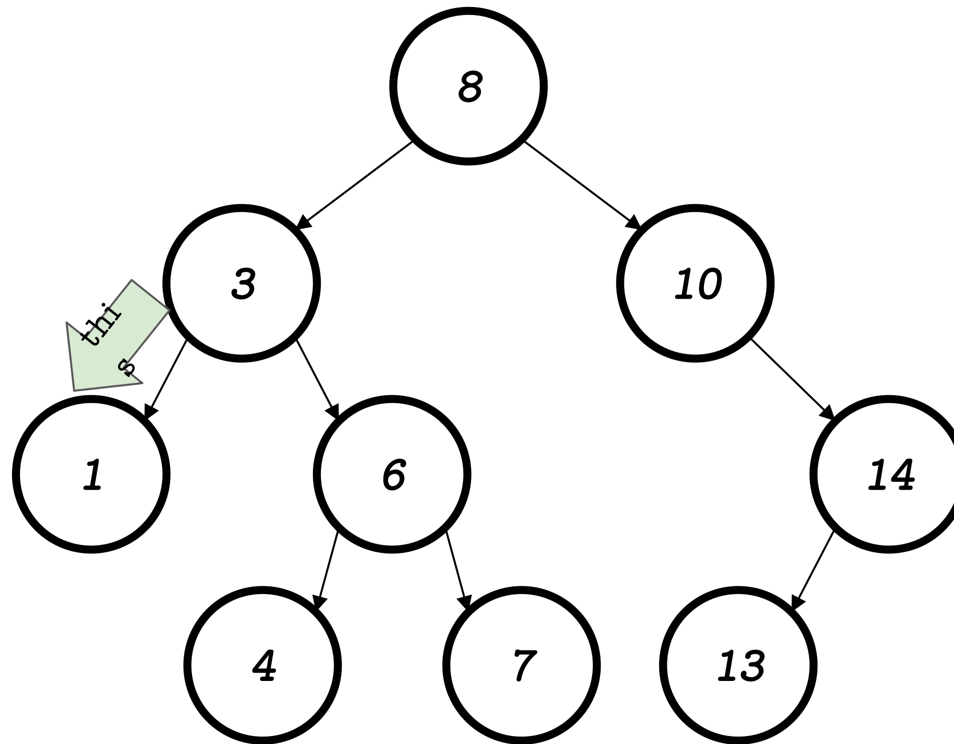
Inserção x = 0



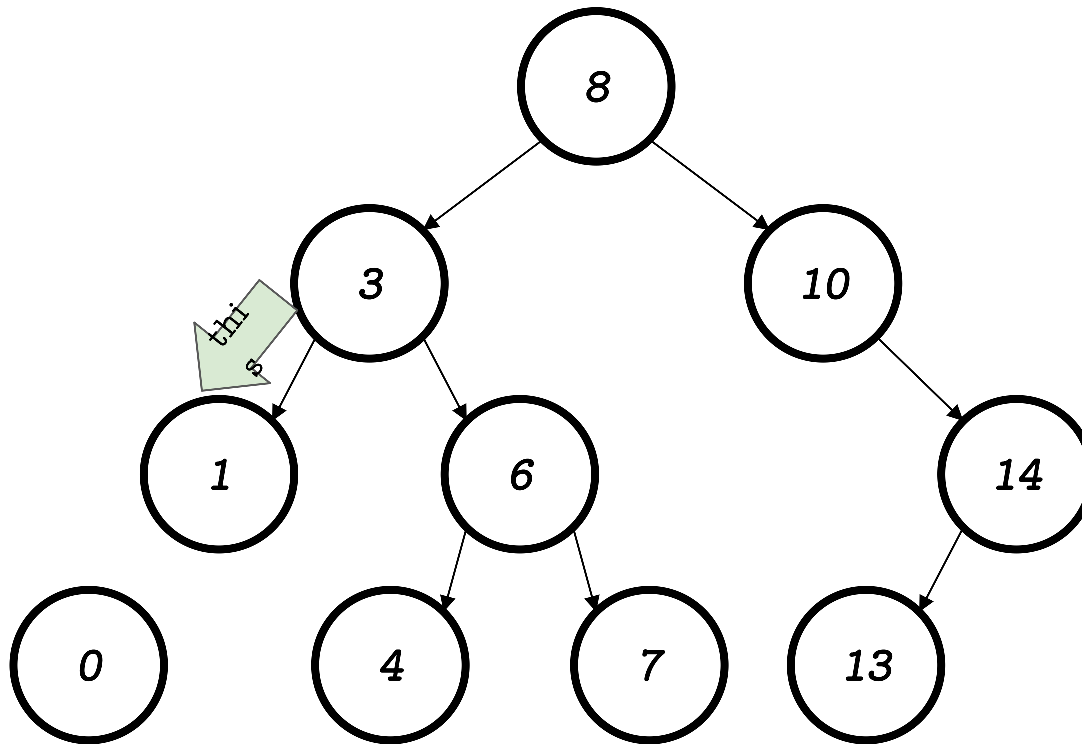
Inserção x = 0



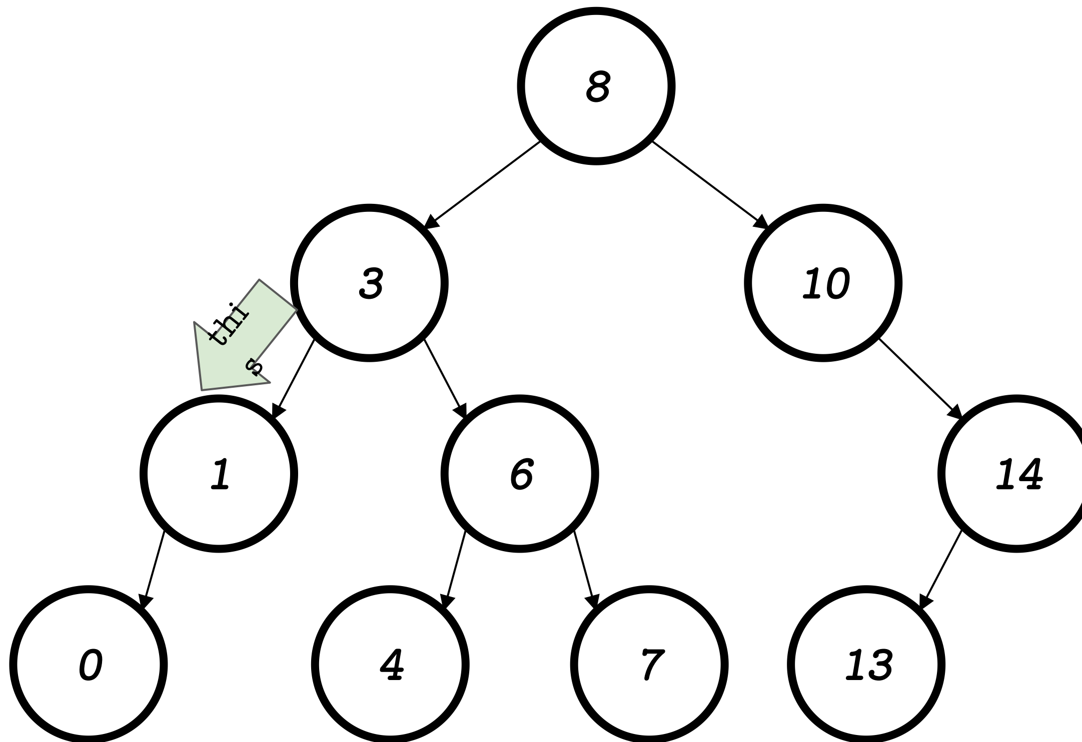
Inserção x = 0



Inserção x = 0



Inserção x = 0



Implementando o TAD BST

Basta guardar um ponteiro para a raiz

```
#ifndef PDS2_BST_H
#define PDS2_BST_H

#include "node.h"

class BST {
private:
  Node *_raiz;
  int _num_elementos_inseridos;
public:
  BST();
  ~BST();
  void inserir_elemento(int elemento);
  bool tem_elemento(int elemento);
  void imprimir();
};

#endif
```

Implementando o TAD BST
Chamamos os métodos da raiz (ver git)

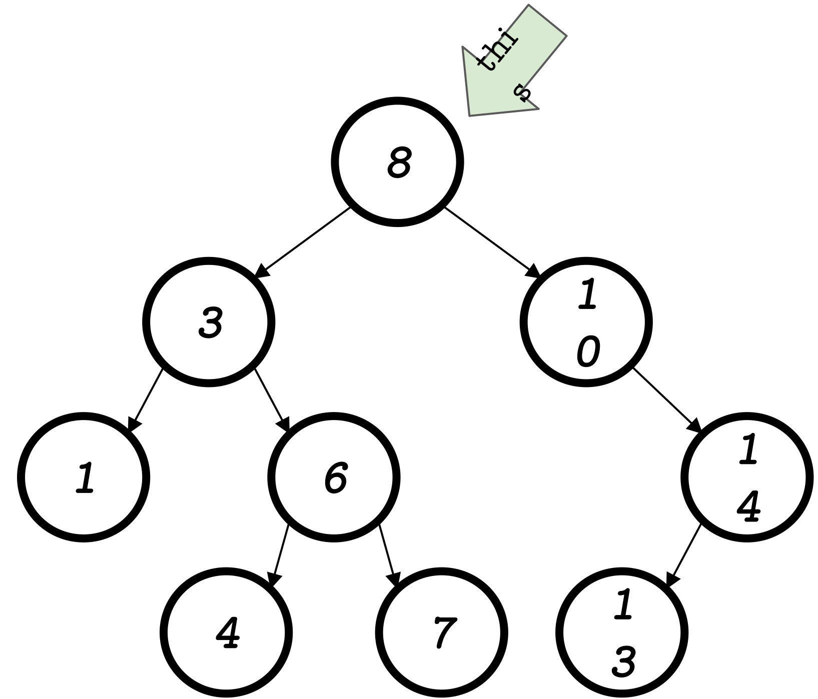
```
#ifndef PDS2_BST_H
#define PDS2_BST_H

#include "node.h"

class BST {
private:
  Node *_raiz;
  int _num_elementos_inseridos;
public:
  BST();
  ~BST();
  void inserir_elemento(int elemento);
  bool tem_elemento(int elemento);
  void imprimir();
};

#endif
```

Impressões e Caminhamentos
Diferentes formas de visitar todos os nós

- Pré-ordem
  - Visita o nó
  - Visita tudo para a esquerda e depois direita
- Pós-ordem
  - Visita tudo para a esquerda e depois direita
  - Visita o nó
- Em-ordem
  - Esquerda → no → direita

Impressão pre-ordem

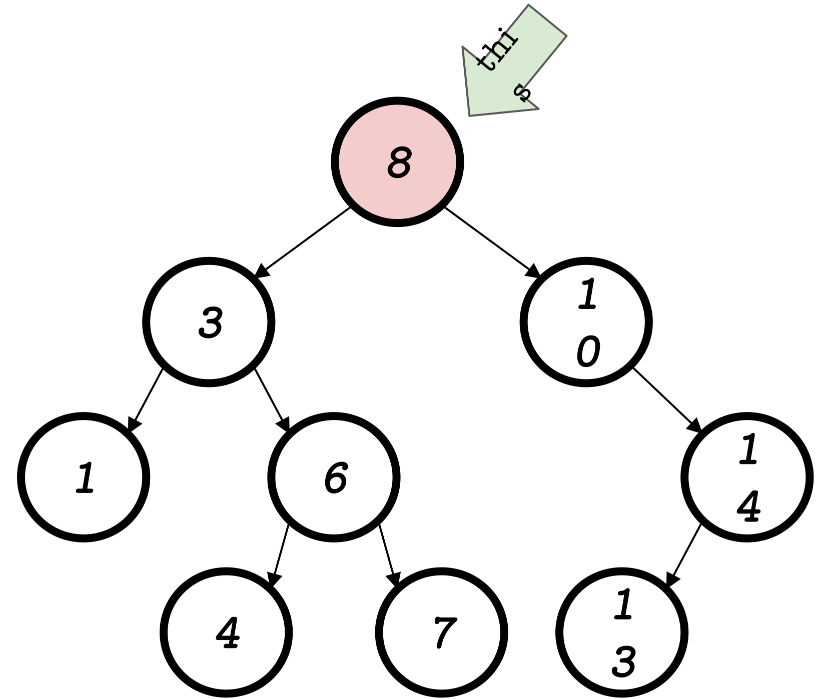
```
void Node::imprimir() {
  std::cout << this->_elemento;
  if (this->_esquerda != nullptr)
            this->_esquerda->imprimir();
  if (this->_direita != nullptr)
            this->_direita->imprimir();
}
```




```
{8}.imprimir()
```

Impressão pre-ordem
8,

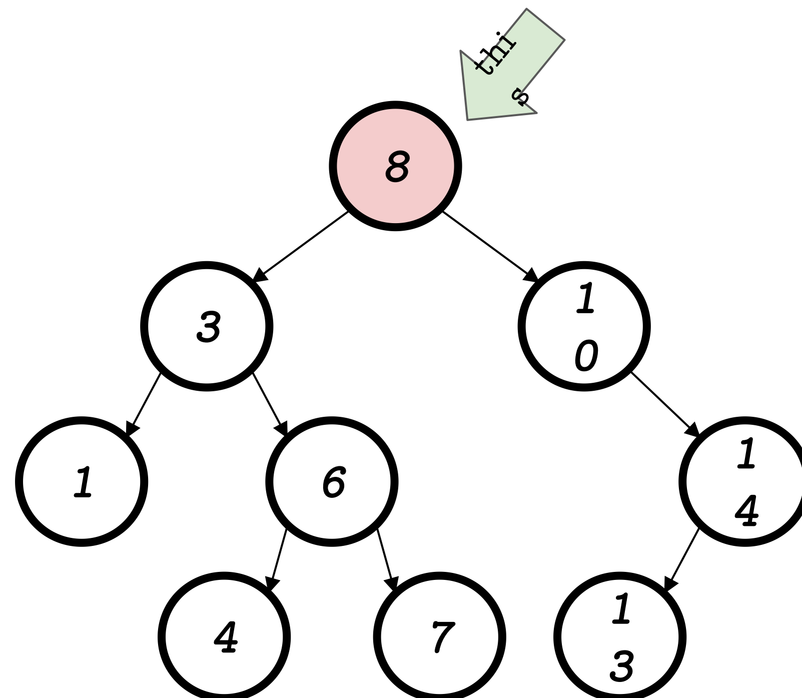
```
void Node::imprimir() {
  std::cout << this->_elemento;
  if (this->_esquerda != nullptr)
           this->_esquerda->imprimir();
  if (this->_direita != nullptr)
           this->_direita->imprimir();
}
```



{8}.imprimir()

Impressão pre-ordem
8,

```
void Node::imprimir() {
  std::cout << this->_elemento;
  if (this->_esquerda != nullptr)
          this->_esquerda->imprimir();
  if (this->_direita != nullptr)
          this->_direita->imprimir();
}
```



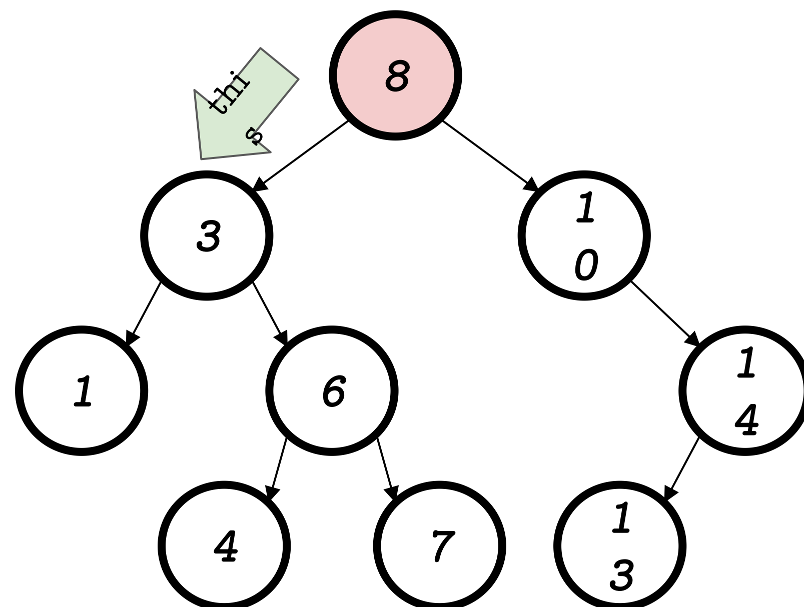

```
{8}.imprimir()
```

Impressão pre-ordem

8,

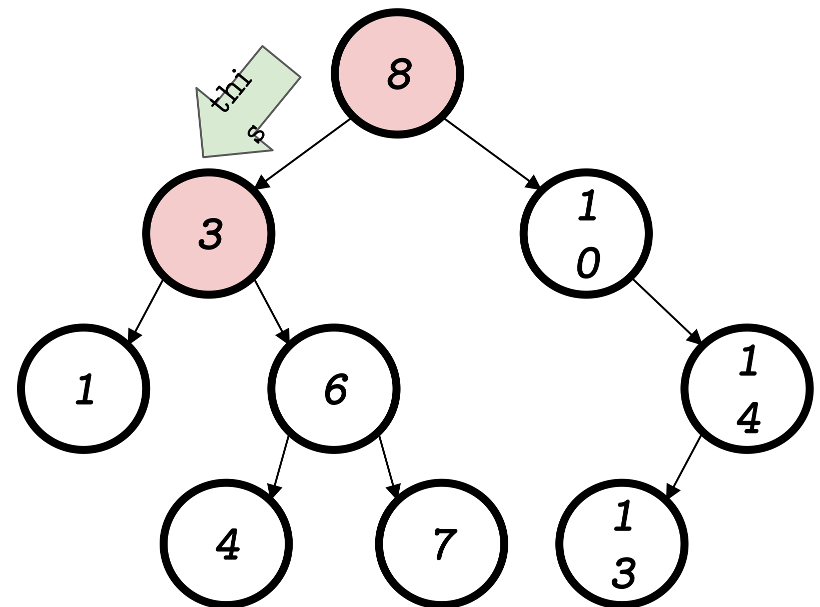
```
void Node::imprimir() {
  std::cout << this->_elemento;
  if (this->_esquerda != nullptr)
            this->_esquerda->imprimir();
  if (this->_direita != nullptr)
            this->_direita->imprimir();
}
```

{3}.imprimir()
{8}.imprimir()



Impressão pre-ordem
8, 3

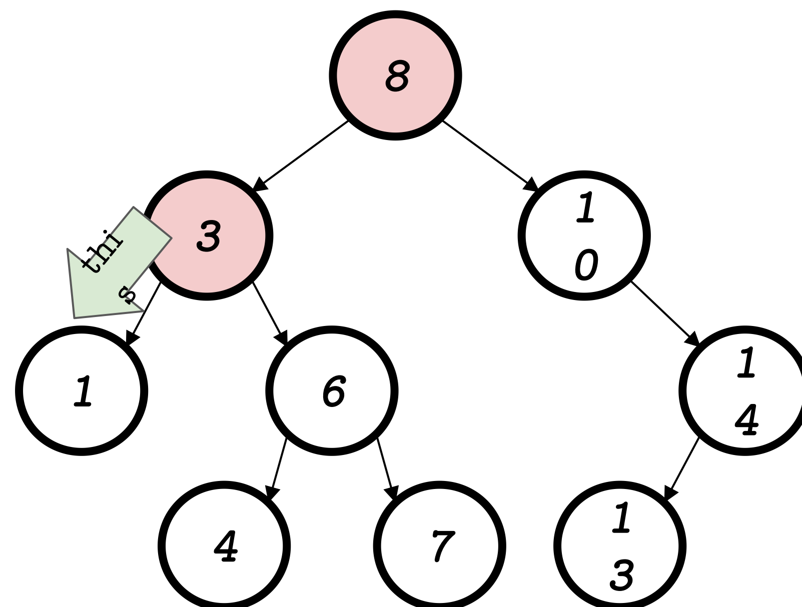
```
void Node::imprimir() {
  std::cout << this->_elemento;
  if (this->_esquerda != nullptr)
          this->_esquerda->imprimir();
  if (this->_direita != nullptr)
          this->_direita->imprimir();
}
```



{3}.imprimir()
{8}.imprimir()

Impressão pre-ordem
8, 3

```
void Node::imprimir() {
  std::cout << this->_elemento;
  if (this->_esquerda != nullptr)
          this->_esquerda->imprimir();
  if (this->_direita != nullptr)
          this->_direita->imprimir();
}
```

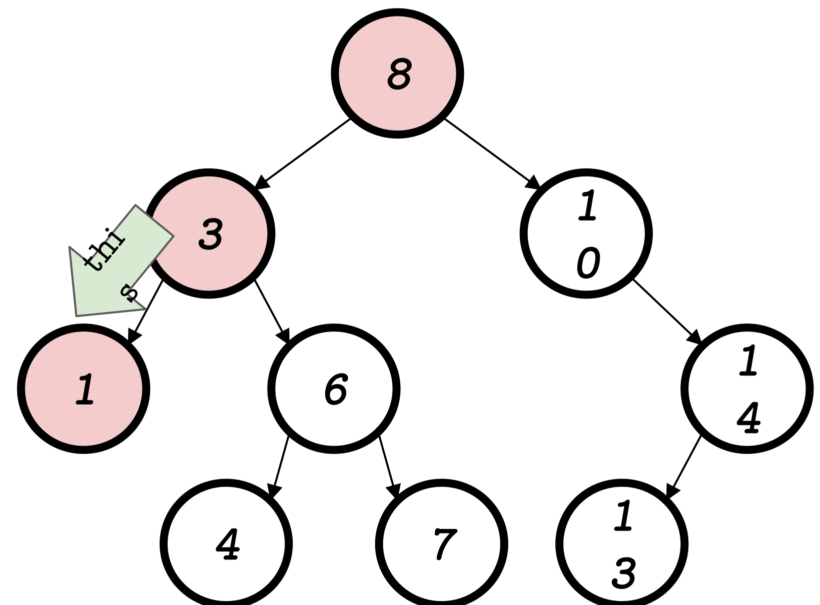


{1}.imprimir()
{3}.imprimir()
{8}.imprimir()

Impressão pre-ordem
8, 3, 1

```
void Node::imprimir() {
  std::cout << this->_elemento;
  if (this->_esquerda != nullptr)
          this->_esquerda->imprimir();
  if (this->_direita != nullptr)
          this->_direita->imprimir();
}
```

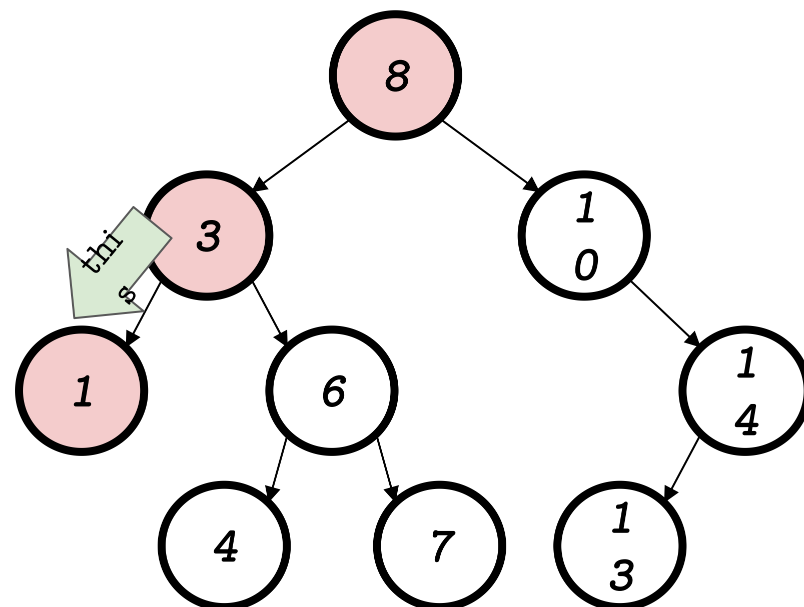
{1}.imprimir()
{3}.imprimir()
{8}.imprimir()



Impressão pre-ordem
8, 3, 1

```
void Node::imprimir() {
  std::cout << this->_elemento;
  if (this->_esquerda != nullptr)
          this->_esquerda->imprimir();
  if (this->_direita != nullptr)
          this->_direita->imprimir();
}
```

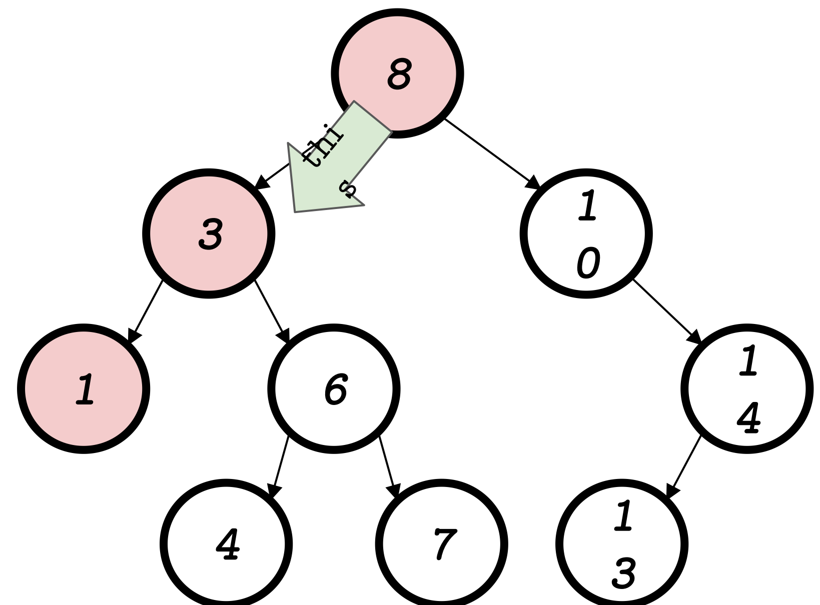
{1}.imprimir()
{3}.imprimir()
{8}.imprimir()



Impressão pre-ordem

Voltamos para o nó 3, olhe a pilha de chamadas

```
void Node::imprimir() {
  std::cout << this->_elemento;
  if (this->_esquerda != nullptr)
          this->_esquerda->imprimir();
  if (this->_direita != nullptr)
          this->_direita->imprimir();
}
```

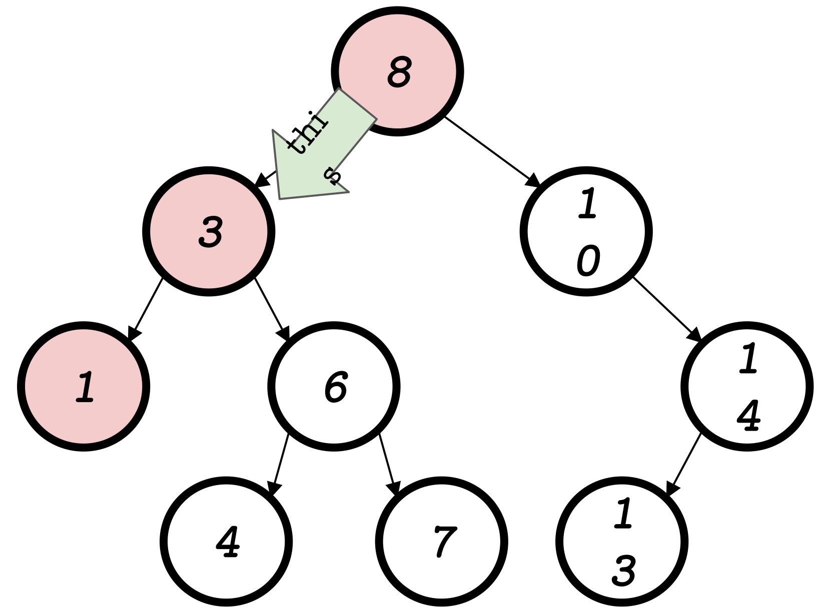


{3}.imprimir()
{8}.imprimir()

Impressão pre-ordem
Agora vamos para a direita

```
void Node::imprimir() {
  std::cout << this->_elemento;
  if (this->_esquerda != nullptr)
           this->_esquerda->imprimir();
  if (this->_direita != nullptr)
           this->_direita->imprimir();
}
```

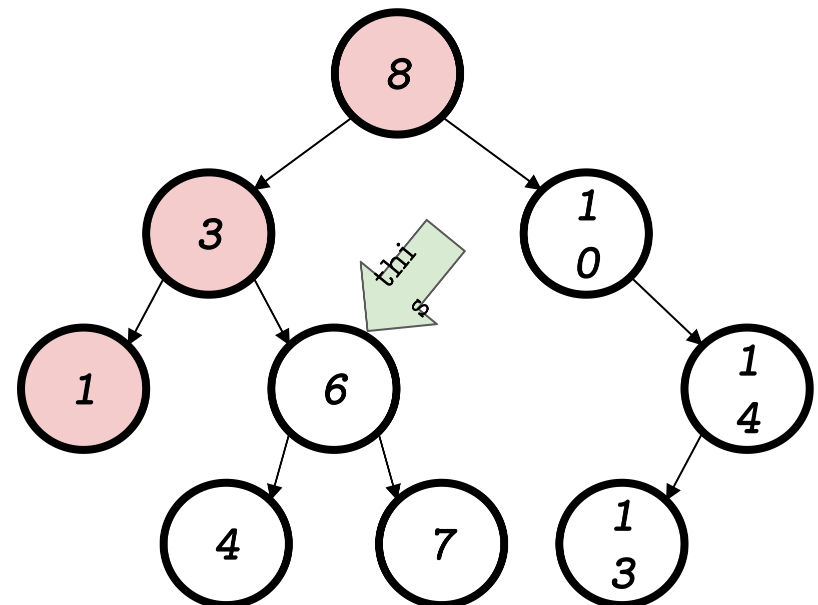
{3}.imprimir()
{8}.imprimir()



Impressão pre ordem
Agora no 6

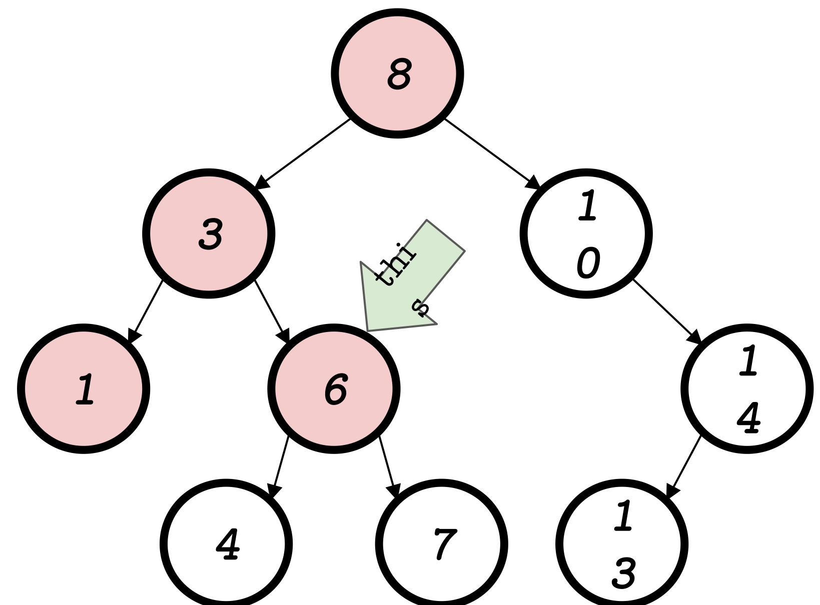
```
void Node::imprimir() {
  std::cout << this->_elemento;
  if (this->_esquerda != nullptr)
          this->_esquerda->imprimir();
  if (this->_direita != nullptr)
          this->_direita->imprimir();
}
```

{6}.imprimir()
{3}.imprimir()
{8}.imprimir()



Impressão pre-ordem
8, 3, 6

```
void Node::imprimir() {
  std::cout << this->_elemento;
  if (this->_esquerda != nullptr)
        this->_esquerda->imprimir();
  if (this->_direita != nullptr)
        this->_direita->imprimir();
}
```

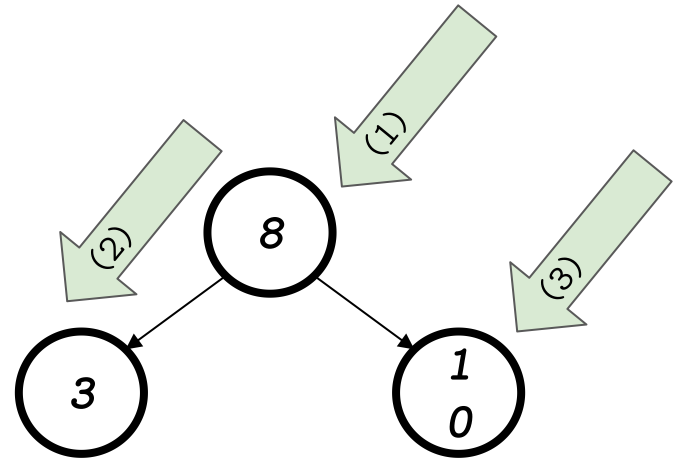


{6}.imprimir()
{3}.imprimir()
{8}.imprimir()

Caminhamentos
Entendendo os tipos

- Pré-ordem
  (visita os nós como forma inseridos)

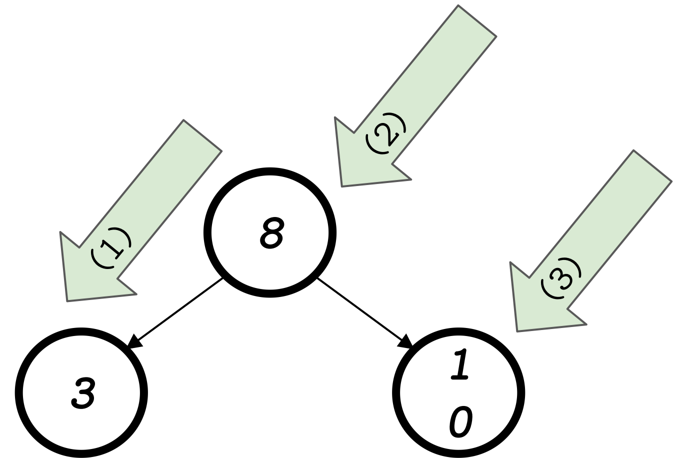
```
void Node::imprimir() {
  std::cout << this->_elemento;
  if (this->_esquerda != nullptr)
            this->_esquerda->imprimir();
  if (this->_direita != nullptr)
            this->_direita->imprimir();
}
```



Caminhamentos
Entendendo os tipos

- Em-ordem
  (visita os nós ordenados por valor)

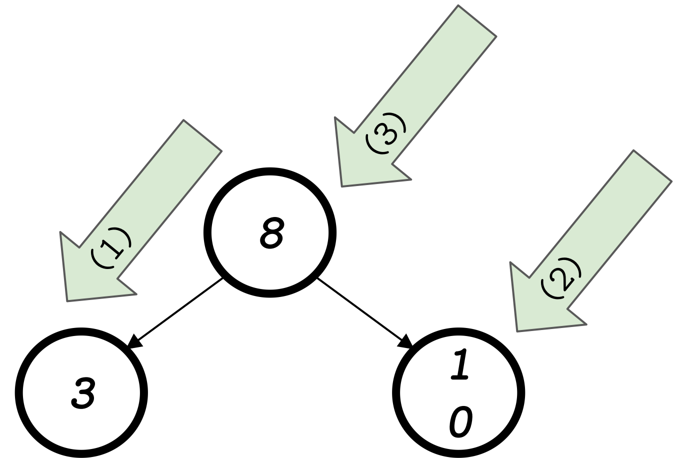
```
void Node::imprimir() {
  if (this->_esquerda != nullptr)
        this->_esquerda->imprimir();
  std::cout << this->_elemento;
  if (this->_direita != nullptr)
        this->_direita->imprimir();
}
```



Caminhamentos
Entendendo os tipos

- Pós-ordem
  (visita as folhas primeiro)

```
void Node::imprimir() {
  if (this->_esquerda != nullptr)
           this->_esquerda->imprimir();
  if (this->_direita != nullptr)
           this->_direita->imprimir();
  std::cout << this->_elemento;
}
```



Até agora...
Quebra cabeça para próximos passos

- Classes e Objetos
- Ponteiros e Referências
  - Ponteiros são mais utilizados internamente
  - Vide nossos TADs
- Precisamos fazer tudo do zero? Não! STL
- Podemos modelar comportamento comum? Sim! Interface.