



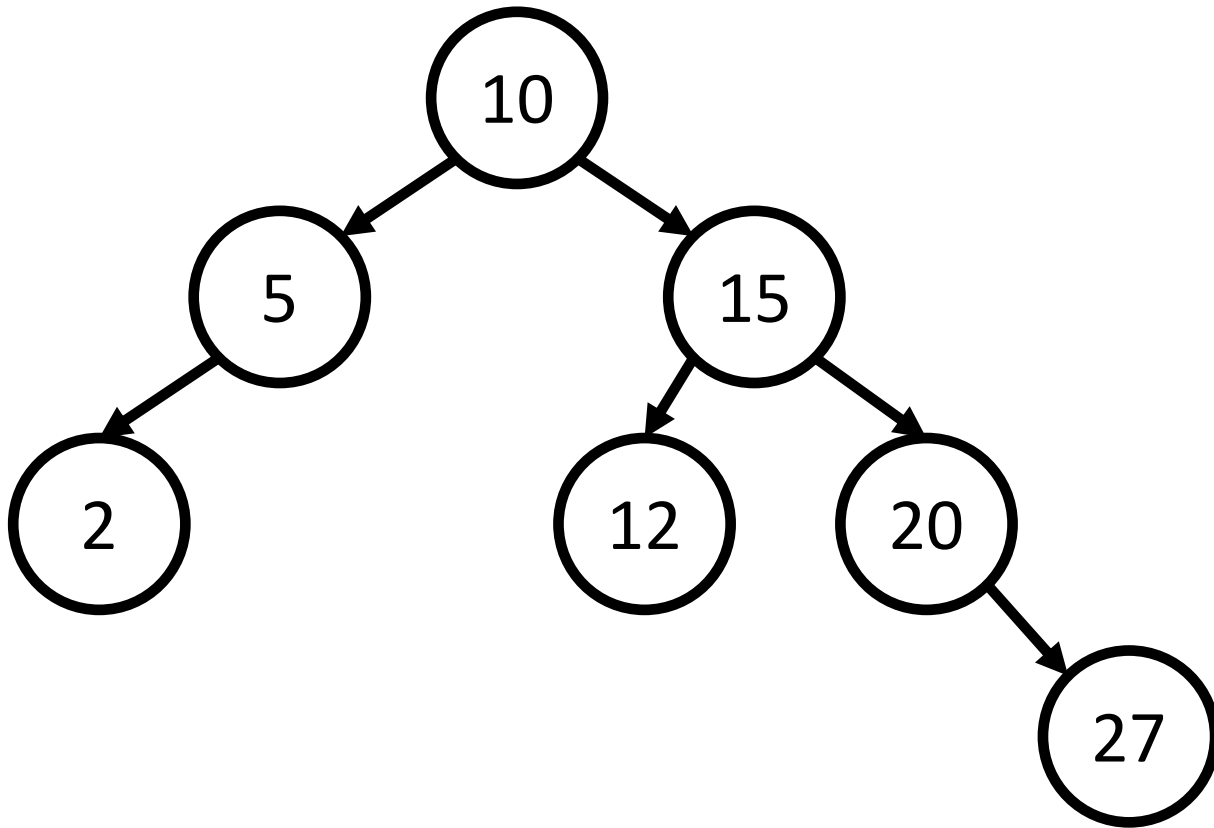
Estruturas de Dados

Árvores Binárias de Busca

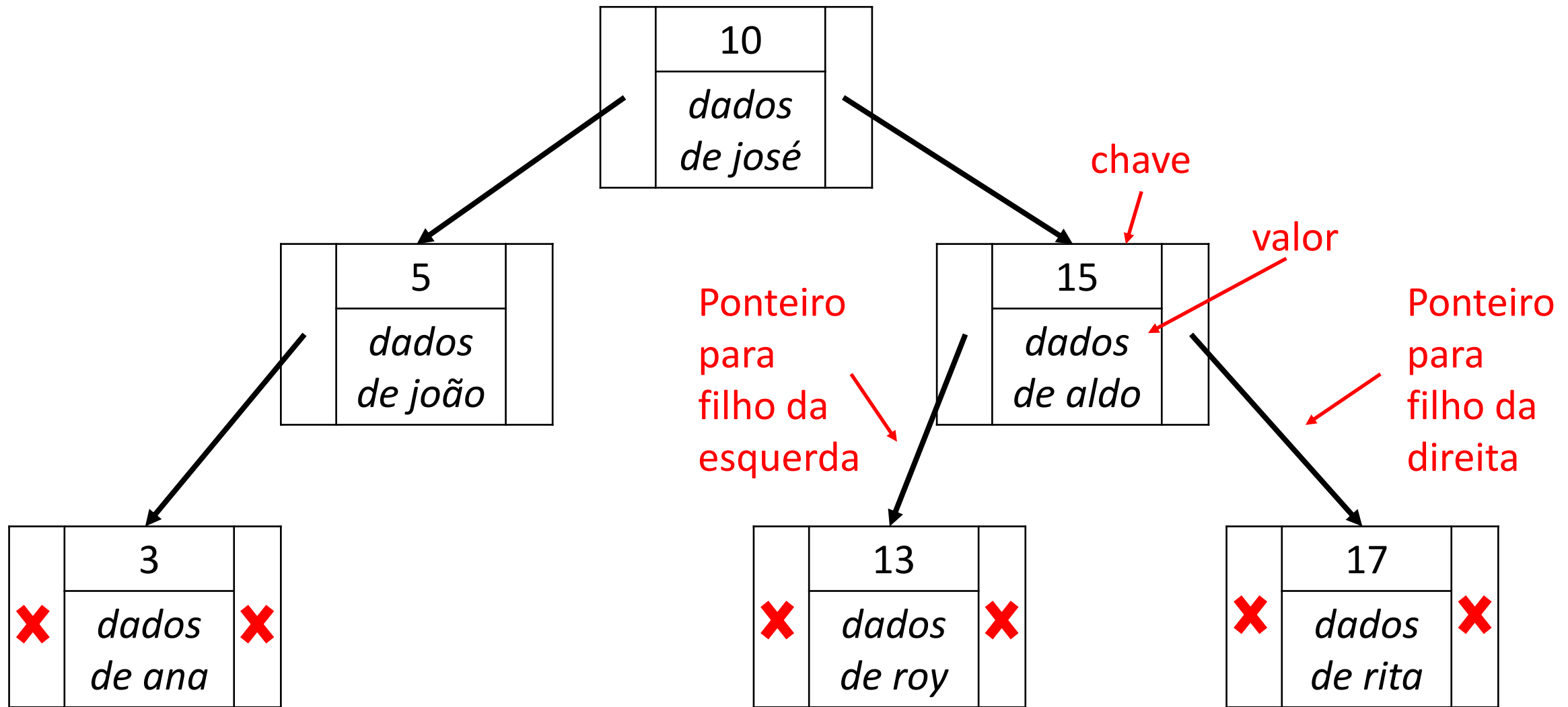
Árvores de Busca

- Árvores de busca são estruturas de dados associativas (armazenam pares chave-valor) que proveem as operações de **inserção, atualização, remoção, consulta (recuperação, busca), mínimo e máximo**.
- Elas podem ser usadas como **dicionários** e como **filas de prioridade**.

Árvores Binárias



- Árvores binárias são árvores em que cada nó possui entre 0 e 2 filhos.
- Em comparação, em árvores n -árias cada nó pode possuir até um número máximo de n filhos.

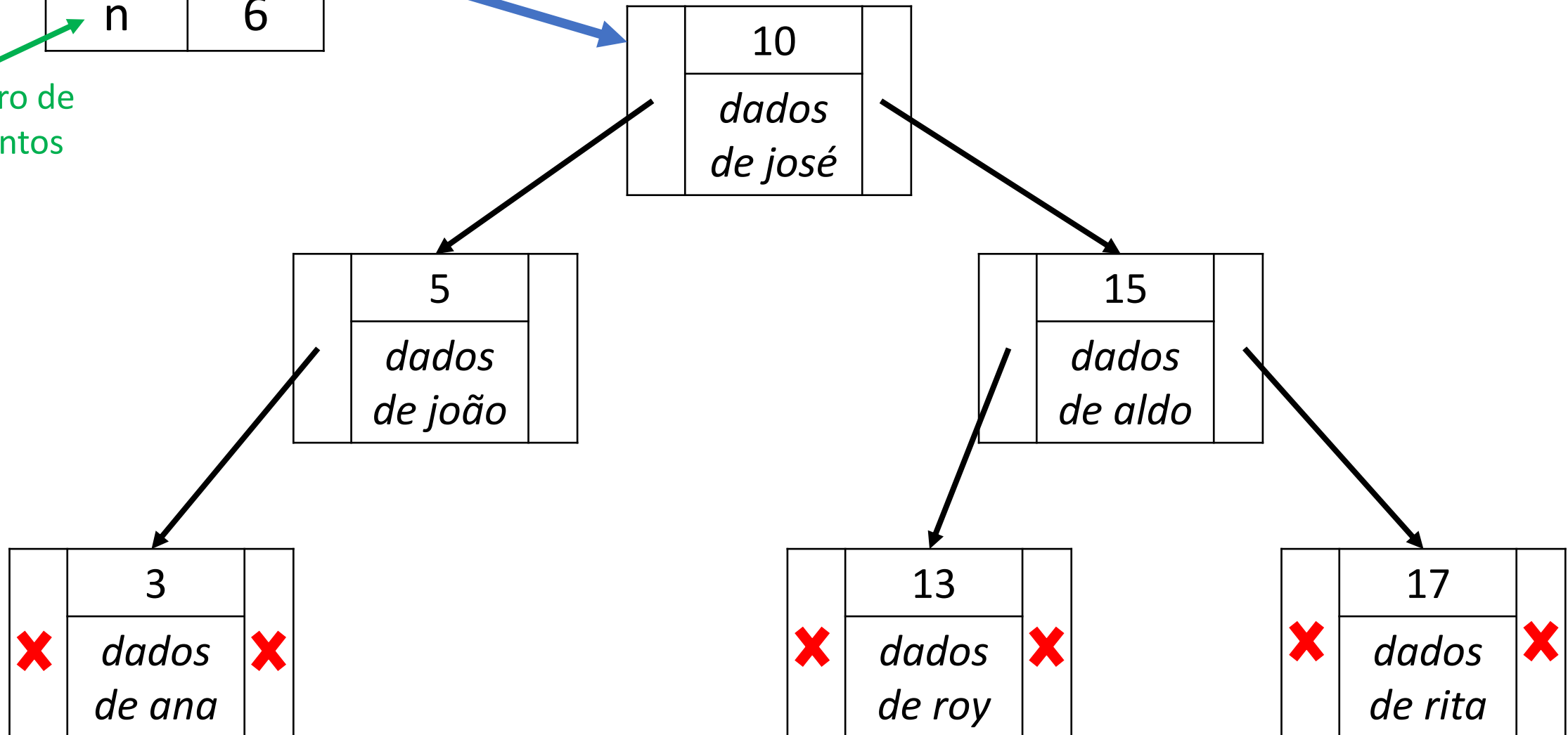


Árvores binárias de busca podem ser representadas por estruturas encadeadas em que cada nó possui uma chave, valores associados à chave (dados satélite) e ponteiros para os filhos da esquerda e direita. Alguns autores adicionam no nó um ponteiro para o pai.

BinaryTree	
raiz	
n	6

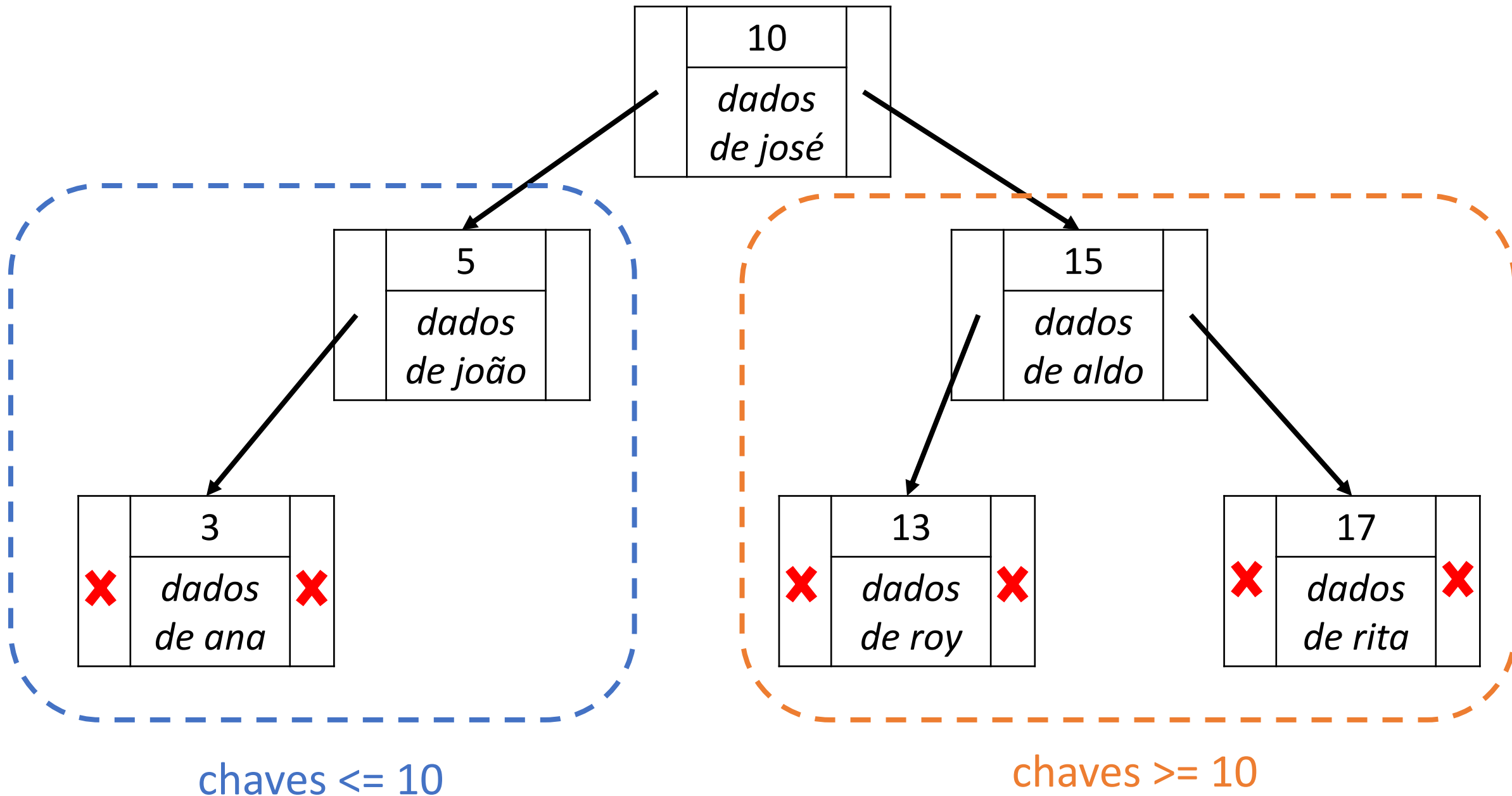
Podemos usar uma estrutura para encapsular e esconder os nós do usuário da árvore.

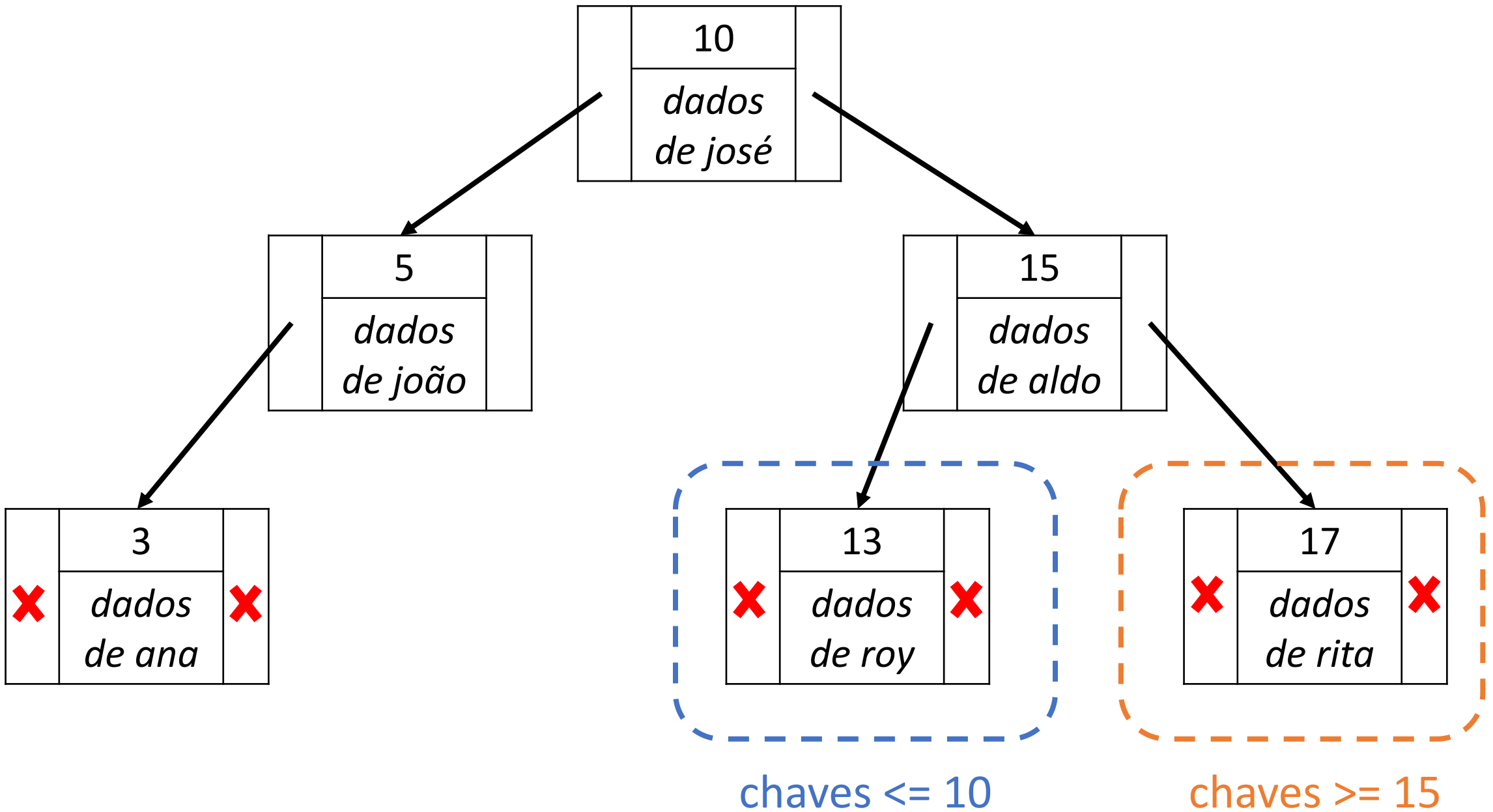
Número de elementos



Propriedade de Árvores Binárias de Busca

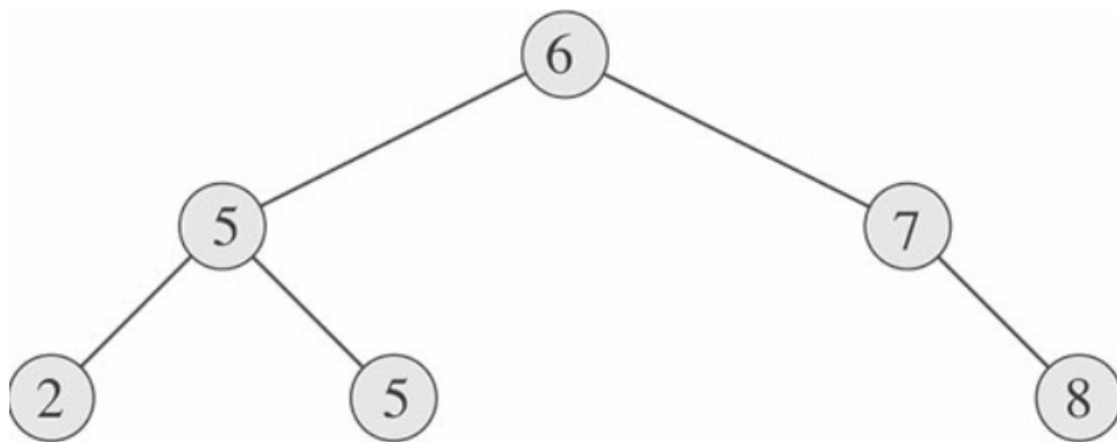
- Seja x um nó de uma árvore binária de busca.
- Se y é um nó na subárvore esquerda de x , então a $\text{chave}[y] \leq \text{chave}[x]$.
- De forma complementar, se y é um nó na subárvore direita de x , então a $\text{chave}[y] \geq \text{chave}[x]$.



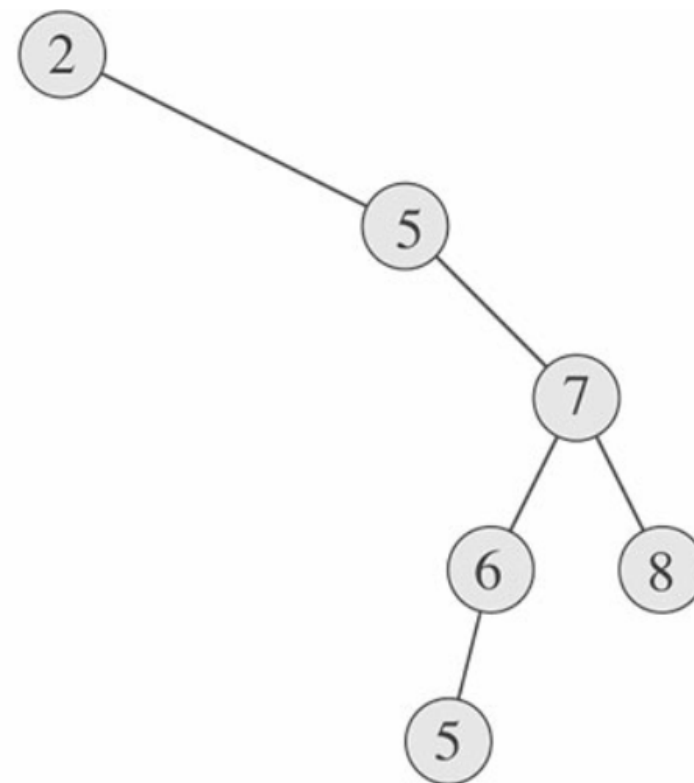


Complexidade de Operações

- As operações de inserção, remoção, consulta (recuperação, busca), mínimo e máximo possuem complexidade proporcional à **altura da árvore**.
- Em árvores **balanceadas**, a complexidade é **$O(\log_2 N)$** no pior caso, onde N é o número de nós na árvore.
- Se a árvore é uma **cadeia linear** de nós, as operações demoram um tempo **$O(N)$** no pior caso.



(a)



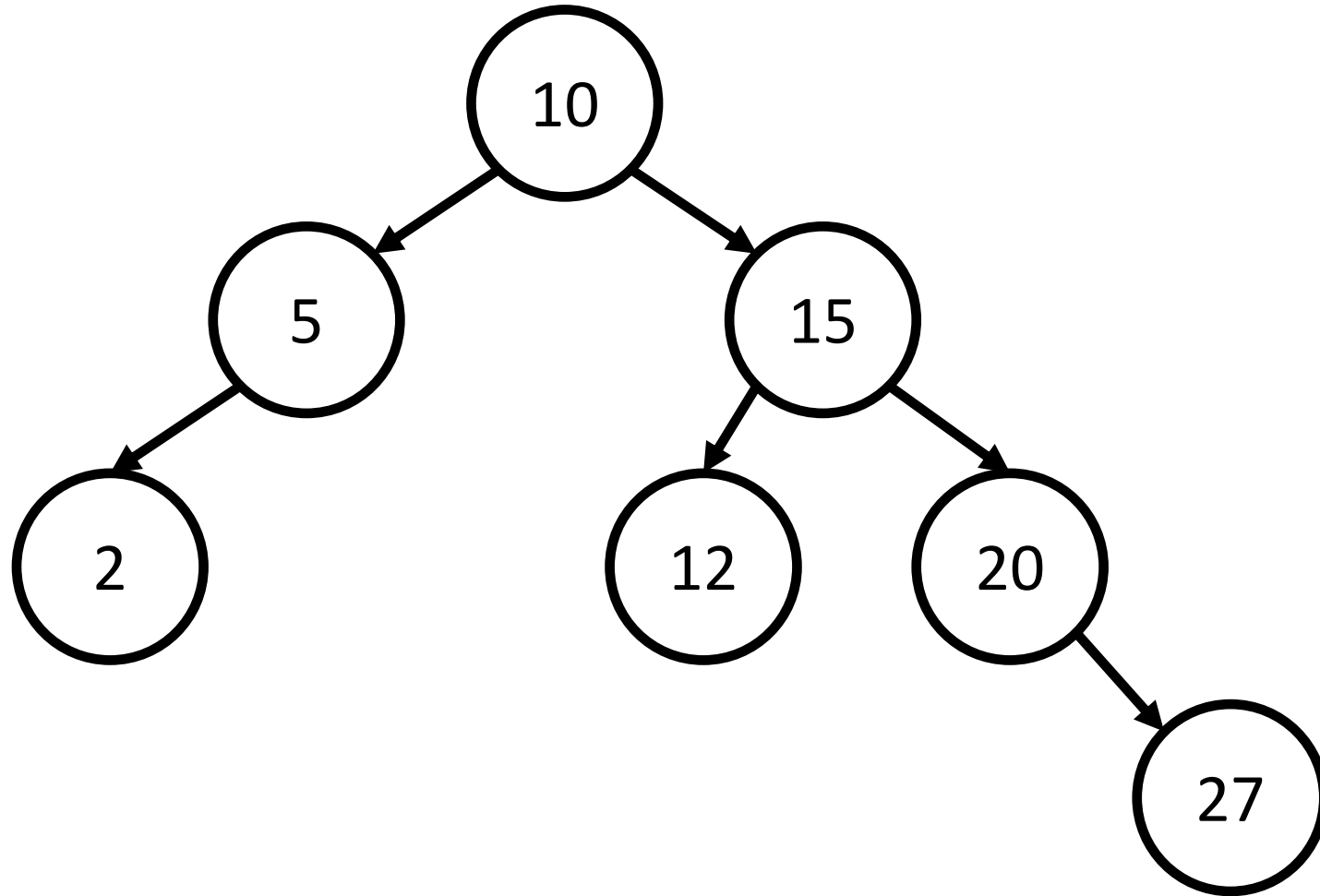
(b)

Figura 12.1 Árvores de busca binária. Para qualquer nó x , as chaves na subárvore esquerda de x são no máximo $x.chave$, e as chaves na subárvore direita de x são no mínimo $x.chave$. Árvores de busca binária diferentes podem representar o mesmo conjunto de valores. O tempo de execução do pior caso para a maioria das operações em árvores de busca é proporcional à altura da árvore. **(a)** Uma árvore de busca binária com seis nós e altura 2. **(b)** Uma árvore de busca binária menos eficiente, com altura 4, que contém as mesmas chaves.

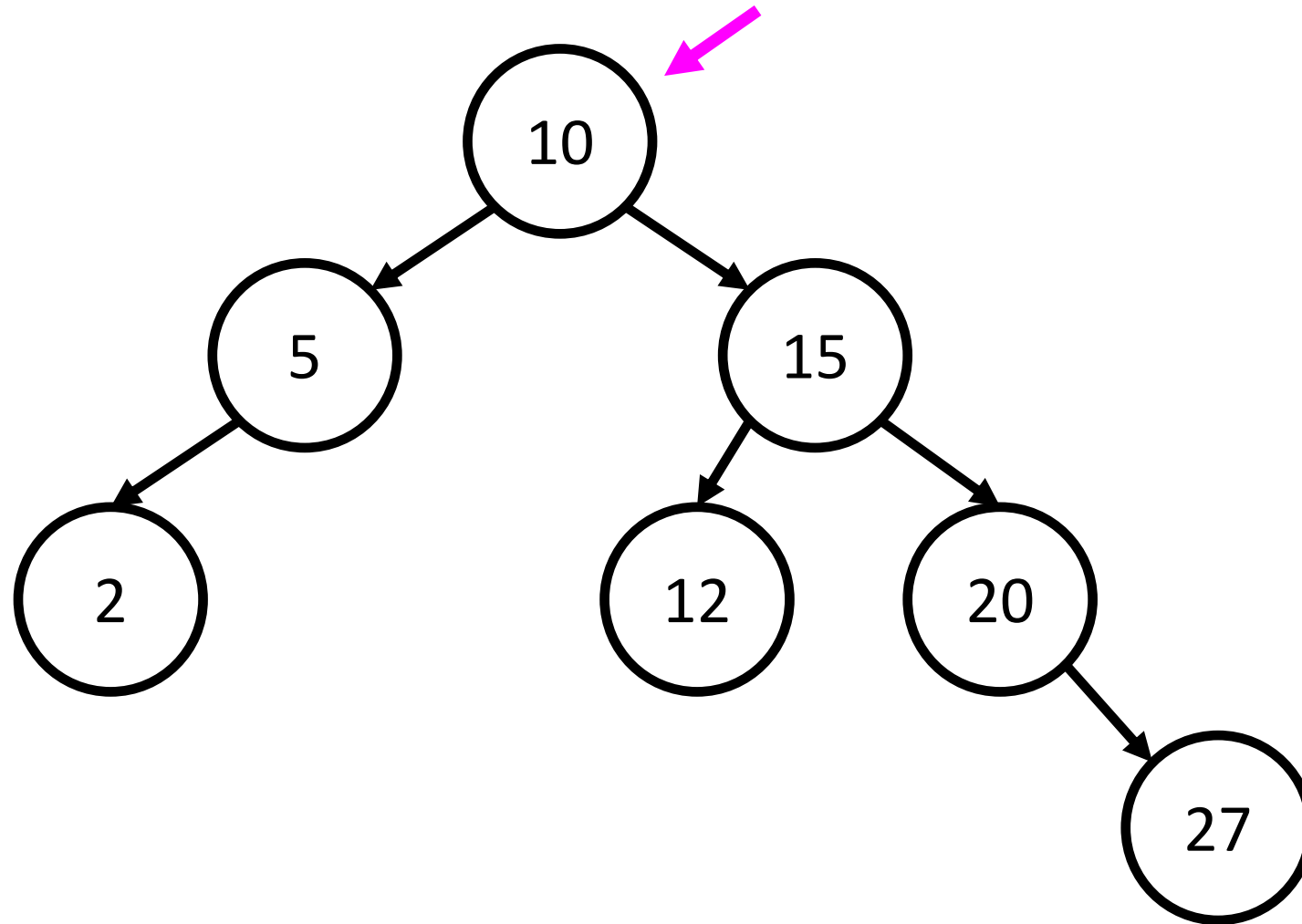
Consulta

1. Se a raíz da subárvore atual é nula, o item não existe, então retornamos NULL.
2. Se a raíz da subárvore atual tem a chave buscada, retorne o valor associado.
3. Se a chave buscada for menor que a da raíz, buscamos na subárvore da esquerda.
4. Caso contrário, buscamos na subárvore da direita.

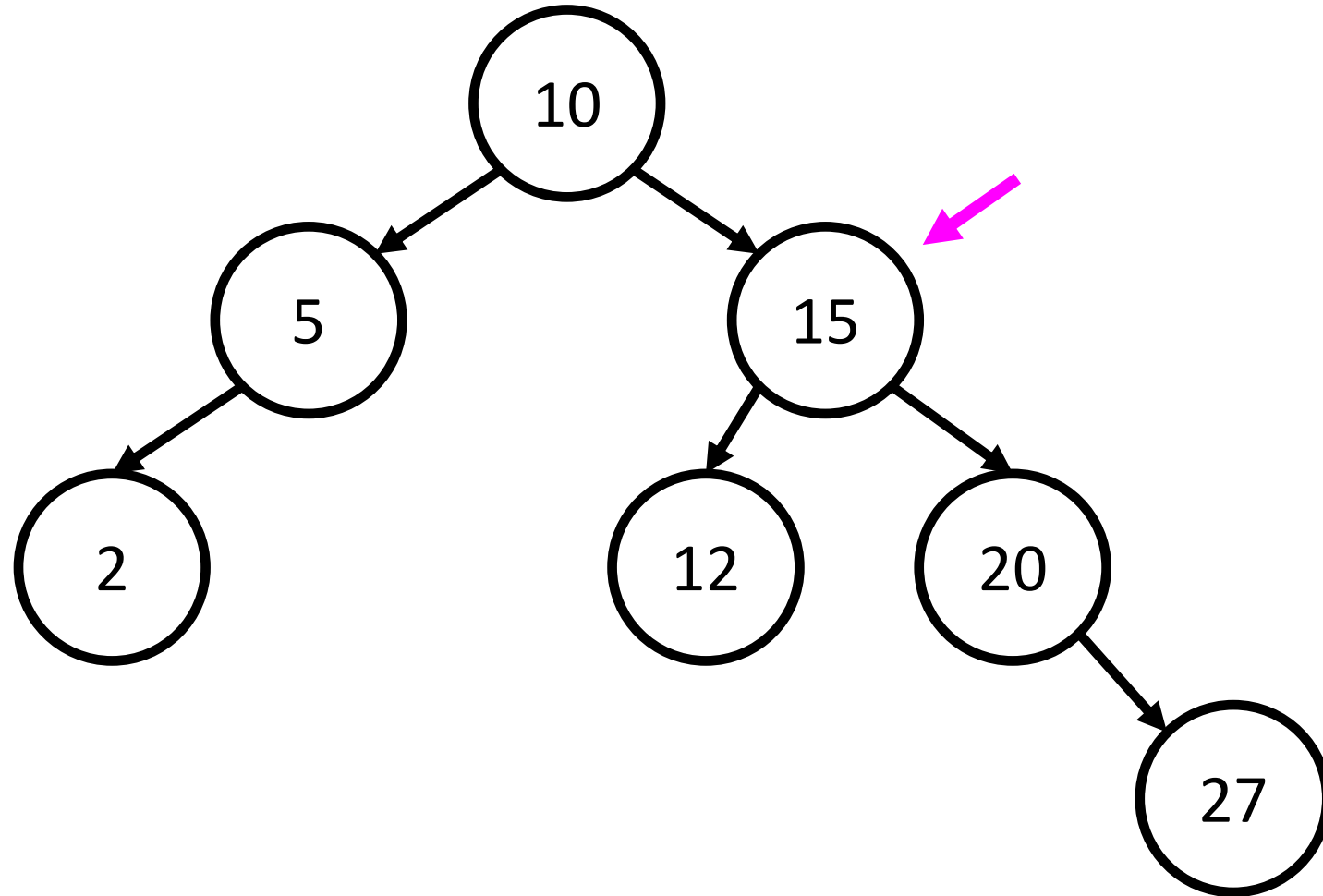
Exemplo: busca por 20



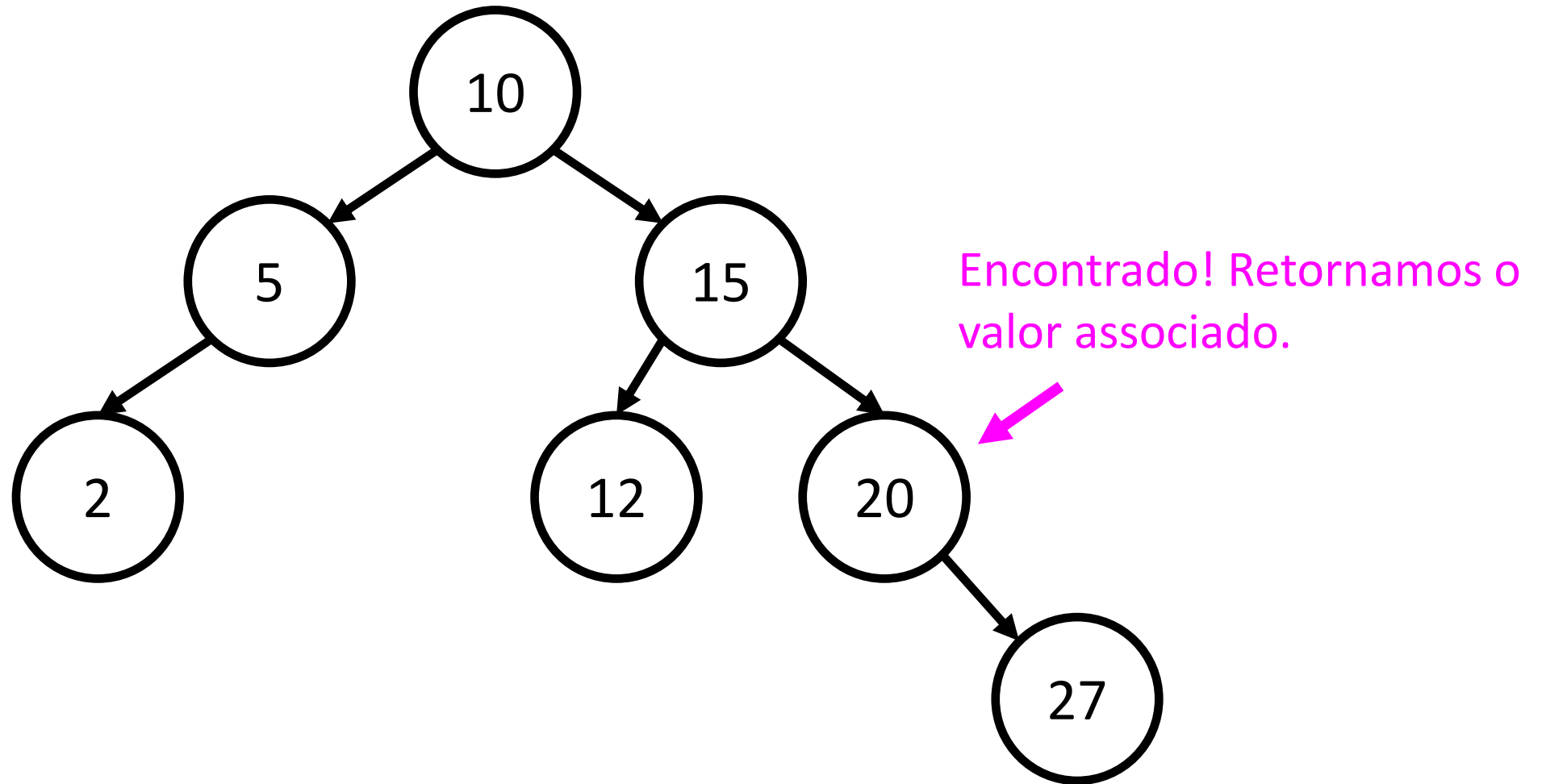
Exemplo: busca por 20



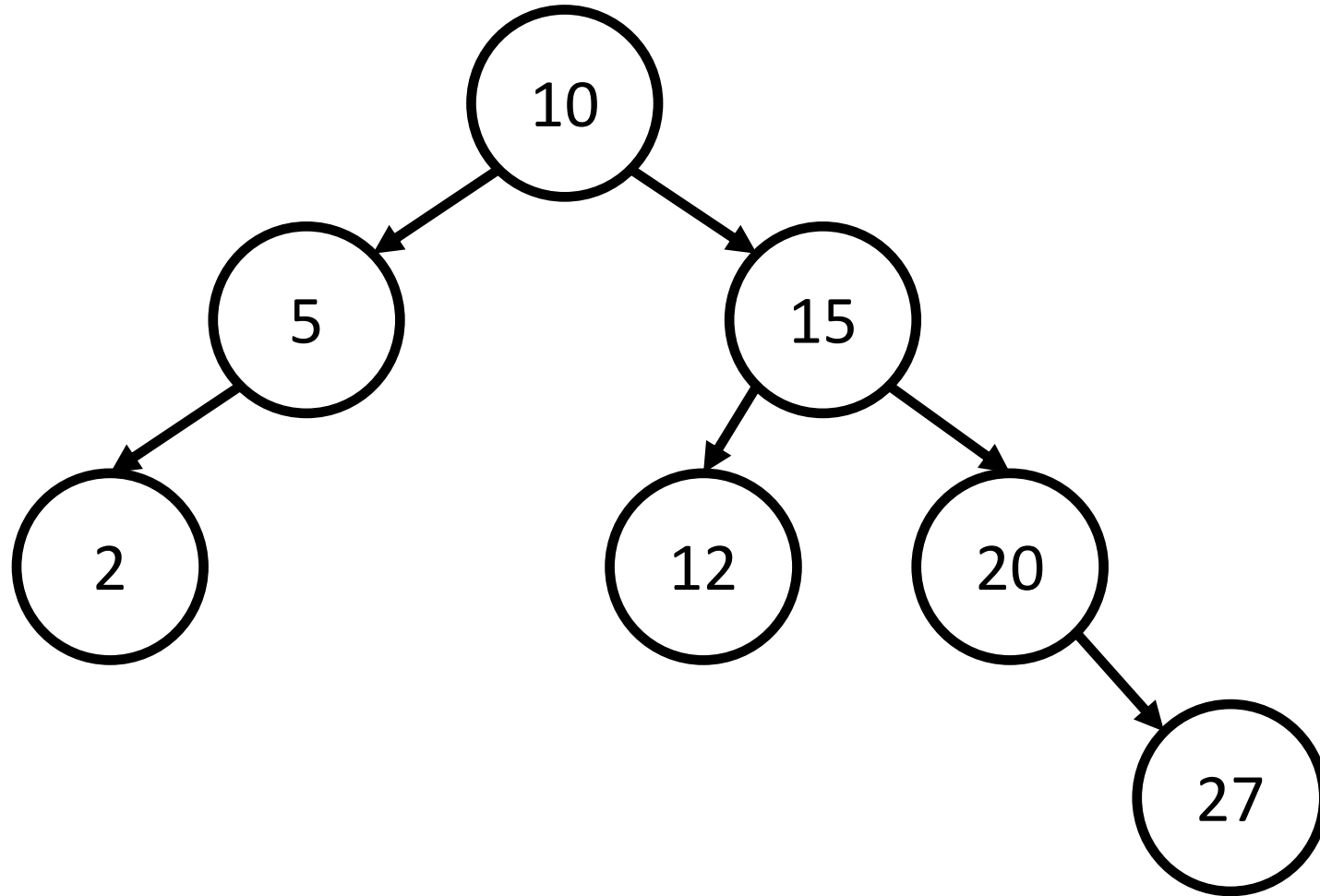
Exemplo: busca por 20



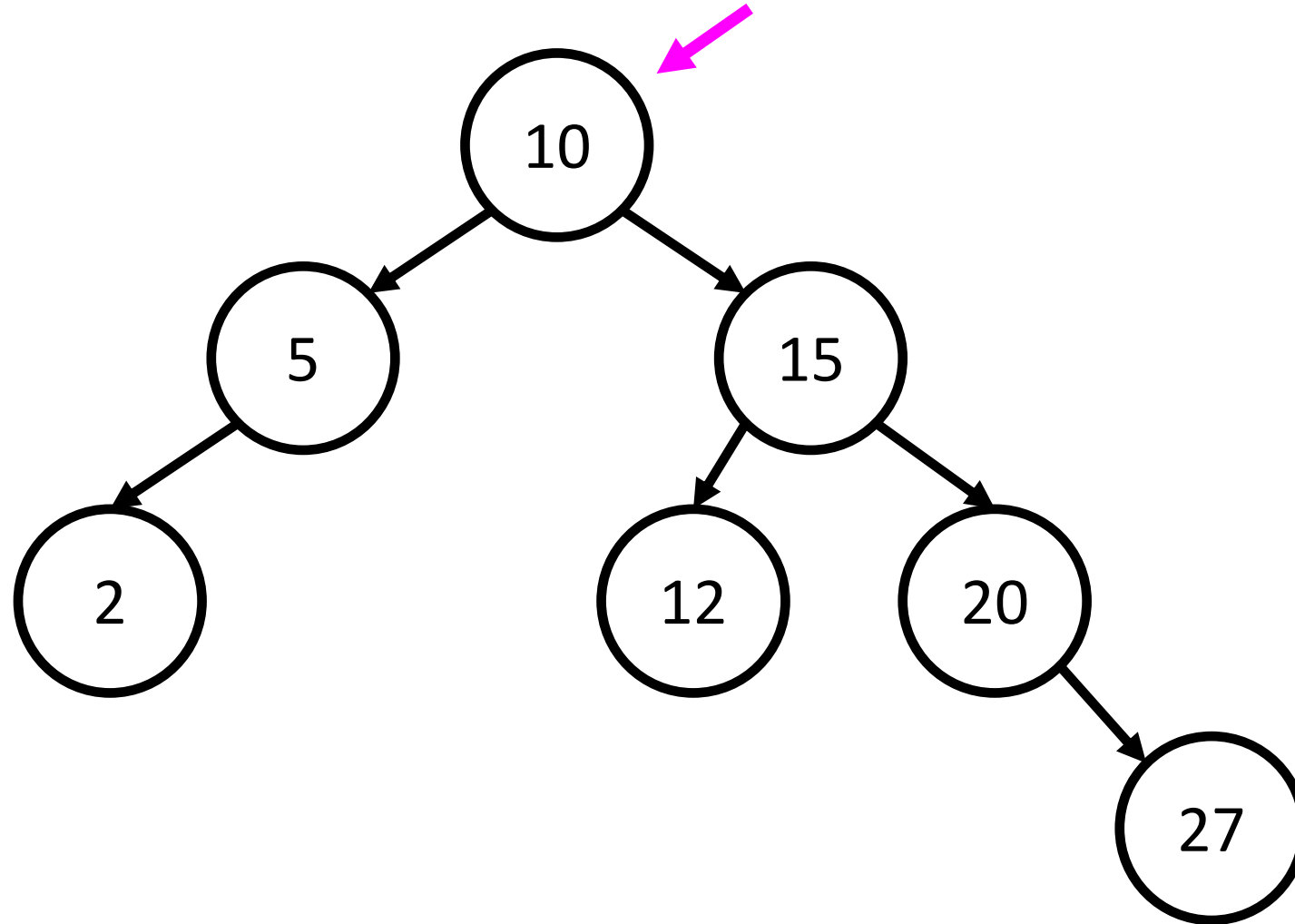
Exemplo: busca por 20



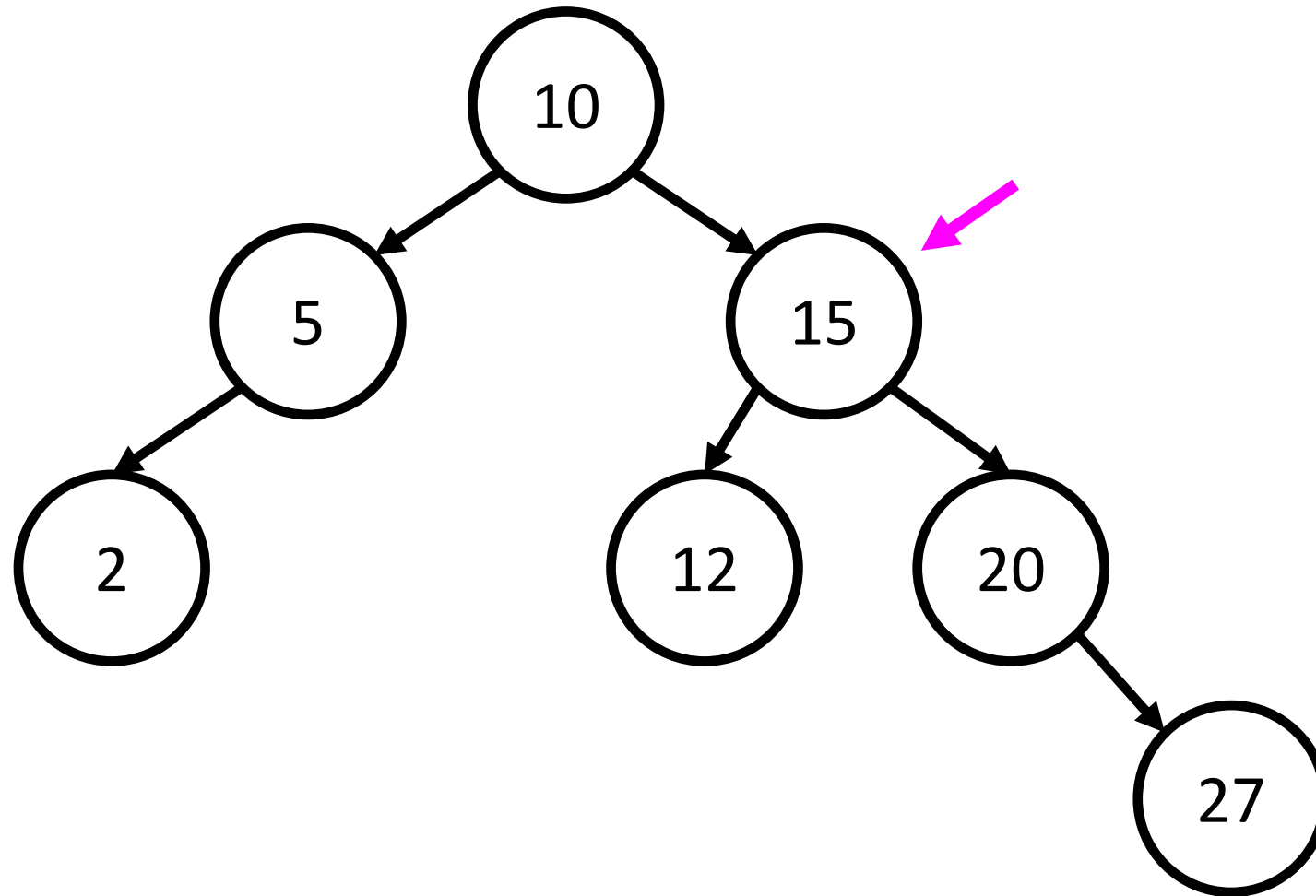
Exemplo: busca por 13



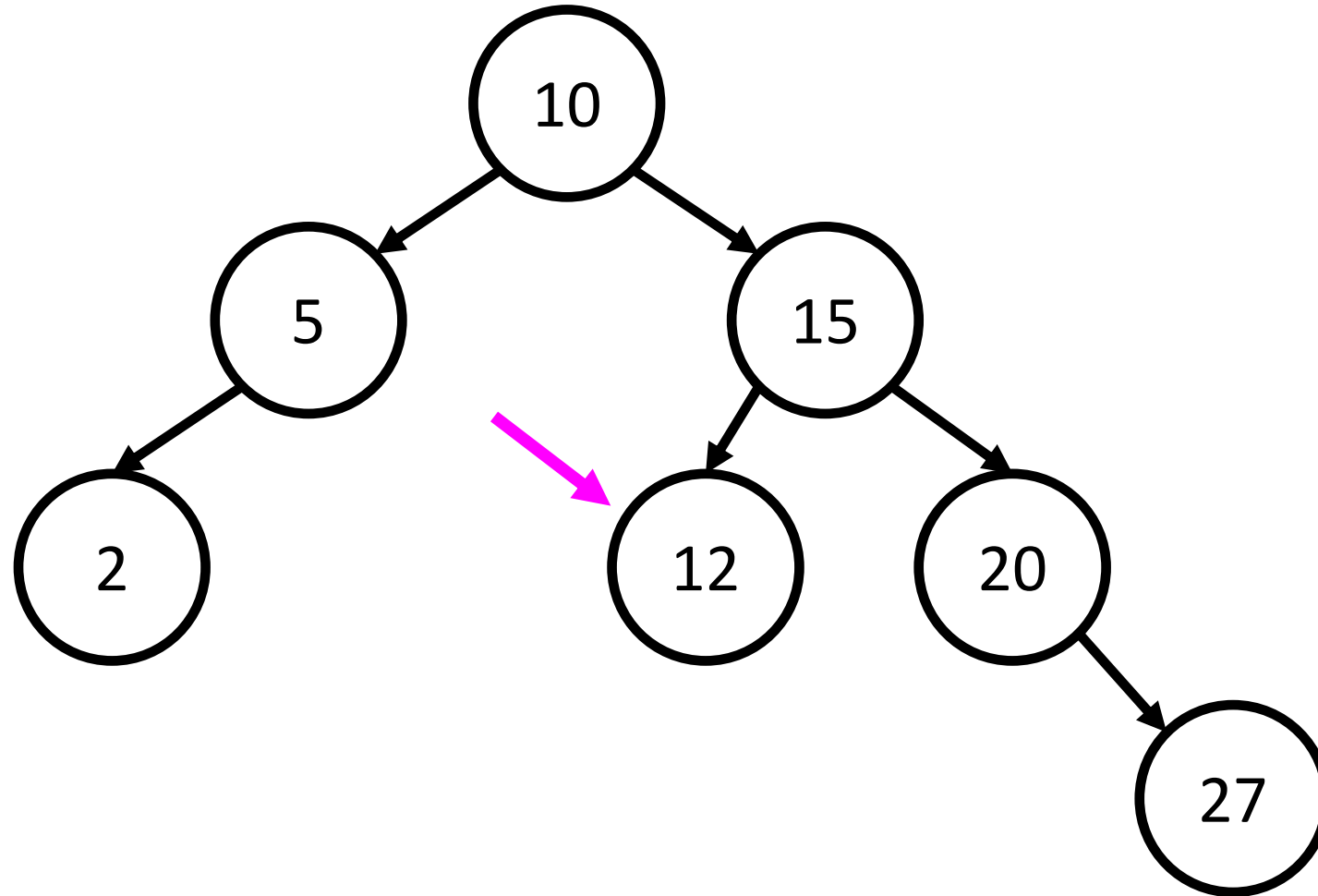
Exemplo: busca por 13



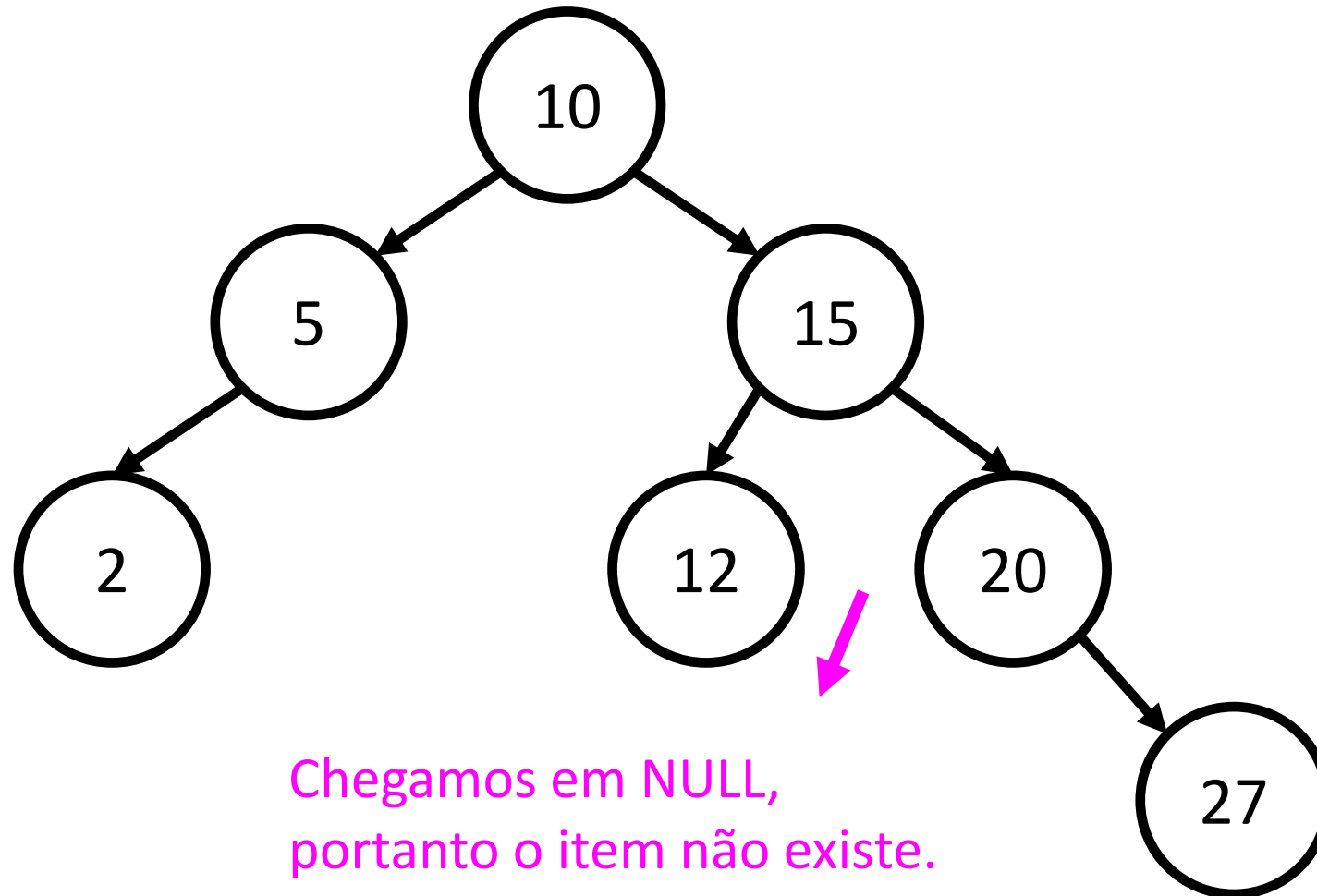
Exemplo: busca por 13



Exemplo: busca por 13



Exemplo: busca por 13



O algoritmo de busca pode ser implementado de forma iterativa ou recursiva. Nos pseudocódigos abaixo, x é um nó (inicialmente a raiz) e k é a chave buscada. As funções retornam o nó que contém a chave ou NULL se ela não existir.

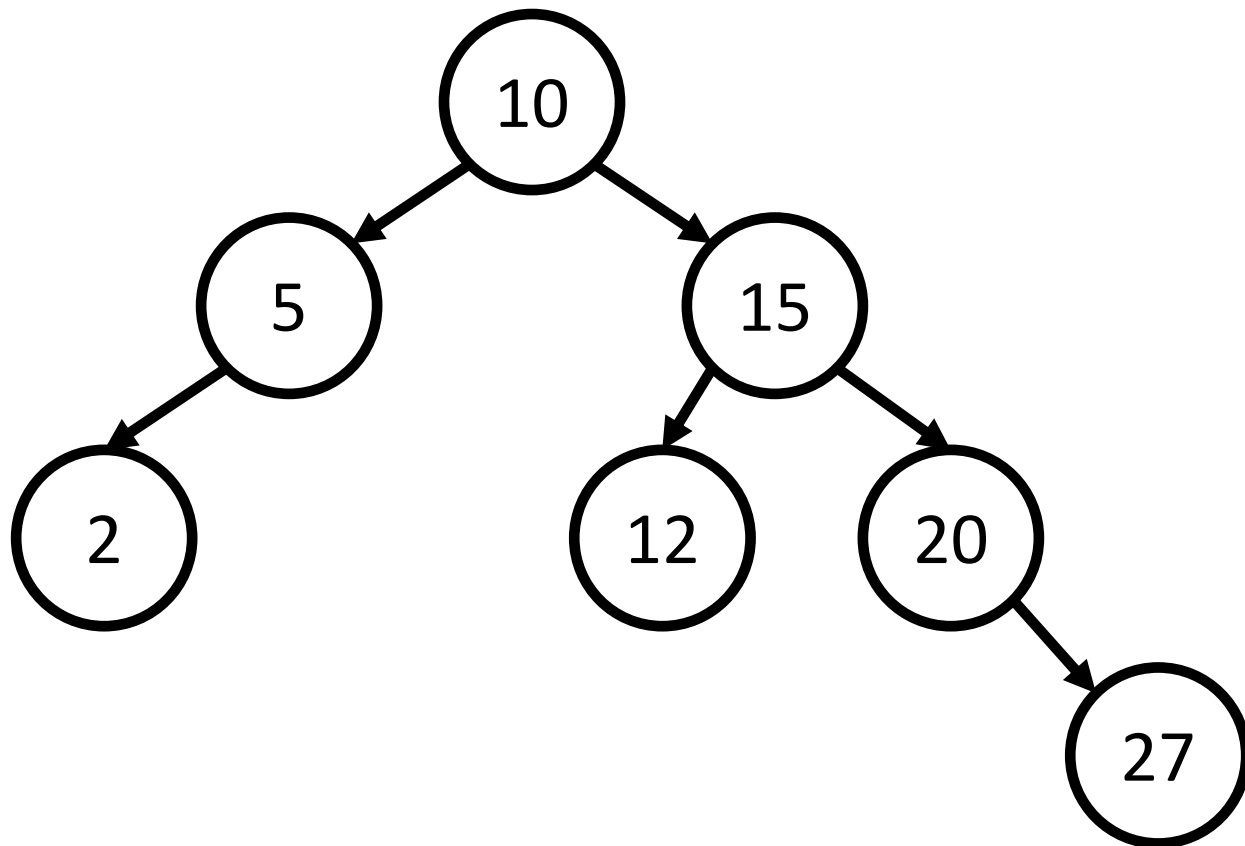
TREE-SEARCH(x, k)

```
1  if  $x == \text{NIL}$  ou  $k == x.chave$ 
2      return  $x$ 
3  if  $k < x.chave$ 
4      return TREE-SEARCH( $x.esquerda, k$ )
5  else return TREE-SEARCH( $x.direita, k$ )
```

ITERATIVE-TREE-SEARCH(x, k)

```
1  while  $x \neq \text{NIL}$  e  $k \neq x.chave$ 
2      if  $k < x.chave$ 
3           $x = x.esquerda$ 
4      else  $x = x.direita$ 
5  return  $x$ 
```

Máximo / Mínimo: O máximo será o elemento mais à direita da árvore, enquanto o mínimo será o elemento mais à esquerda.



TREE-MINIMUM(x)

```
1  while  $x.esquerda \neq \text{NIL}$   
2       $x = x.esquerda$   
3  return  $x$ 
```

TREE-MAXIMUM(x)

```
1  while  $x.direita \neq \text{NIL}$   
2       $x = x.direita$   
3  return  $x$ 
```

Inserção

Lógica básica da inserção:

- Desça na árvore como se estivesse fazendo uma busca até chegar à uma folha.
- Crie um novo nó na esquerda se o novo item for menor que a folha ou na direita, caso contrário.

Inserção – Versão Recursiva

- Se a árvore está vazia, nós retornamos um novo nó contendo o item.
- Se a chave sendo inserida é menor que a chave na raiz, definimos o nó da esquerda como o resultado da inserção do item na subárvore da esquerda.
- Caso contrário, definimos o nó da direita como o resultado da inserção do item na subárvore da direita.

Inserção – Versão Recursiva

```
Node *_add_recursive(Node *node, key_type *key, data_type value) {  
    if (node == NULL)  
        return node_construct(key, value, NULL, NULL);  
  
    if (strcmp(key, node->key) < 0)  
        node->left = _add_recursive(node->left, key, value);  
    else  
        node->right = _add_recursive(node->right, key, value);  
  
    return node;  
}  
  
void binary_tree_add(BinaryTree *bt,  
                    key_type *key, data_type value) {  
    bt->root = _add_recursive(bt->root, key, value);  
}
```

BinaryTree	
raiz	✗

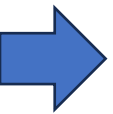
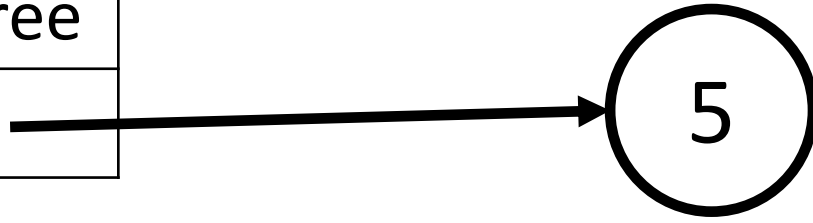
INSERT 5
INSERT 3
INSERT 7
INSERT 1
INSERT 2
INSERT 6

```
rec(n, key, value)
  if n == NULL
    return new node(key, value);

  if key < n->key
    n->left = rec(n->left,
                  key, value)
  else
    n->right = rec(n->right,
                   key, value)
  return n

add(bt, key, val)
  root = rec(root, key, val)
```

BinaryTree	
raiz	



INSERT 5

INSERT 3

INSERT 7

INSERT 1

INSERT 2

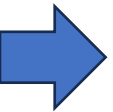
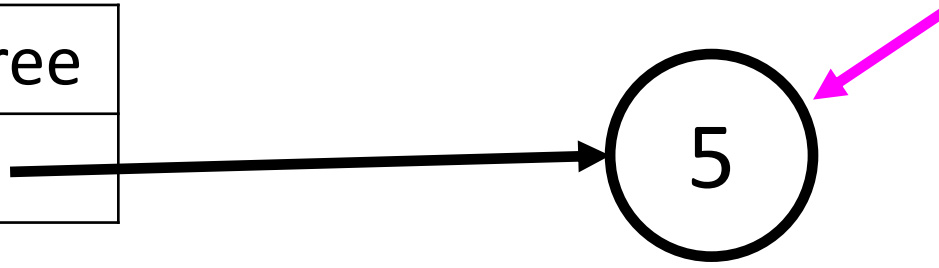
INSERT 6

```
rec(n, key, value)
  if n == NULL
    return new node(key, value);

  if key < n->key
    n->left = rec(n->left,
                  key, value)
  else
    n->right = rec(n->right,
                   key, value)
  return n

add(bt, key, val)
  root = rec(root, key, val)
```

BinaryTree	
raiz	



INSERT 5

INSERT 3

INSERT 7

INSERT 1

INSERT 2

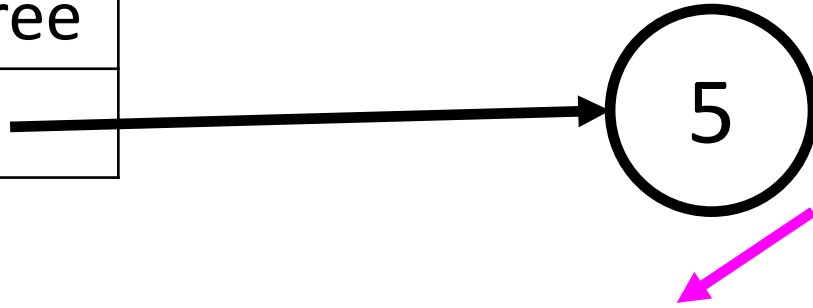
INSERT 6

```
rec(n, key, value)
if n == NULL
    return new node(key, value);

if key < n->key
    n->left = rec(n->left,
                  key, value)
else
    n->right = rec(n->right,
                   key, value)
return n

add(bt, key, val)
root = rec(root, key, val)
```

BinaryTree	
raiz	



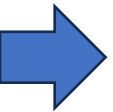
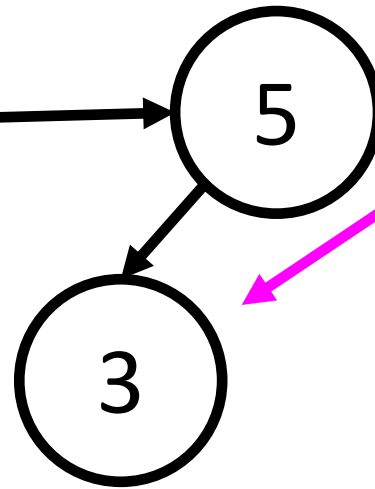
→ INSERT 5
INSERT 3
INSERT 7
INSERT 1
INSERT 2
INSERT 6

```
rec(n, key, value)
if n == NULL
    return new node(key, value);

if key < n->key
    n->left = rec(n->left,
                  key, value)
else
    n->right = rec(n->right,
                   key, value)
return n

add(bt, key, val)
root = rec(root, key, val)
```

BinaryTree	
raiz	



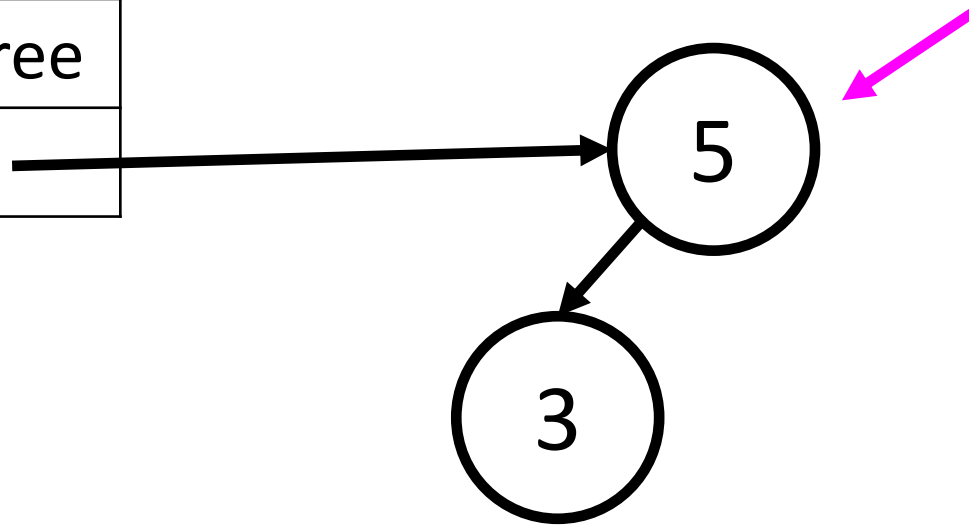
INSERT 5
INSERT 3
INSERT 7
INSERT 1
INSERT 2
INSERT 6

```
rec(n, key, value)
  if n == NULL
    return new node(key, value);

  if key < n->key
    n->left = rec(n->left,
                  key, value)
  else
    n->right = rec(n->right,
                  key, value)
  return n

add(bt, key, val)
  root = rec(root, key, val)
```

BinaryTree	
raiz	



INSERT 5

INSERT 3

➡ INSERT 7

INSERT 1

INSERT 2

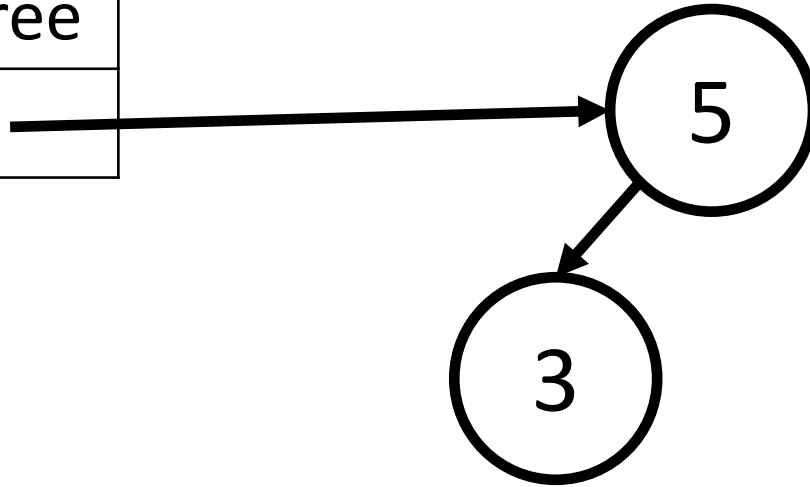
INSERT 6

```
rec(n, key, value)
  if n == NULL
    return new node(key, value);

  if key < n->key
    n->left = rec(n->left,
                  key, value)
  else
    n->right = rec(n->right,
                  key, value)
  return n

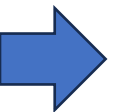
add(bt, key, val)
  root = rec(root, key, val)
```

BinaryTree	
raiz	



INSERT 5

INSERT 3



INSERT 7

INSERT 1

INSERT 2

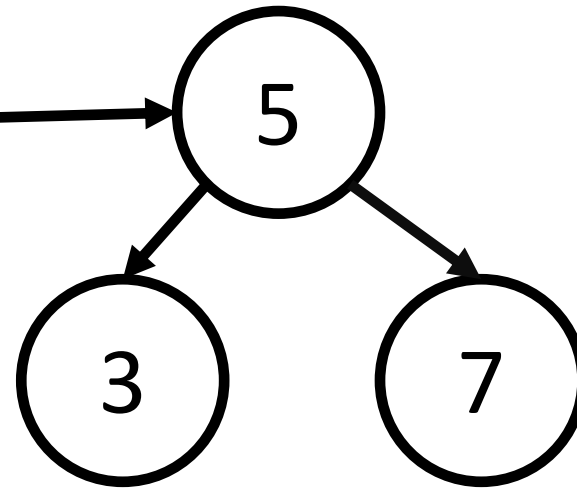
INSERT 6

```
rec(n, key, value)
if n == NULL
    return new node(key, value);

if key < n->key
    n->left = rec(n->left,
                  key, value)
else
    n->right = rec(n->right,
                  key, value)
return n

add(bt, key, val)
root = rec(root, key, val)
```


BinaryTree	
raiz	



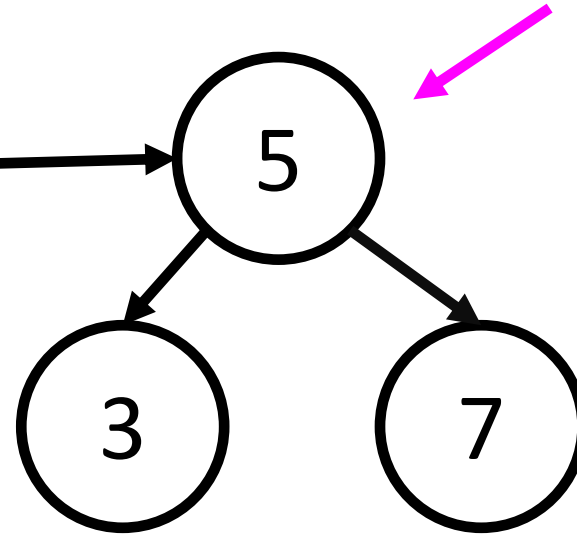
```
rec(n, key, value)
if n == NULL
    return new node(key, value);

if key < n->key
    n->left = rec(n->left,
                  key, value)
else
    n->right = rec(n->right,
                  key, value)
return n

add(bt, key, val)
root = rec(root, key, val)
```

INSERT 5
INSERT 3
INSERT 7
INSERT 1
INSERT 2
INSERT 6

BinaryTree	
raiz	



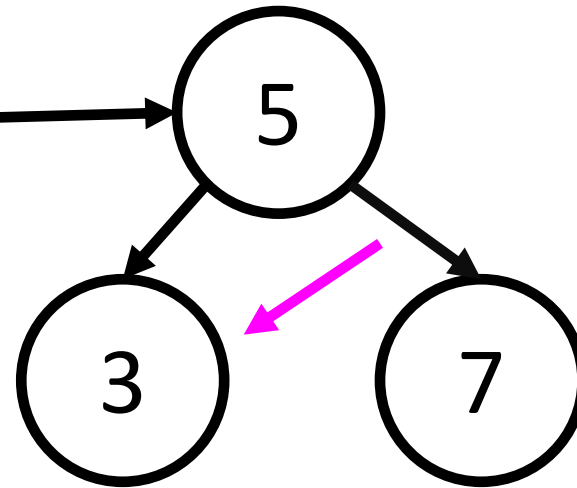
INSERT 5
INSERT 3
INSERT 7
→ INSERT 1
INSERT 2
INSERT 6

```
rec(n, key, value)
  if n == NULL
    return new node(key, value);

  if key < n->key
    n->left = rec(n->left,
                  key, value)
  else
    n->right = rec(n->right,
                   key, value)
  return n

add(bt, key, val)
  root = rec(root, key, val)
```

BinaryTree	
raiz	



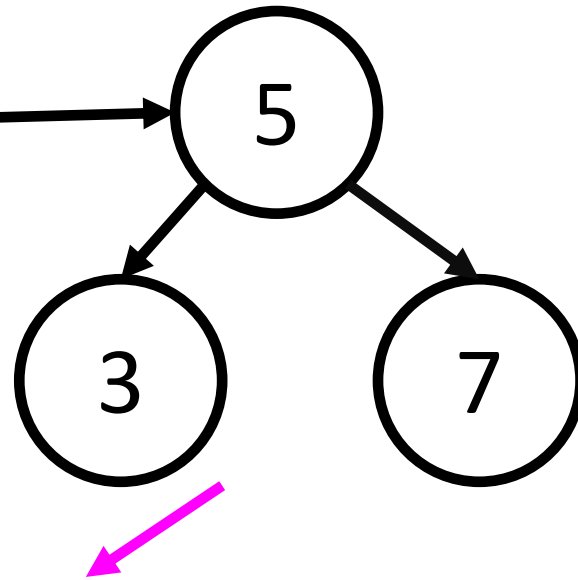
INSERT 5
INSERT 3
INSERT 7
→ INSERT 1
INSERT 2
INSERT 6

```
rec(n, key, value)
  if n == NULL
    return new node(key, value);

  if key < n->key
    n->left = rec(n->left,
                  key, value)
  else
    n->right = rec(n->right,
                  key, value)
  return n

add(bt, key, val)
  root = rec(root, key, val)
```

BinaryTree	
raiz	



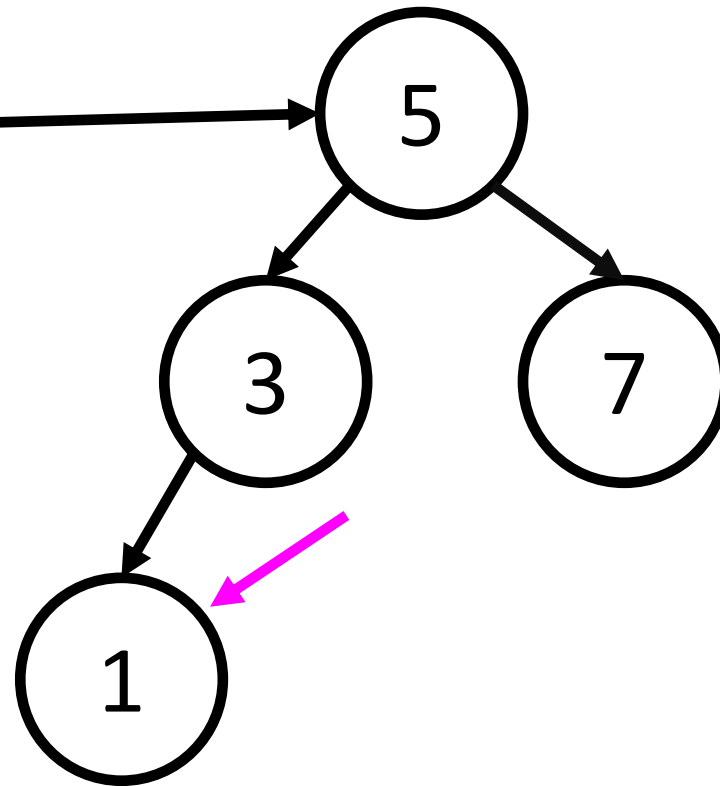
INSERT 5
INSERT 3
INSERT 7
→ INSERT 1
INSERT 2
INSERT 6

```
rec(n, key, value)
  if n == NULL
    return new node(key, value);

  if key < n->key
    n->left = rec(n->left,
                  key, value)
  else
    n->right = rec(n->right,
                   key, value)
  return n

add(bt, key, val)
  root = rec(root, key, val)
```

BinaryTree	
raiz	



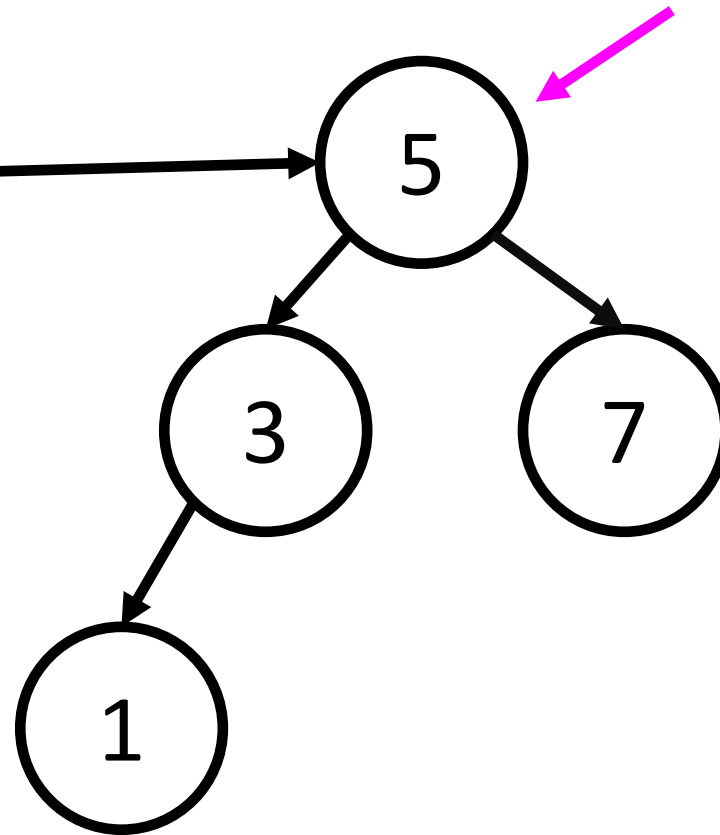
INSERT 5
INSERT 3
INSERT 7
INSERT 1
INSERT 2
INSERT 6

```
rec(n, key, value)
  if n == NULL
    return new node(key, value);

  if key < n->key
    n->left = rec(n->left,
                  key, value)
  else
    n->right = rec(n->right,
                   key, value)
  return n

add(bt, key, val)
  root = rec(root, key, val)
```

BinaryTree	
raiz	



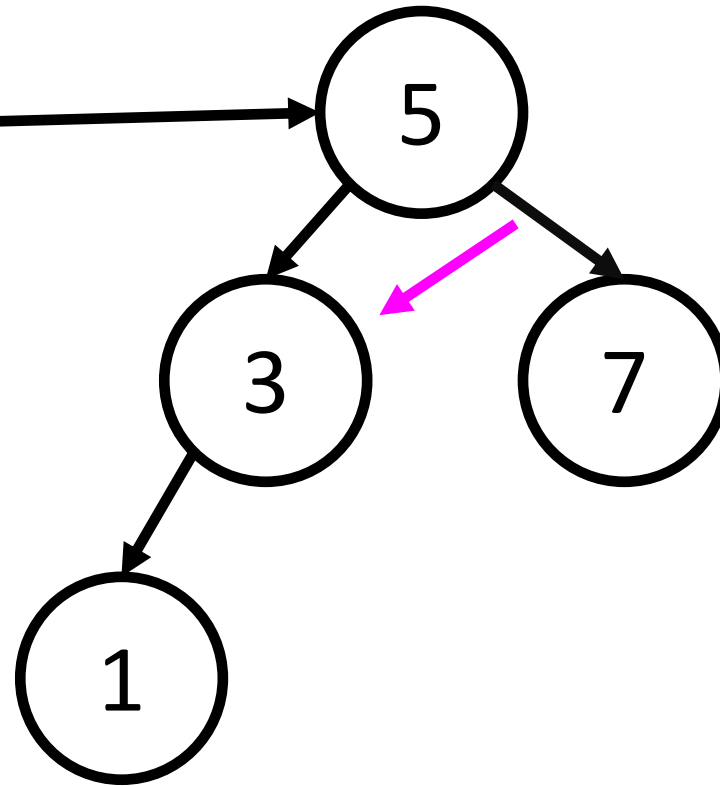
INSERT 5
INSERT 3
INSERT 7
INSERT 1
→ INSERT 2
INSERT 6

```
rec(n, key, value)
if n == NULL
    return new node(key, value);

if key < n->key
    n->left = rec(n->left,
                  key, value)
else
    n->right = rec(n->right,
                  key, value)
return n

add(bt, key, val)
root = rec(root, key, val)
```

BinaryTree	
raiz	



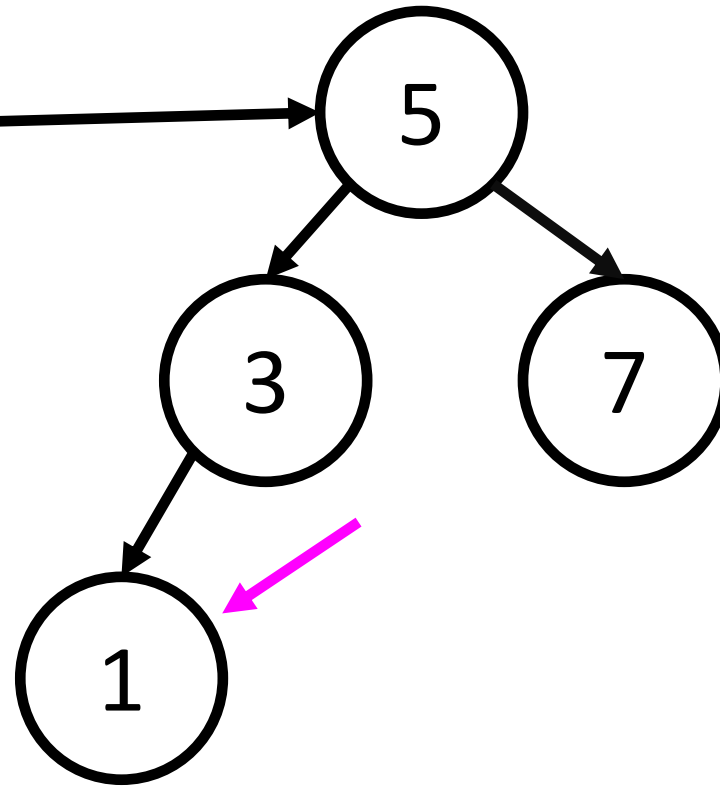
INSERT 5
INSERT 3
INSERT 7
INSERT 1
➡ INSERT 2
INSERT 6

```
rec(n, key, value)
  if n == NULL
    return new node(key, value);

  if key < n->key
    n->left = rec(n->left,
                  key, value)
  else
    n->right = rec(n->right,
                   key, value)
  return n

add(bt, key, val)
  root = rec(root, key, val)
```

BinaryTree	
raiz	



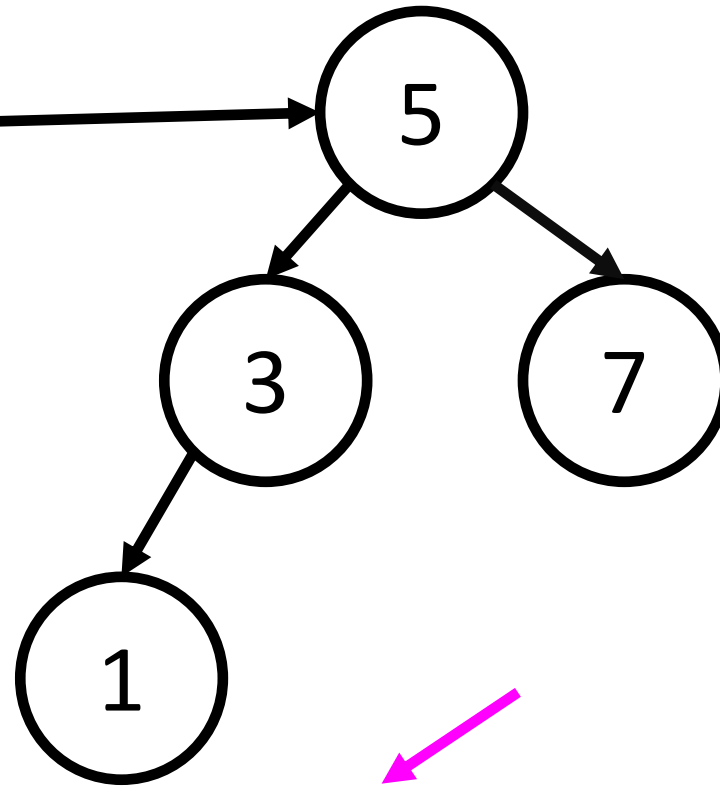
INSERT 5
INSERT 3
INSERT 7
INSERT 1
→ INSERT 2
INSERT 6

```
rec(n, key, value)
  if n == NULL
    return new node(key, value);

  if key < n->key
    n->left = rec(n->left,
                  key, value)
  else
    n->right = rec(n->right,
                  key, value)
  return n

add(bt, key, val)
  root = rec(root, key, val)
```


BinaryTree	
raiz	



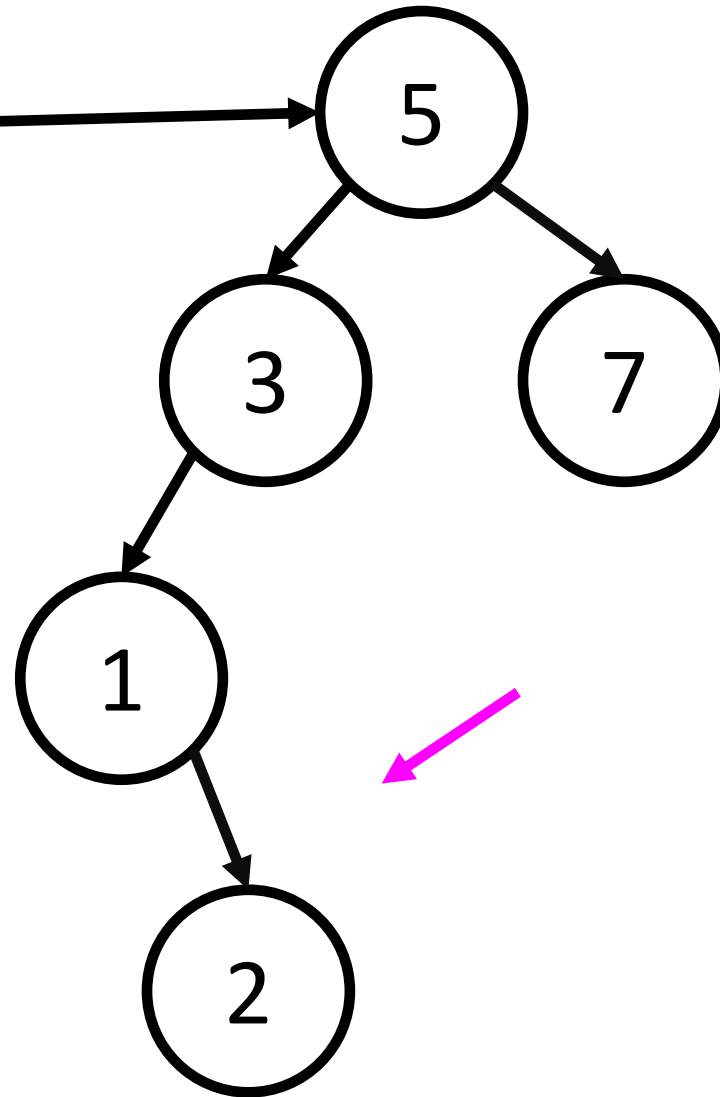
INSERT 5
INSERT 3
INSERT 7
INSERT 1
→ INSERT 2
INSERT 6

```
rec(n, key, value)
if n == NULL
    return new node(key, value);

if key < n->key
    n->left = rec(n->left,
                  key, value)
else
    n->right = rec(n->right,
                  key, value)
return n

add(bt, key, val)
root = rec(root, key, val)
```

BinaryTree	
raiz	



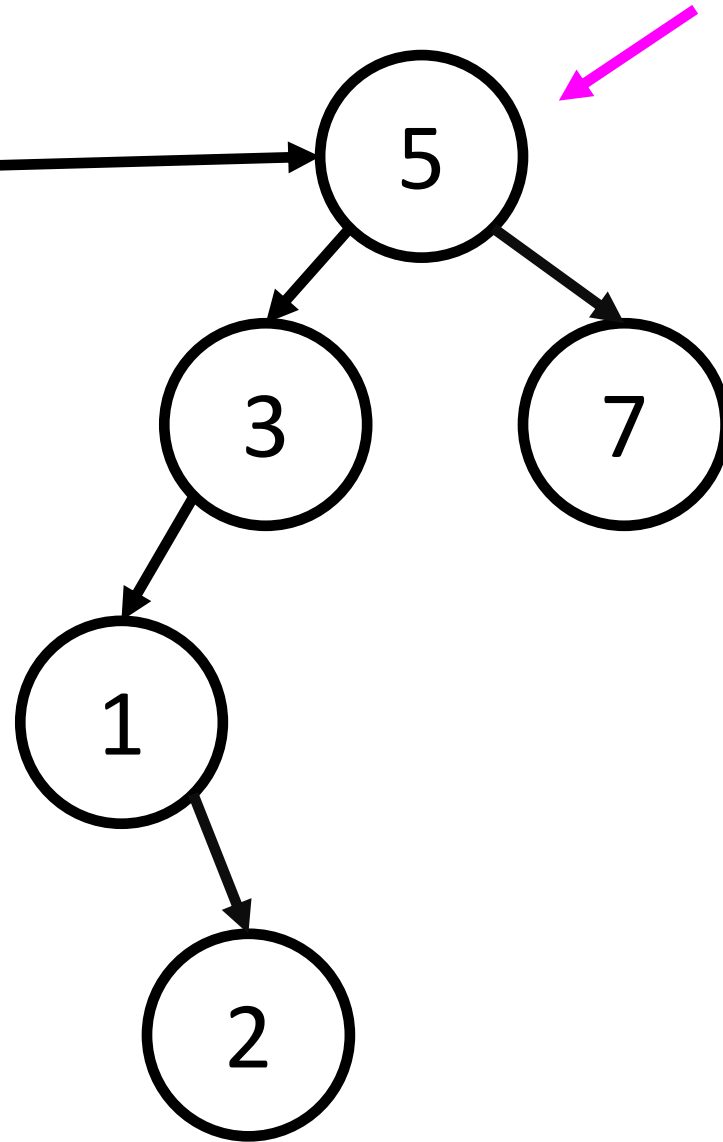
INSERT 5
INSERT 3
INSERT 7
INSERT 1
INSERT 2
INSERT 6

```
rec(n, key, value)
if n == NULL
    return new node(key, value);

if key < n->key
    n->left = rec(n->left,
                  key, value)
else
    n->right = rec(n->right,
                  key, value)
return n

add(bt, key, val)
root = rec(root, key, val)
```

BinaryTree	
raiz	



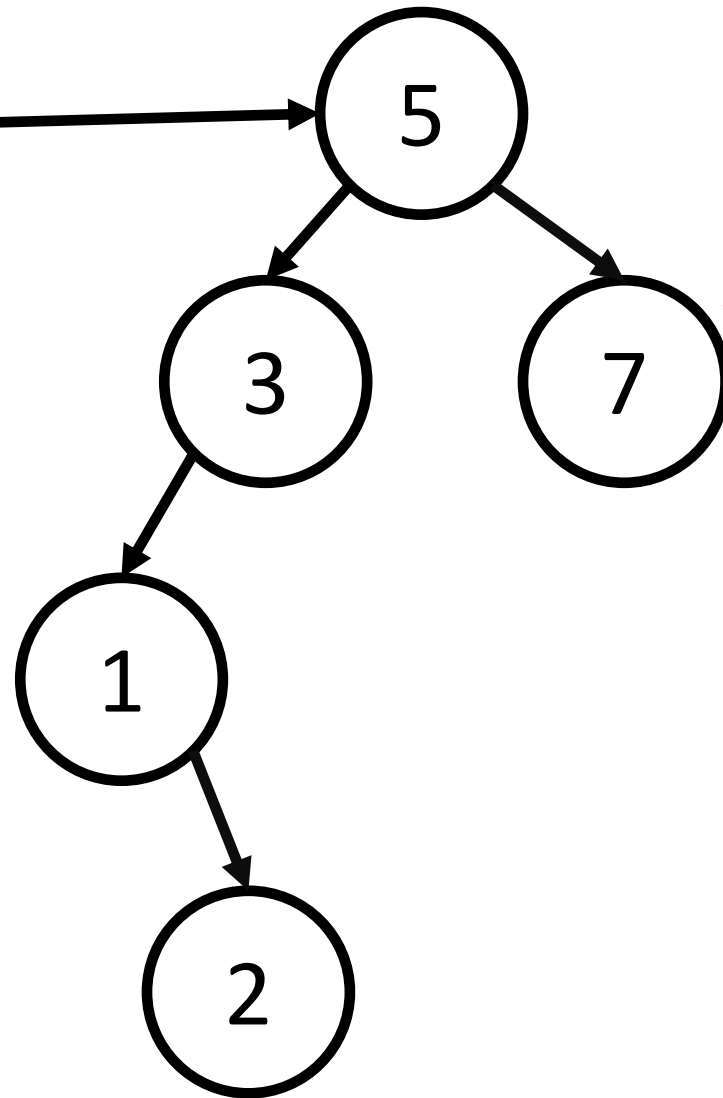
INSERT 5
INSERT 3
INSERT 7
INSERT 1
INSERT 2
→ INSERT 6

```
rec(n, key, value)
if n == NULL
    return new node(key, value);

if key < n->key
    n->left = rec(n->left,
                  key, value)
else
    n->right = rec(n->right,
                  key, value)
return n

add(bt, key, val)
root = rec(root, key, val)
```

BinaryTree	
raiz	



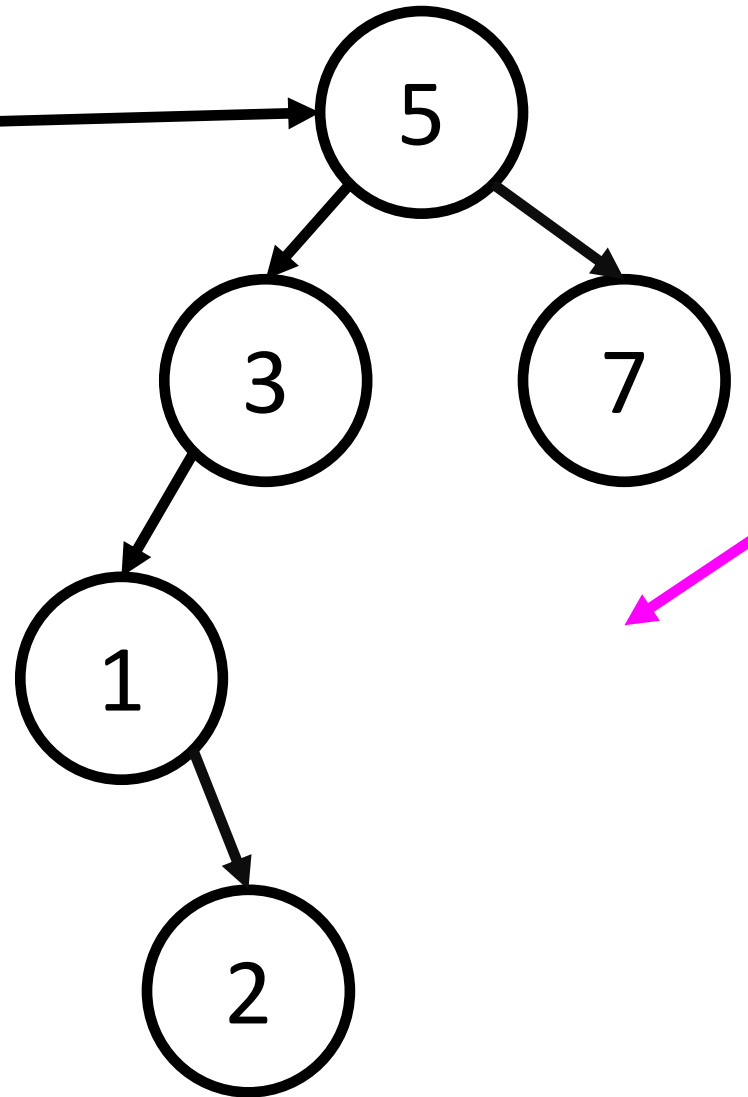
```
rec(n, key, value)
if n == NULL
    return new node(key, value);

if key < n->key
    n->left = rec(n->left,
                  key, value)
else
    n->right = rec(n->right,
                  key, value)
return n

add(bt, key, val)
root = rec(root, key, val)
```

INSERT 5
INSERT 3
INSERT 7
INSERT 1
INSERT 2
→ INSERT 6

BinaryTree	
raiz	

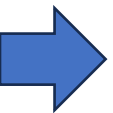


```
rec(n, key, value)
if n == NULL
    return new node(key, value);

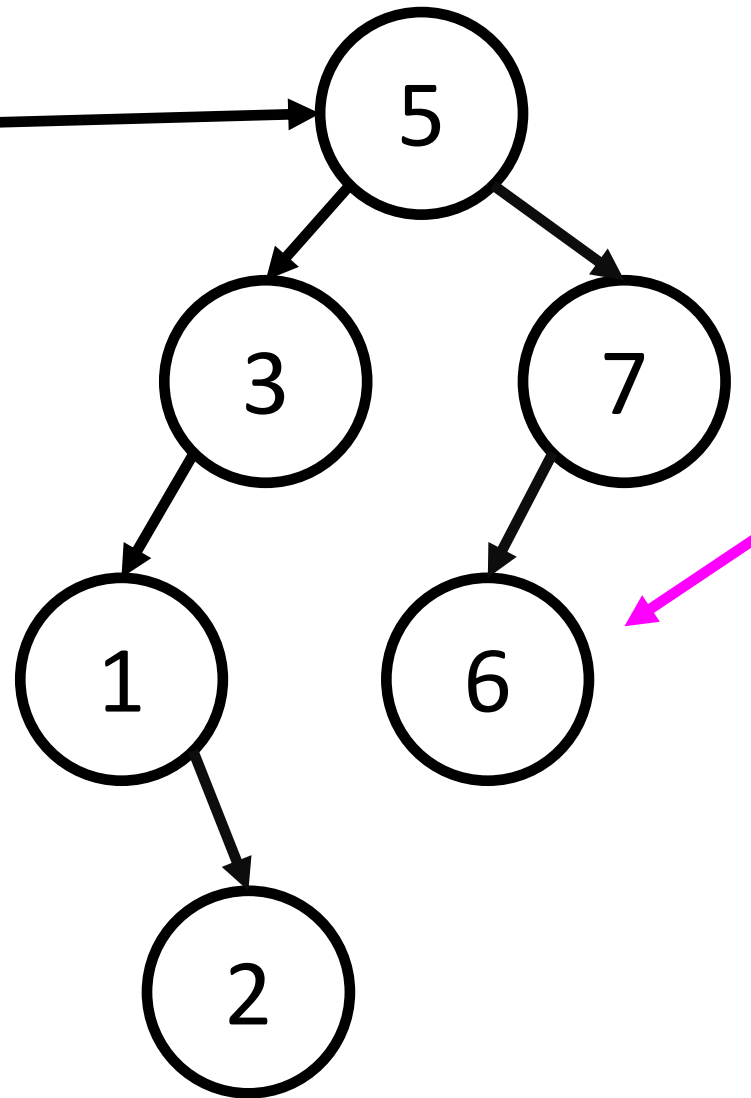
if key < n->key
    n->left = rec(n->left,
                  key, value)
else
    n->right = rec(n->right,
                  key, value)
return n

add(bt, key, val)
root = rec(root, key, val)
```

INSERT 5
INSERT 3
INSERT 7
INSERT 1
INSERT 2
INSERT 6



BinaryTree	
raiz	

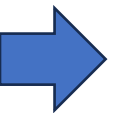


```
rec(n, key, value)
if n == NULL
    return new node(key, value);

if key < n->key
    n->left = rec(n->left,
                  key, value)
else
    n->right = rec(n->right,
                  key, value)
return n

add(bt, key, val)
root = rec(root, key, val)
```

INSERT 5
INSERT 3
INSERT 7
INSERT 1
INSERT 2
INSERT 6



Inserção – Versão Iterativa

- A dificuldade da inserção é que o pai do novo nó precisa ser modificado.
- Para viabilizar esta mudança, vamos precisar descer pela árvore usando dois ponteiros, um para o nó e outro para o pai.
- Assim que nó for NULL, modificamos o pai para que o filho seja o novo nó inserido.

TREE-INSERT(T, z)

$y = \text{NIL}$ ← Y é o ponteiro para o pai

$x = T.\text{raiz}$ ← X é o ponteiro para o nó atual

while $x \neq \text{NIL}$

$y = x$

if $z.\text{chave} < x.\text{chave}$

$x = x.\text{esquerda}$

else $x = x.\text{direita}$

Enquanto x não for nulo, desça na árvore procurando o local onde o item deve ser inserido.

if $y = \text{NIL}$

$T.\text{raiz} = z$ // a árvore T era vazia

Se o pai é nulo, a árvore era vazia, então crie um novo nó e defina como a raiz

else if $z.\text{chave} < y.\text{chave}$

$y.\text{esquerda} = z$

else $y.\text{direita} = z$

Caso contrário, se a chave for menor que a do pai, insira o novo item na esquerda. Senão, insira na direita.

BinaryTree	
raiz	✗

INSERT 5
INSERT 3
INSERT 7
INSERT 1
INSERT 2
INSERT 6

TREE-INSERT(T, z)

$y = \text{NIL}$

$x = T.\text{raiz}$

while $x \neq \text{NIL}$

$y = x$

if $z.\text{chave} < x.\text{chave}$

$x = x.\text{esquerda}$

else $x = x.\text{direita}$

if $y = \text{NIL}$

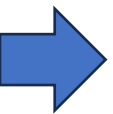
$T.\text{raiz} = z$

else if $z.\text{chave} < y.\text{chave}$

$y.\text{esquerda} = z$

else $y.\text{direita} = z$

BinaryTree	
raiz	✗



INSERT 5
 INSERT 3
 INSERT 7
 INSERT 1
 INSERT 2
 INSERT 6

TREE-INSERT(T, z)

$y = \text{NIL}$

$x = T.\text{raiz}$

while $x \neq \text{NIL}$

$y = x$

if $z.\text{chave} < x.\text{chave}$

$x = x.\text{esquerda}$

else $x = x.\text{direita}$

if $y = \text{NIL}$

$T.\text{raiz} = z$

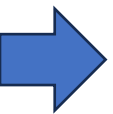
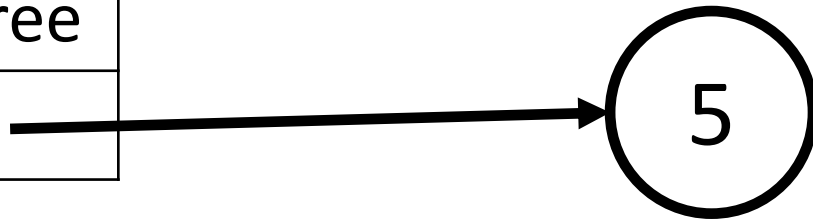
else if $z.\text{chave} < y.\text{chave}$

$y.\text{esquerda} = z$

else $y.\text{direita} = z$

x	NULL
y	NULL

BinaryTree	
raiz	



INSERT 5
 INSERT 3
 INSERT 7
 INSERT 1
 INSERT 2
 INSERT 6

TREE-INSERT(T, z)

$y = \text{NIL}$

$x = T.\text{raiz}$

while $x \neq \text{NIL}$

$y = x$

if $z.\text{chave} < x.\text{chave}$

$x = x.\text{esquerda}$

else $x = x.\text{direita}$

if $y = \text{NIL}$


$T.\text{raiz} = z$

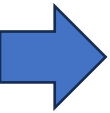
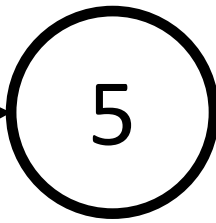
else if $z.\text{chave} < y.\text{chave}$

$y.\text{esquerda} = z$

else $y.\text{direita} = z$

x	NULL
y	NULL

BinaryTree	
raiz	



INSERT 5
 INSERT 3
 INSERT 7
 INSERT 1
 INSERT 2
 INSERT 6

TREE-INSERT(T, z)

$y = \text{NIL}$

$x = T.\text{raiz}$

while $x \neq \text{NIL}$

$y = x$

if $z.\text{chave} < x.\text{chave}$

$x = x.\text{esquerda}$

else $x = x.\text{direita}$

if $y = \text{NIL}$

$T.\text{raiz} = z$

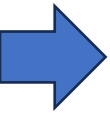
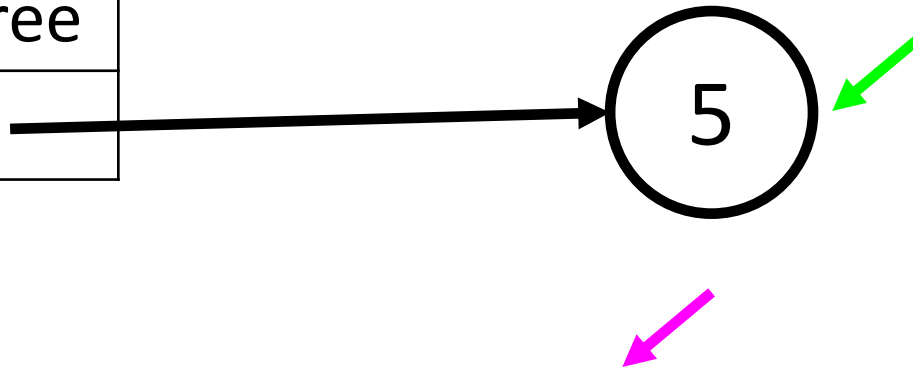
else if $z.\text{chave} < y.\text{chave}$

$y.\text{esquerda} = z$

else $y.\text{direita} = z$

x	5
y	NULL

BinaryTree	
raiz	



INSERT 5
 INSERT 3
 INSERT 7
 INSERT 1
 INSERT 2
 INSERT 6

TREE-INSERT(T, z)

$y = \text{NIL}$

$x = T.\text{raiz}$

while $x \neq \text{NIL}$

$y = x$

if $z.\text{chave} < x.\text{chave}$

$x = x.\text{esquerda}$

else $x = x.\text{direita}$

if $y = \text{NIL}$

$T.\text{raiz} = z$

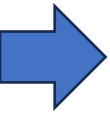
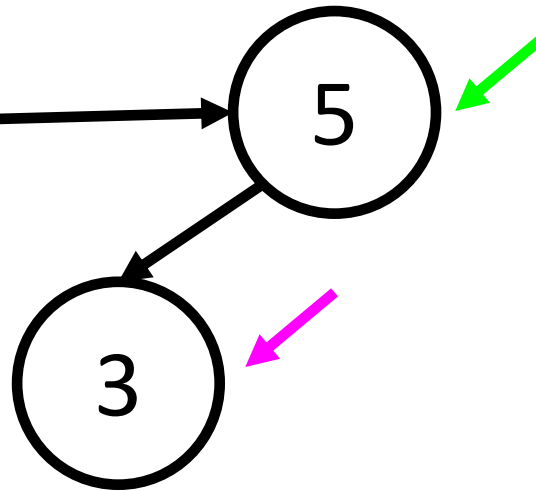
else if $z.\text{chave} < y.\text{chave}$

$y.\text{esquerda} = z$

else $y.\text{direita} = z$

x	NULL
y	5

BinaryTree	
raiz	



INSERT 5
 INSERT 3
 INSERT 7
 INSERT 1
 INSERT 2
 INSERT 6

TREE-INSERT(T, z)

$y = \text{NIL}$

$x = T.\text{raiz}$

while $x \neq \text{NIL}$

$y = x$

if $z.\text{chave} < x.\text{chave}$

$x = x.\text{esquerda}$

else $x = x.\text{direita}$

if $y = \text{NIL}$


$T.\text{raiz} = z$

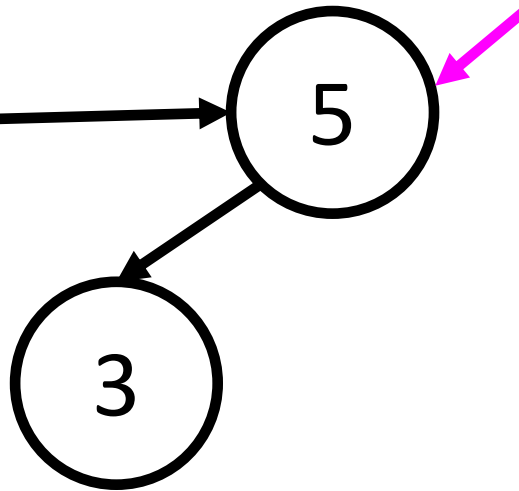
else if $z.\text{chave} < y.\text{chave}$

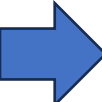
$y.\text{esquerda} = z$

else $y.\text{direita} = z$

x	NULL
y	5

BinaryTree	
raiz	



INSERT 5
 INSERT 3
 INSERT 7
 INSERT 1
 INSERT 2
 INSERT 6

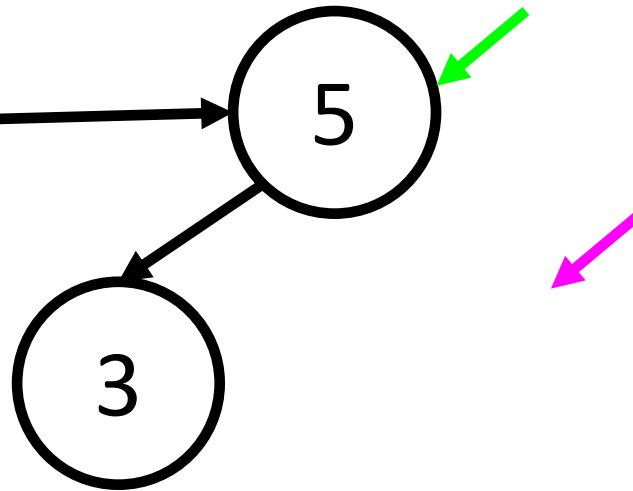
TREE-INSERT(T, z)

```

 $y = \text{NIL}$ 
 $x = T.\text{raiz}$ 
while  $x \neq \text{NIL}$ 
     $y = x$ 
    if  $z.\text{chave} < x.\text{chave}$ 
         $x = x.\text{esquerda}$ 
    else  $x = x.\text{direita}$ 
if  $y = \text{NIL}$ 
     $T.\text{raiz} = z$ 
else if  $z.\text{chave} < y.\text{chave}$ 
     $y.\text{esquerda} = z$ 
else  $y.\text{direita} = z$ 
  
```

x	5
y	NULL

BinaryTree	
raiz	



INSERT 5
 INSERT 3
 INSERT 7
 INSERT 1
 INSERT 2
 INSERT 6

TREE-INSERT(T, z)

$y = \text{NIL}$

$x = T.\text{raiz}$

while $x \neq \text{NIL}$

$y = x$

if $z.\text{chave} < x.\text{chave}$

$x = x.\text{esquerda}$

else $x = x.\text{direita}$

if $y = \text{NIL}$


$T.\text{raiz} = z$

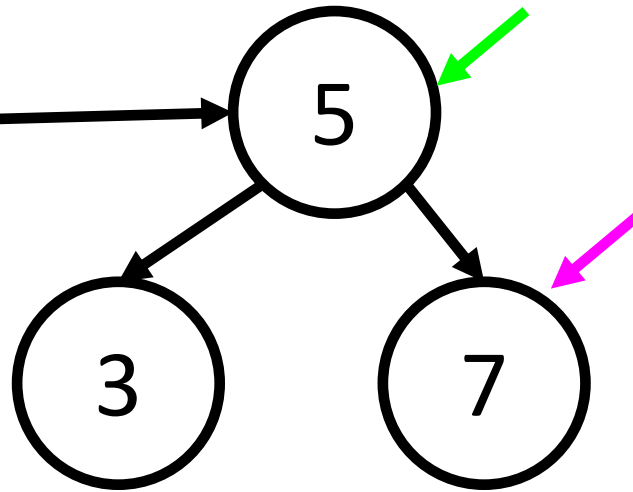
else if $z.\text{chave} < y.\text{chave}$

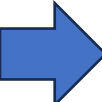
$y.\text{esquerda} = z$

else $y.\text{direita} = z$

x	NULL
y	5

BinaryTree	
raiz	



INSERT 5
 INSERT 3
 INSERT 7
 INSERT 1
 INSERT 2
 INSERT 6

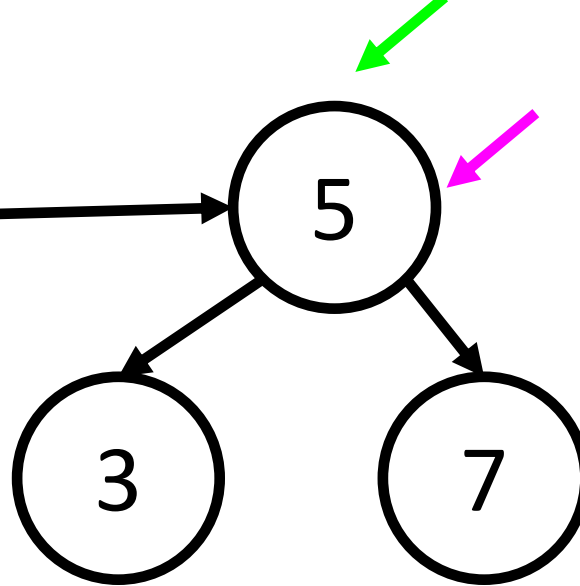
TREE-INSERT(T, z)

```

 $y = \text{NIL}$ 
 $x = T.\text{raiz}$ 
while  $x \neq \text{NIL}$ 
     $y = x$ 
    if  $z.\text{chave} < x.\text{chave}$ 
         $x = x.\text{esquerda}$ 
    else  $x = x.\text{direita}$ 
if  $y = \text{NIL}$ 
     $T.\text{raiz} = z$ 
else if  $z.\text{chave} < y.\text{chave}$ 
     $y.\text{esquerda} = z$ 
else  $y.\text{direita} = z$ 
  
```

x	NULL
y	5

BinaryTree	
raiz	



INSERT 5
 INSERT 3
 INSERT 7
 INSERT 1
 INSERT 2
 INSERT 6

TREE-INSERT(T, z)

$y = \text{NIL}$

$x = T.\text{raiz}$

while $x \neq \text{NIL}$

$y = x$

if $z.\text{chave} < x.\text{chave}$

$x = x.\text{esquerda}$

else $x = x.\text{direita}$

if $y = \text{NIL}$

$T.\text{raiz} = z$

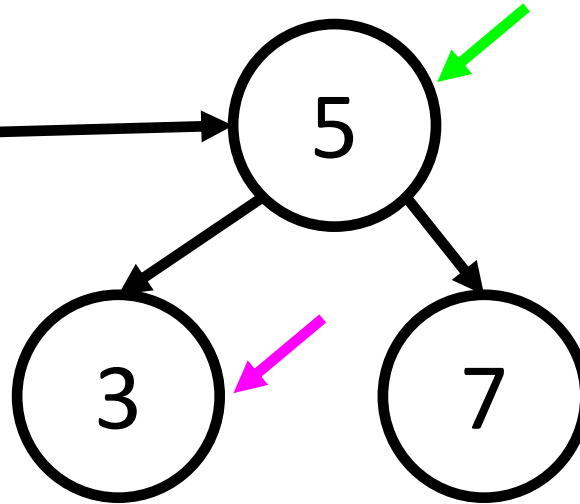
else if $z.\text{chave} < y.\text{chave}$


$y.\text{esquerda} = z$

else $y.\text{direita} = z$

x	5
y	NULL

BinaryTree	
raiz	



INSERT 5
 INSERT 3
 INSERT 7
 INSERT 1
 INSERT 2
 INSERT 6

TREE-INSERT(T, z)

$y = \text{NIL}$

$x = T.\text{raiz}$

while $x \neq \text{NIL}$

$y = x$

if $z.\text{chave} < x.\text{chave}$

$x = x.\text{esquerda}$

else $x = x.\text{direita}$

if $y = \text{NIL}$

$T.\text{raiz} = z$

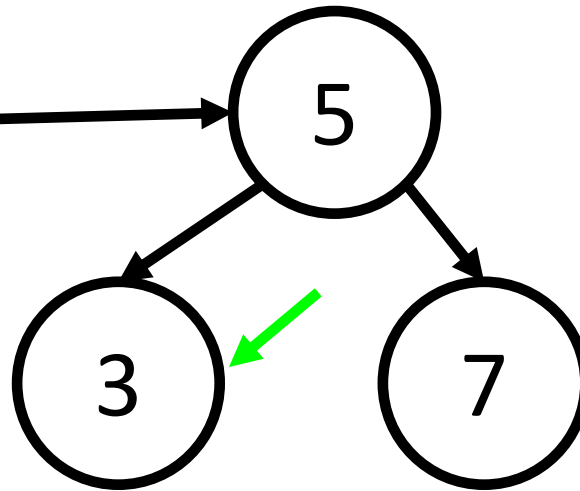
else if $z.\text{chave} < y.\text{chave}$

$y.\text{esquerda} = z$

else $y.\text{direita} = z$

x	3
y	5

BinaryTree	
raiz	



INSERT 5
 INSERT 3
 INSERT 7
 INSERT 1
 INSERT 2
 INSERT 6

TREE-INSERT(T, z)

$y = \text{NIL}$

$x = T.\text{raiz}$

while $x \neq \text{NIL}$

$y = x$

if $z.\text{chave} < x.\text{chave}$

$x = x.\text{esquerda}$

else $x = x.\text{direita}$

if $y = \text{NIL}$

$T.\text{raiz} = z$

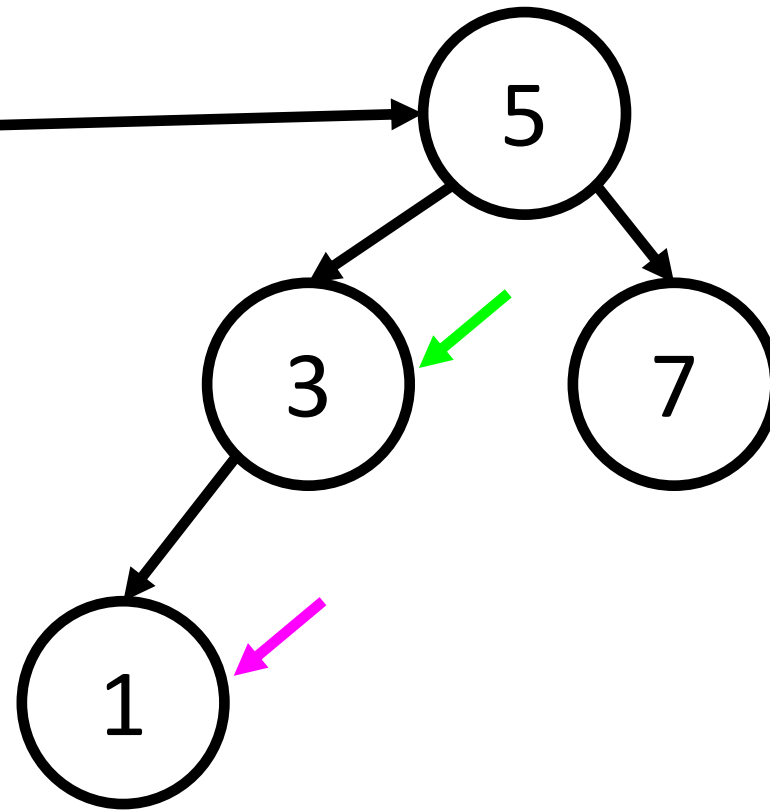
else if $z.\text{chave} < y.\text{chave}$

$y.\text{esquerda} = z$

else $y.\text{direita} = z$

x	NULL
y	3

BinaryTree	
raiz	



INSERT 5
 INSERT 3
 INSERT 7
 INSERT 1
 INSERT 2
 INSERT 6

TREE-INSERT(T, z)

$y = \text{NIL}$

$x = T.\text{raiz}$

while $x \neq \text{NIL}$

$y = x$

if $z.\text{chave} < x.\text{chave}$

$x = x.\text{esquerda}$

else $x = x.\text{direita}$

if $y = \text{NIL}$


$T.\text{raiz} = z$

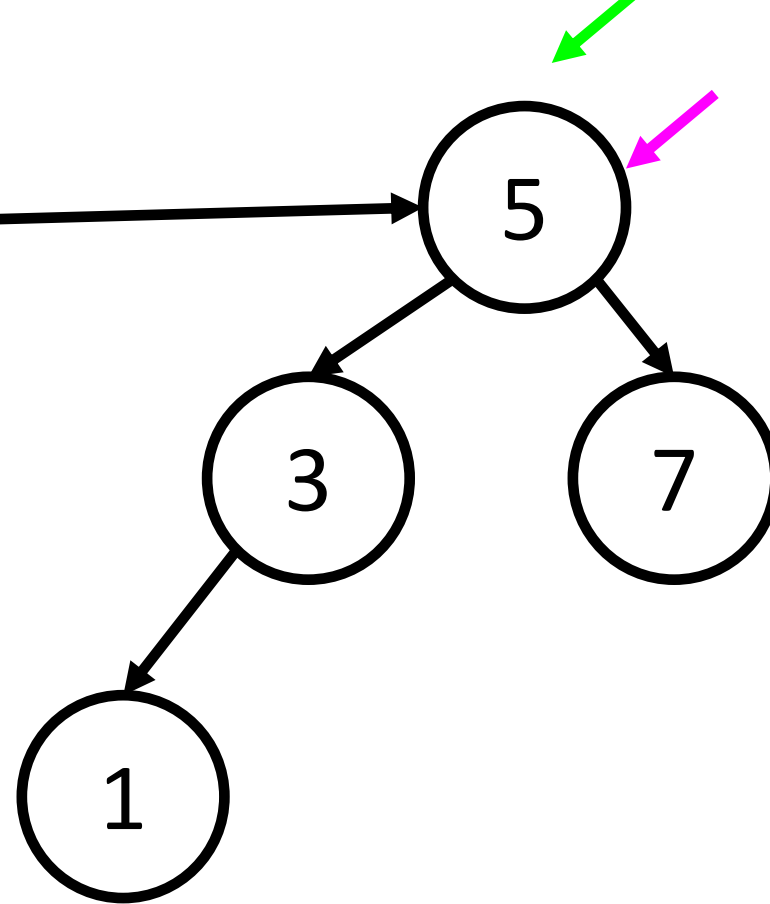
else if $z.\text{chave} < y.\text{chave}$

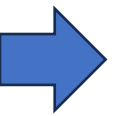
$y.\text{esquerda} = z$

else $y.\text{direita} = z$

x	NULL
y	3

BinaryTree	
raiz	



INSERT 5
 INSERT 3
 INSERT 7
 INSERT 1
 INSERT 2
 INSERT 6

TREE-INSERT(T, z)

$y = \text{NIL}$

$x = T.\text{raiz}$

while $x \neq \text{NIL}$

$y = x$

if $z.\text{chave} < x.\text{chave}$

$x = x.\text{esquerda}$

else $x = x.\text{direita}$

if $y = \text{NIL}$


$T.\text{raiz} = z$

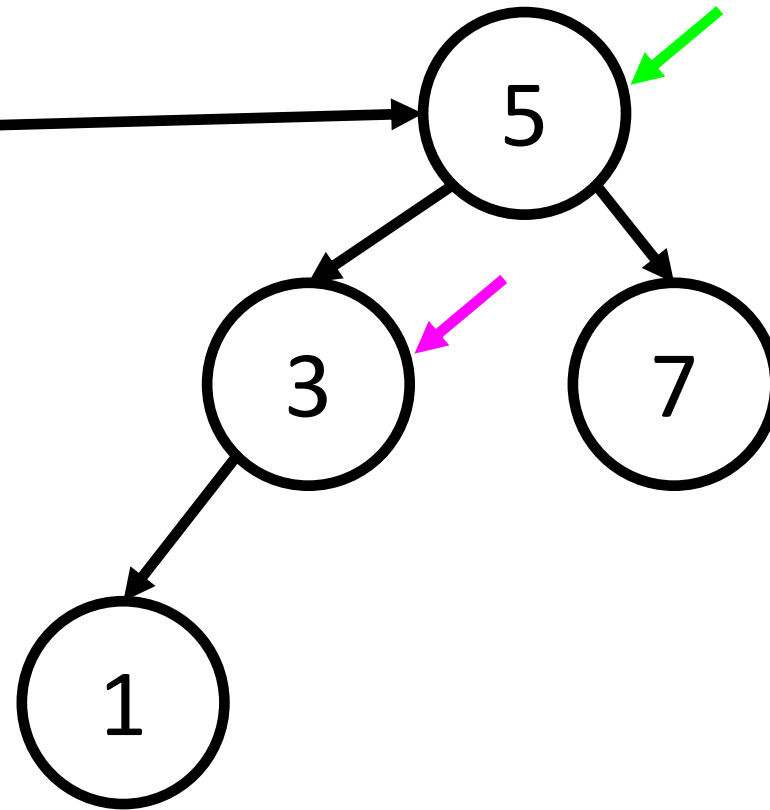
else if $z.\text{chave} < y.\text{chave}$

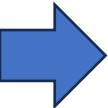
$y.\text{esquerda} = z$

else $y.\text{direita} = z$

x	5
y	NULL

BinaryTree	
raiz	



INSERT 5
 INSERT 3
 INSERT 7
 INSERT 1
 INSERT 2
 INSERT 6

TREE-INSERT(T, z)

$y = \text{NIL}$

$x = T.\text{raiz}$

while $x \neq \text{NIL}$

$y = x$

if $z.\text{chave} < x.\text{chave}$

$x = x.\text{esquerda}$

else $x = x.\text{direita}$

if $y = \text{NIL}$

$T.\text{raiz} = z$

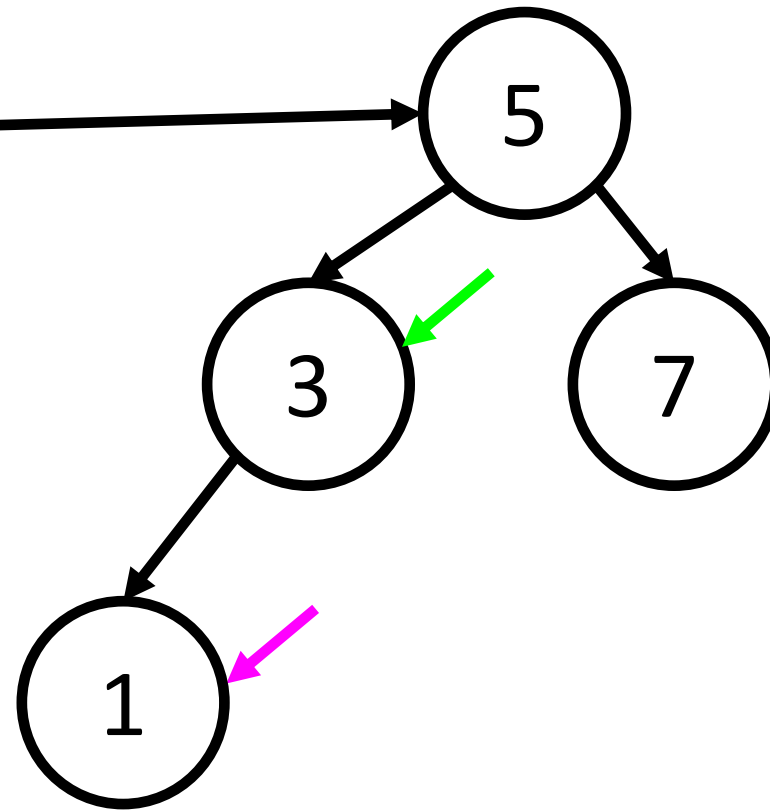
else if $z.\text{chave} < y.\text{chave}$

$y.\text{esquerda} = z$

else $y.\text{direita} = z$

x	3
y	5

BinaryTree	
raiz	



INSERT 5
 INSERT 3
 INSERT 7
 INSERT 1
 INSERT 2
 INSERT 6

TREE-INSERT(T, z)

$y = \text{NIL}$

$x = T.\text{raiz}$

while $x \neq \text{NIL}$

$y = x$

if $z.\text{chave} < x.\text{chave}$

$x = x.\text{esquerda}$

else $x = x.\text{direita}$

if $y = \text{NIL}$


$T.\text{raiz} = z$

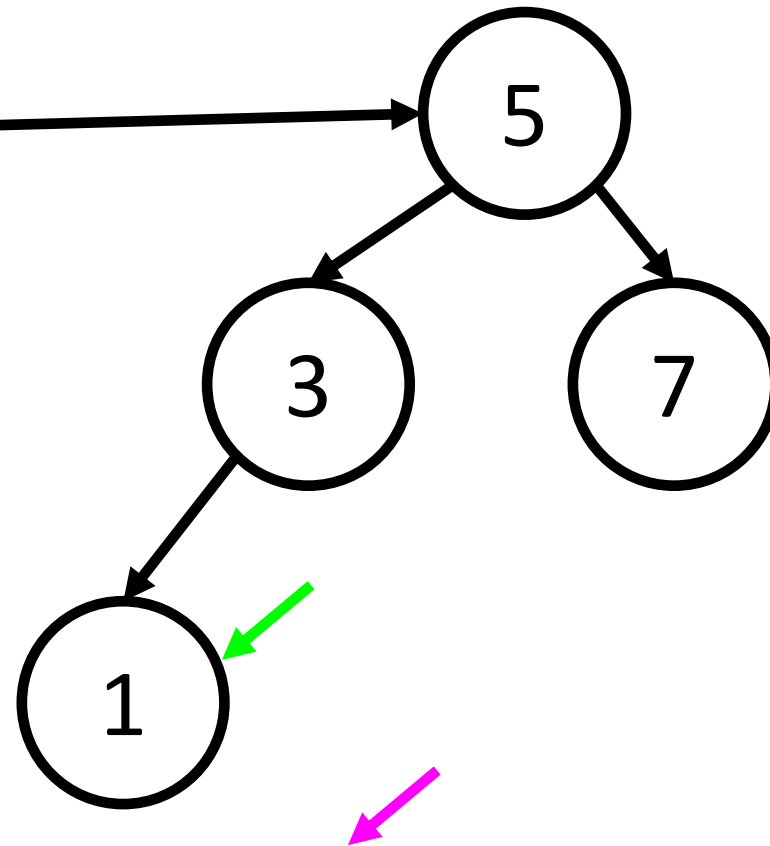
else if $z.\text{chave} < y.\text{chave}$

$y.\text{esquerda} = z$

else $y.\text{direita} = z$

x	1
y	3

BinaryTree	
raiz	



INSERT 5
 INSERT 3
 INSERT 7
 INSERT 1
 INSERT 2
 INSERT 6

TREE-INSERT(T, z)

$y = \text{NIL}$

$x = T.\text{raiz}$

while $x \neq \text{NIL}$

$y = x$

if $z.\text{chave} < x.\text{chave}$

$x = x.\text{esquerda}$

else $x = x.\text{direita}$

if $y = \text{NIL}$


$T.\text{raiz} = z$

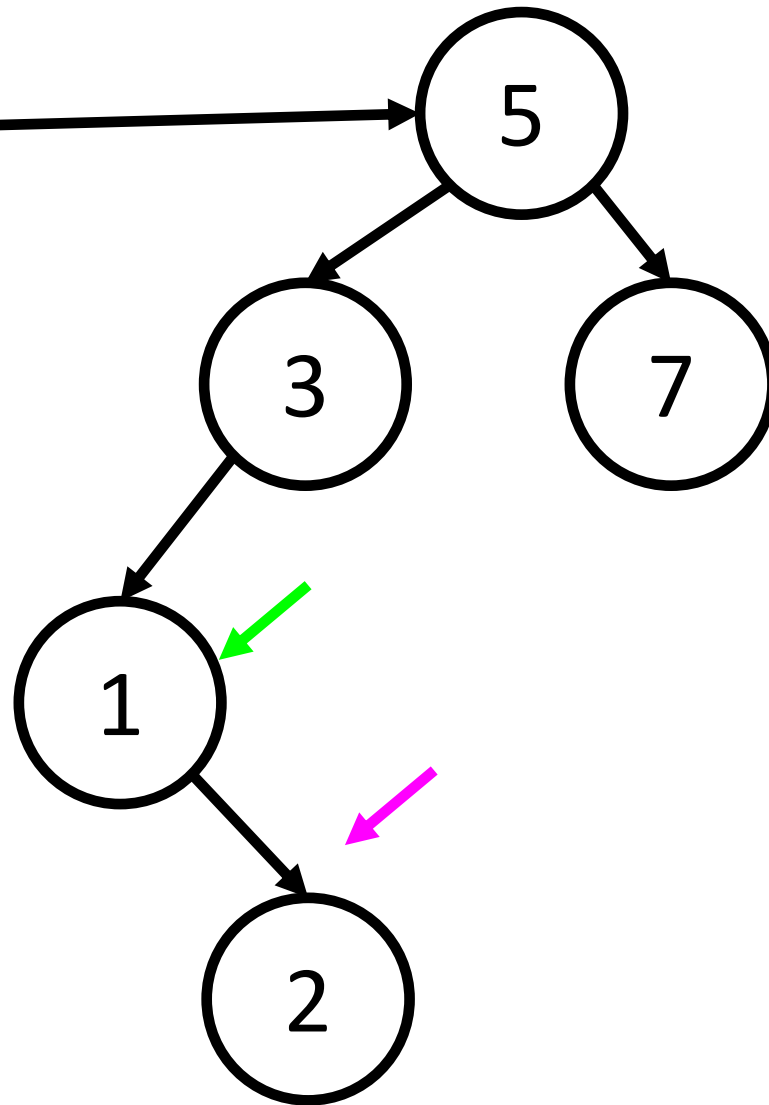
else if $z.\text{chave} < y.\text{chave}$

$y.\text{esquerda} = z$

else $y.\text{direita} = z$

x	NULL
y	1

BinaryTree	
raiz	



INSERT 5
 INSERT 3
 INSERT 7
 INSERT 1
 INSERT 2
 INSERT 6

TREE-INSERT(T, z)

$y = \text{NIL}$

$x = T.\text{raiz}$

while $x \neq \text{NIL}$

$y = x$

if $z.\text{chave} < x.\text{chave}$

$x = x.\text{esquerda}$

else $x = x.\text{direita}$

if $y = \text{NIL}$


$T.\text{raiz} = z$

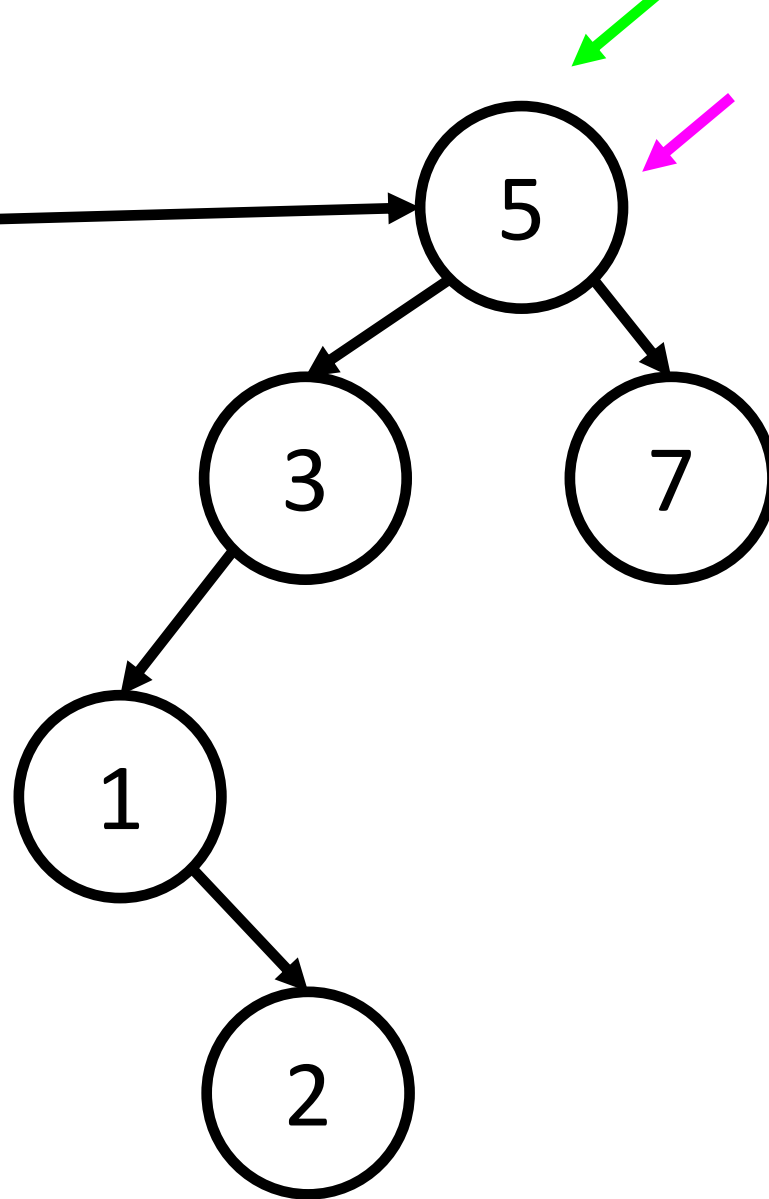
else if $z.\text{chave} < y.\text{chave}$

$y.\text{esquerda} = z$

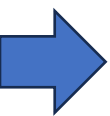
else $y.\text{direita} = z$

x	NULL
y	1

BinaryTree	
raiz	



INSERT 5
 INSERT 3
 INSERT 7
 INSERT 1
 INSERT 2
 INSERT 6



TREE-INSERT(T, z)

$y = \text{NIL}$

$x = T.\text{raiz}$

while $x \neq \text{NIL}$

$y = x$

if $z.\text{chave} < x.\text{chave}$

$x = x.\text{esquerda}$

else $x = x.\text{direita}$

if $y = \text{NIL}$

$T.\text{raiz} = z$

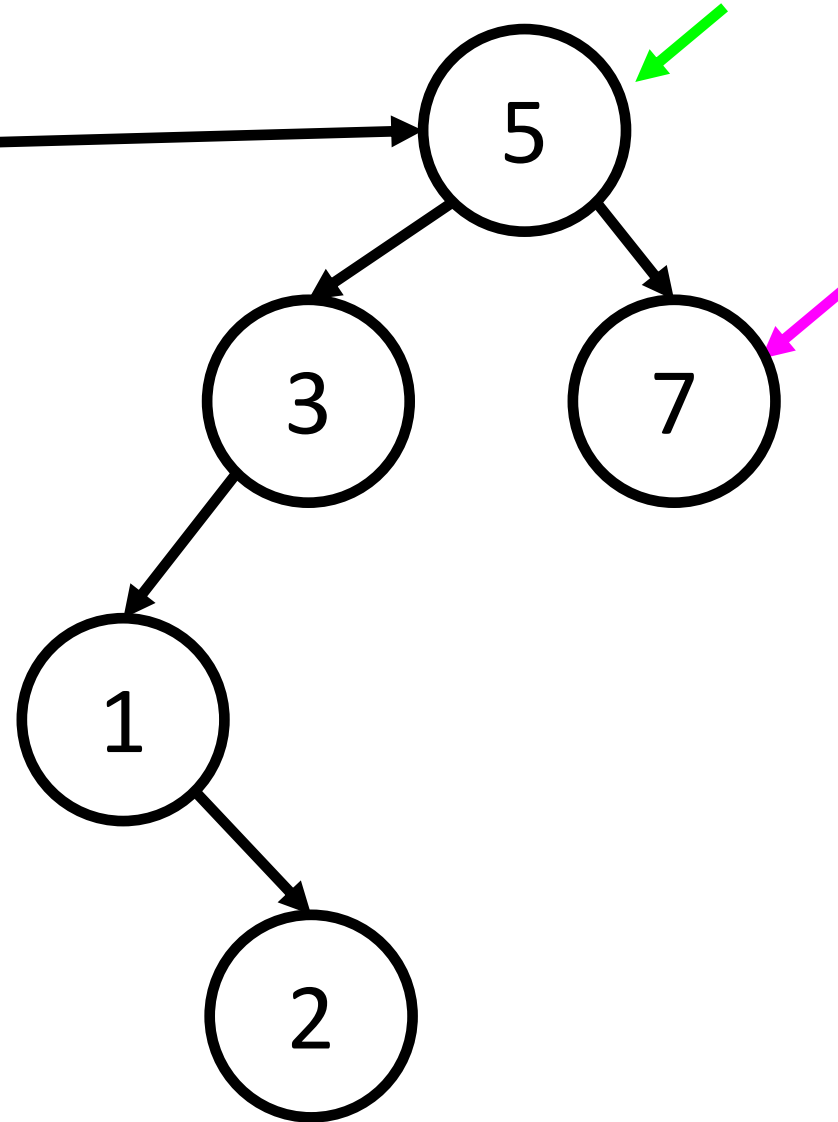
else if $z.\text{chave} < y.\text{chave}$

$y.\text{esquerda} = z$

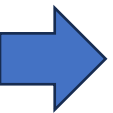
else $y.\text{direita} = z$

x	5
y	NULL

BinaryTree	
raiz	



INSERT 5
 INSERT 3
 INSERT 7
 INSERT 1
 INSERT 2
 INSERT 6



TREE-INSERT(T, z)

$y = \text{NIL}$

$x = T.\text{raiz}$

while $x \neq \text{NIL}$

$y = x$

if $z.\text{chave} < x.\text{chave}$

$x = x.\text{esquerda}$

else $x = x.\text{direita}$

if $y = \text{NIL}$


$T.\text{raiz} = z$

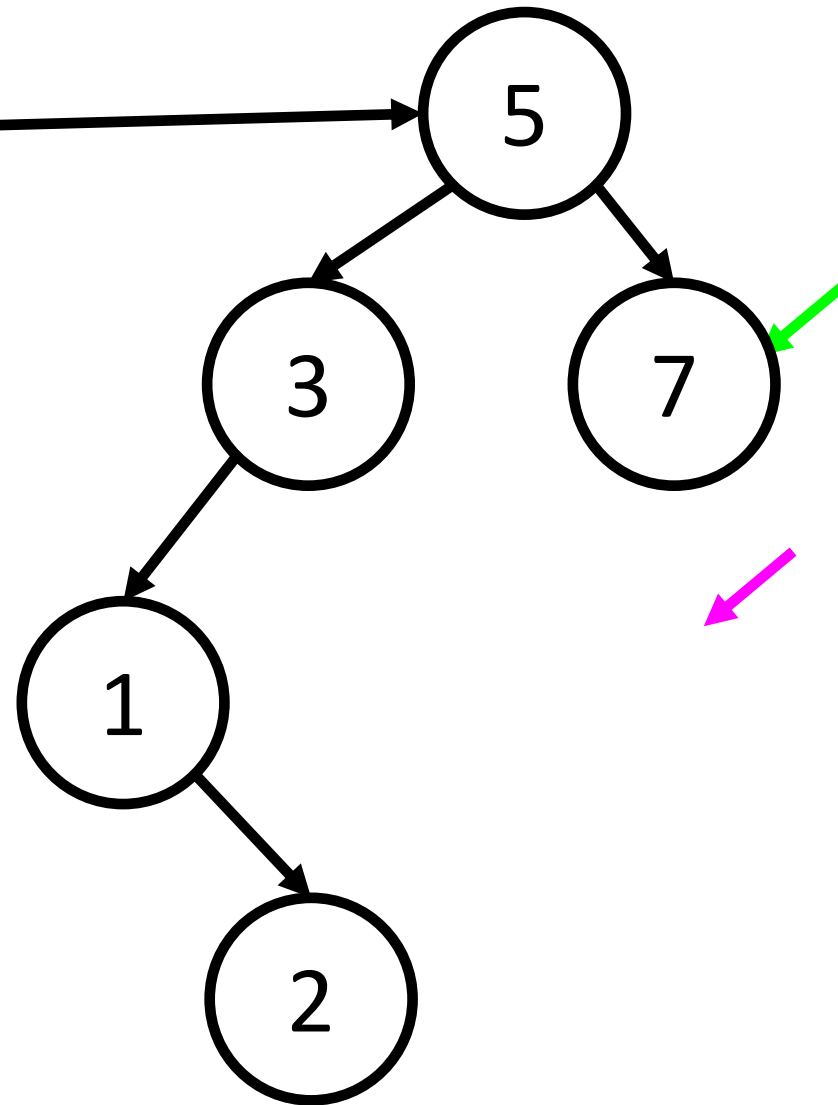
else if $z.\text{chave} < y.\text{chave}$

$y.\text{esquerda} = z$

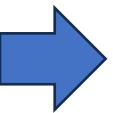
else $y.\text{direita} = z$

x	7
y	5

BinaryTree	
raiz	



INSERT 5
 INSERT 3
 INSERT 7
 INSERT 1
 INSERT 2
 INSERT 6



TREE-INSERT(T, z)

$y = \text{NIL}$

$x = T.\text{raiz}$

while $x \neq \text{NIL}$

$y = x$

if $z.\text{chave} < x.\text{chave}$

$x = x.\text{esquerda}$

else $x = x.\text{direita}$

if $y = \text{NIL}$


$T.\text{raiz} = z$

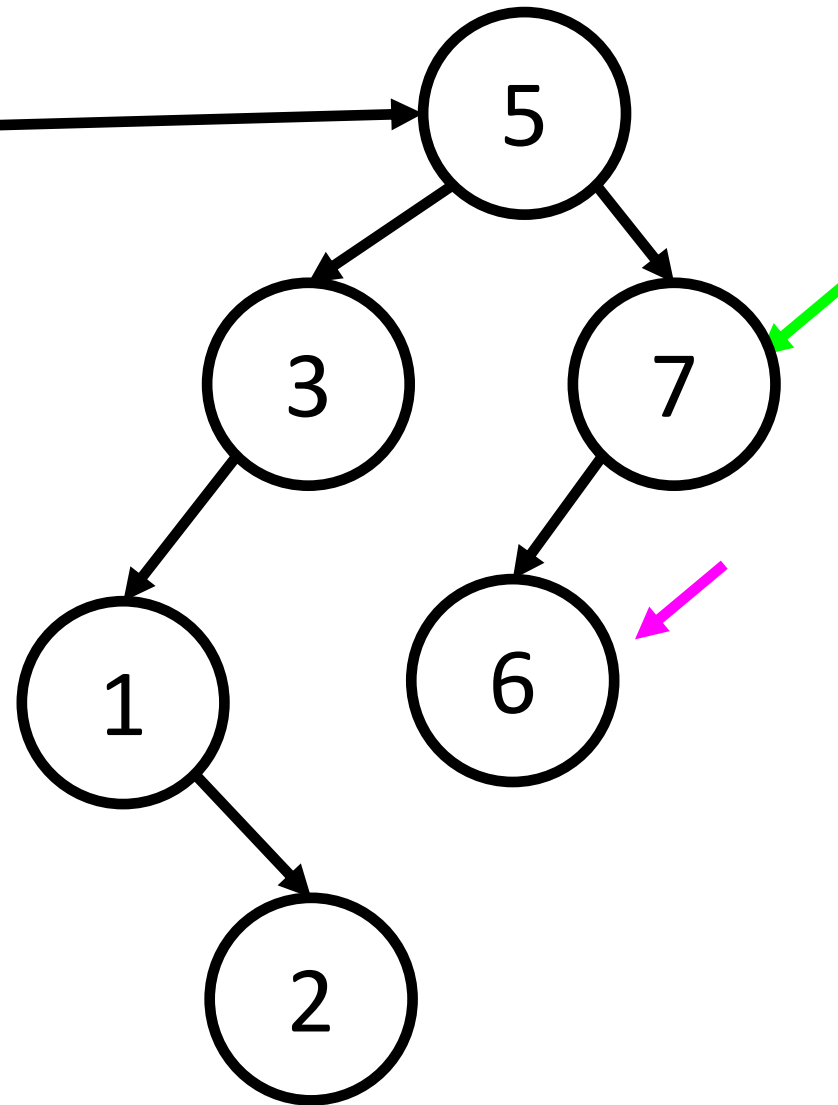
else if $z.\text{chave} < y.\text{chave}$

$y.\text{esquerda} = z$

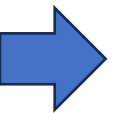
else $y.\text{direita} = z$

x	NULL
y	7

BinaryTree	
raiz	



INSERT 5
 INSERT 3
 INSERT 7
 INSERT 1
 INSERT 2
 INSERT 6



TREE-INSERT(T, z)

$y = \text{NIL}$

$x = T.\text{raiz}$

while $x \neq \text{NIL}$

$y = x$

if $z.\text{chave} < x.\text{chave}$

$x = x.\text{esquerda}$

else $x = x.\text{direita}$

if $y = \text{NIL}$

$T.\text{raiz} = z$

else if $z.\text{chave} < y.\text{chave}$

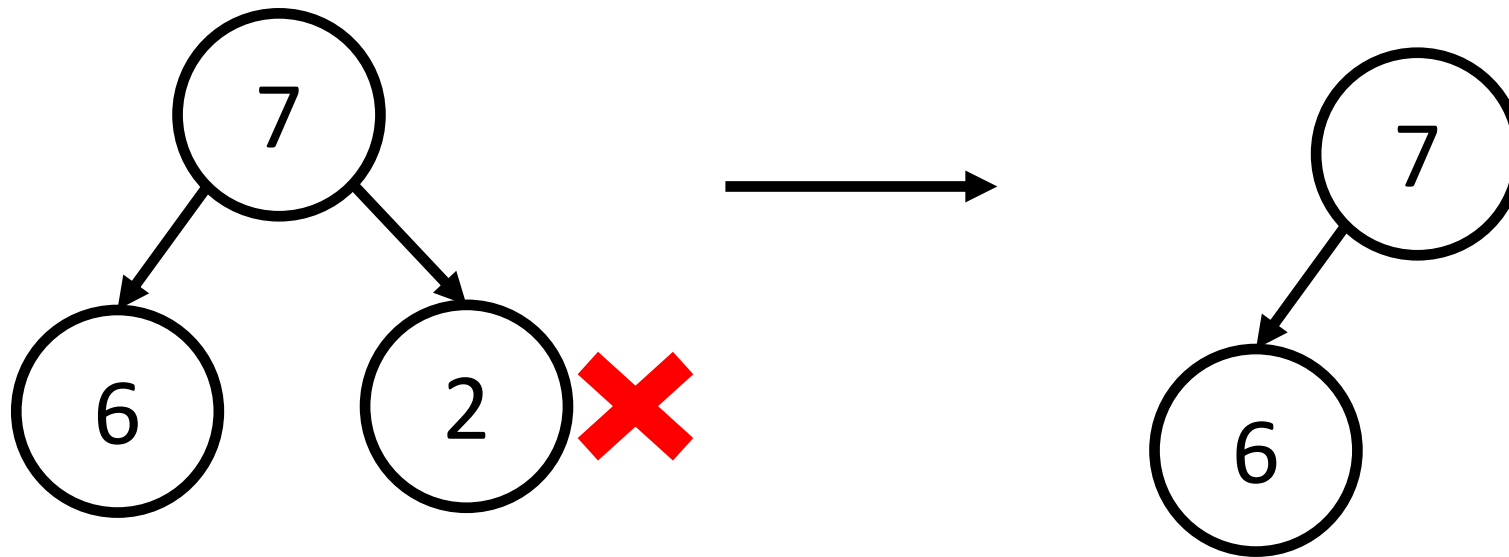
$y.\text{esquerda} = z$

else $y.\text{direita} = z$

x	NULL
y	7

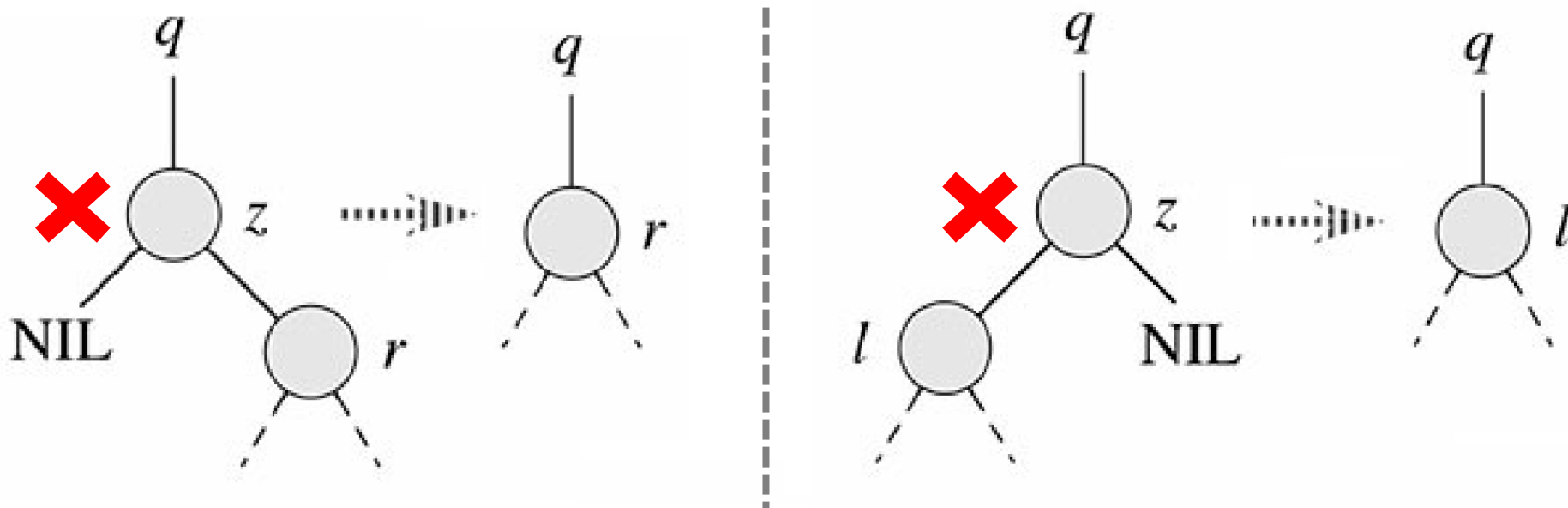
Remoção

Caso 1: Se z não tem nenhum filho, então simplesmente o removemos modificando seu pai de modo a substituir z por NULL como seu filho.



Remoção

Caso 2: Se o nó tem apenas um filho, então elevamos esse filho para que ocupe a posição de z na árvore modificando o pai de z de modo a substituir z pelo filho de z .

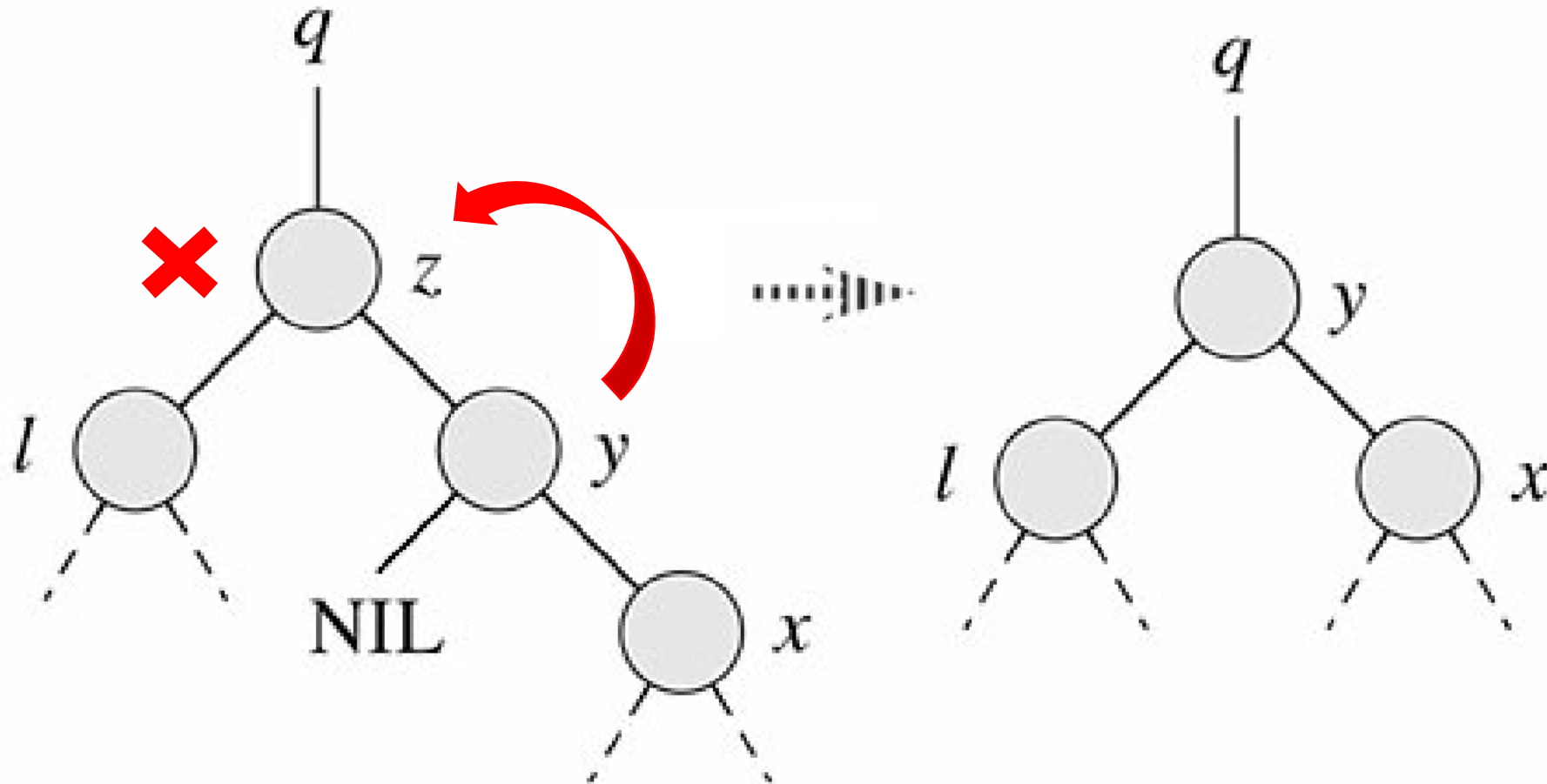


Remoção

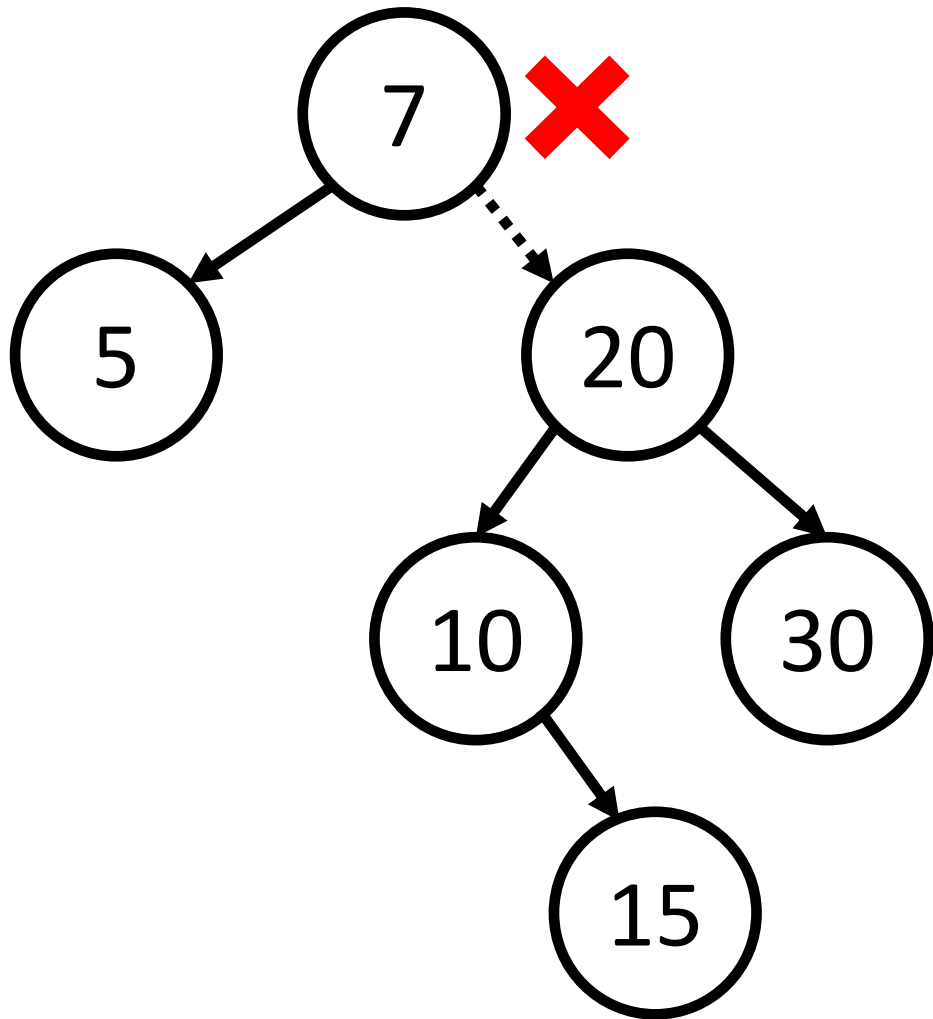
Caso 3: Se z tiver dois filhos, encontramos o sucessor de z , y , que deve estar na subárvore direita de z , e obrigamos y a tomar a posição de z na árvore.

- Esse é um caso complicado porque, como veremos, o fato de y ser ou não o filho à direita de z é importante.

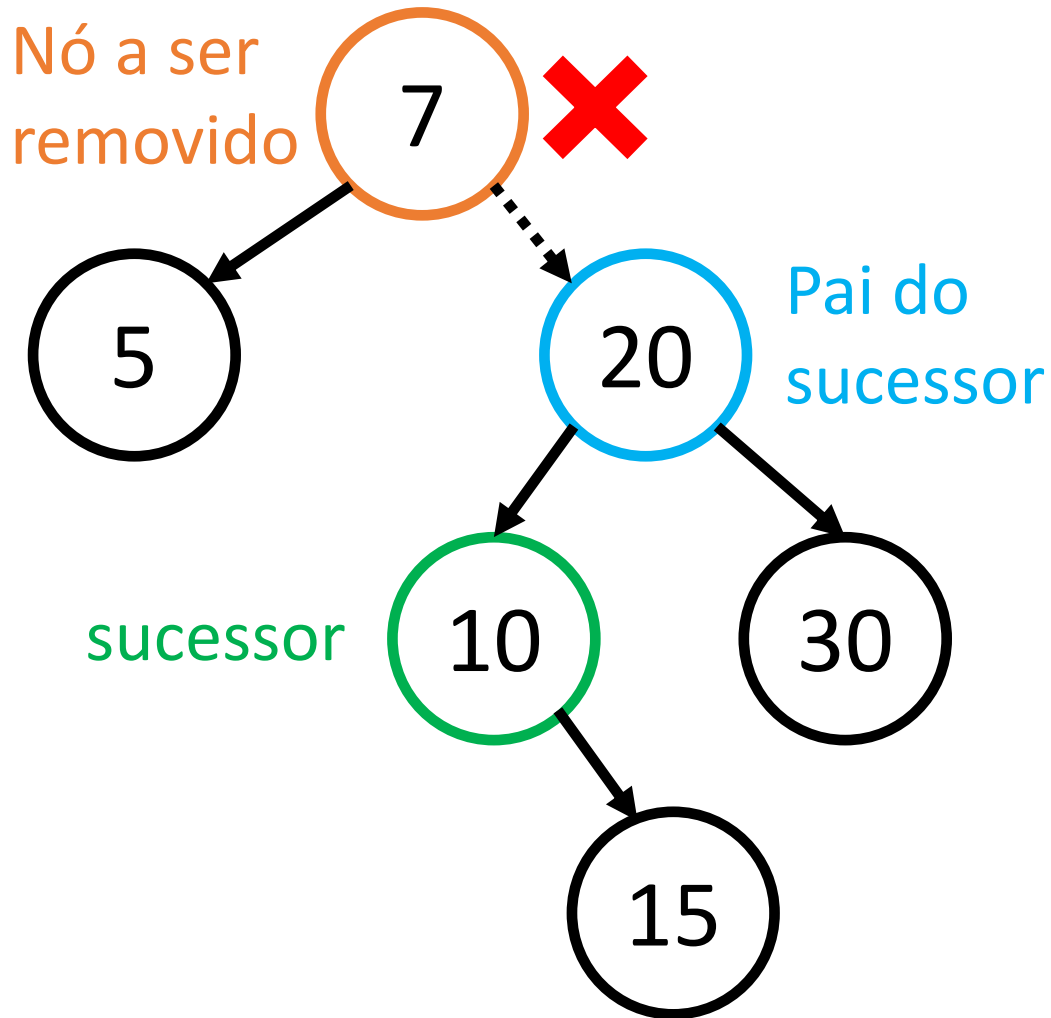
Caso 3.1: Se y é o filho à direita de z , substituímos z por y .



Caso 3.2: Se y encontra-se na subárvore direita de z , mas não é o filho à direita de z , primeiro substituímos y por seu próprio filho à direita e depois substituímos z por y .



Caso 3.2: Se y encontra-se na subárvore direita de z, mas não é o filho à direita de z, primeiro substituímos y por seu próprio filho à direita e depois substituímos z por y.

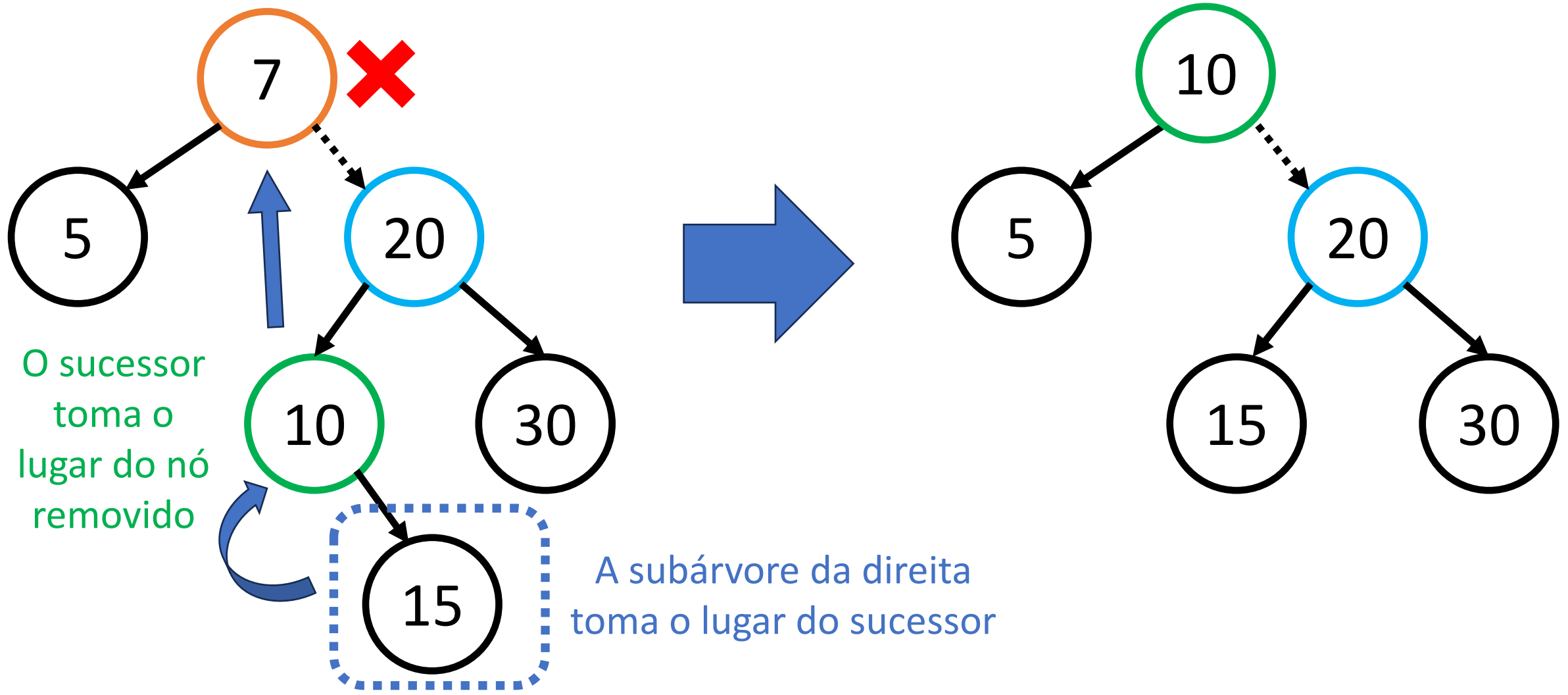


Três nós estão envolvidos no processo de remoção deste caso:

- o nó a ser removido,
- o sucessor e
- o pai do sucessor.

O sucessor é o menor elemento (o mais à esquerda) da subárvore da direita do nó a ser removido. Ele não possui outro nó à esquerda.

Caso 3.2: Se y encontra-se na subárvore direita de z , mas não é o filho à direita de z , primeiro substituímos y por seu próprio filho à direita e depois substituímos z por y .



Implementação da Remoção

- Para simplificar a implementação da remoção, vamos assumir que os nós possuem um atributo **p** que é um **ponteiro para o nó pai**.
- Vamos começar implementando uma função TRANSPLANT que movimenta subárvores.
- A função de remoção será definida usando a função TRANSPLANT.

TRANSPLANT (T, u, v)

```
1  if  $u.p == \text{NIL}$ 
2       $T.raiz = v$ 
3  elseif  $u == u.p.esquerda$ 
4       $u.p.esquerda = v$ 
5  else  $u.p.direita = v$ 
6  if  $v \neq \text{NIL}$ 
7       $v.p = u.p$ 
```

Se o pai de u é nulo, então u é a raiz. Neste caso, apenas faça a raiz passar a ser v .

Se u é o filho da esquerda de seu pai, faça v passar a ser o filho da esquerda.

Caso contrário, se u é o filho da direita, faça v se tornar o filho da direita.

Após a troca, atualize o pai de v

A função rotina TRANSPLANT substitui a subárvore enraizada no nó u pela subárvore enraizada no nó v fazendo com que o pai de u se torne pai de v .

TREE-DELETE (T, z)

```
1  if  $z.esquerda == \text{NIL}$ 
2      TRANSPLANT ( $T, z, z.direita$ )
3  elseif  $z.direita == \text{NIL}$ 
4      TRANSPLANT ( $T, z, z.esquerda$ )
5  else  $y = \text{TREE-MINIMUM}(z.direita)$ 
6      if  $y.p \neq z$ 
7          TRANSPLANT ( $T, y, y.direita$ )
8           $y.direita = z.direita$ 
9           $y.direita.p = y$ 
10     TRANSPLANT ( $T, z, y$ )
11      $y.esquerda = z.esquerda$ 
12      $y.esquerda.p = y$ 
```

Se z não tiver nenhum filho à esquerda, substituímos z por seu filho à direita, que pode ser ou não NULL. Quando ele é NULL, esse caso trata a situação na qual z não tem nenhum filho.

Se z tiver apenas um filho, que é o seu filho à esquerda, substituímos z por ele.

← Encontra o sucessor de z

Casos nos quais z tem dois filhos. Vamos recortar y para ele substituir z .

- Se y é o filho à direita de z , então as linhas 10–12 substituem z como um filho de seu pai por y e substitui o filho à esquerda de y pelo filho à esquerda de z .
- Se y não é o filho à direita de z , as linhas 7–9 substituem y como um filho de seu pai pelo filho à direita de y e transforma o filho à direita de z em filho à direita de y , e então as linhas 10–12 substituem z como um filho de seu pai por y e substituem o filho à esquerda de y pelo filho à esquerda de z .

