Estruturas de Dados

Árvores Red-Black

2022/2023

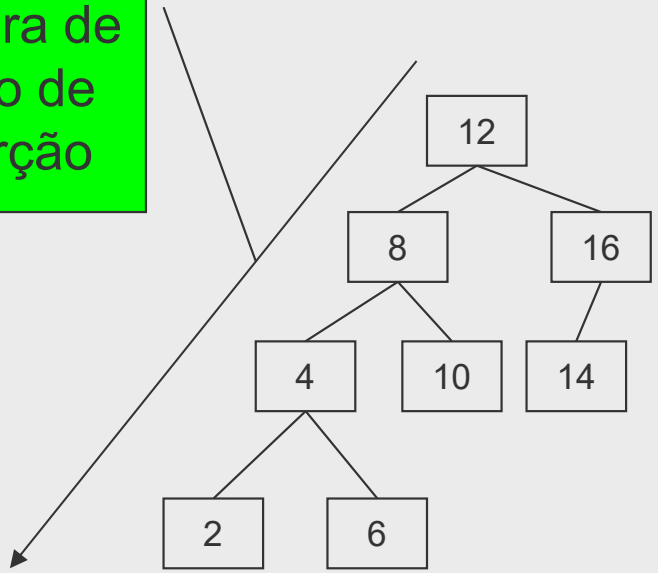
Árvores Red-Black

- A árvore AVL tem uma condição de equilibrio “exigente”:
  - De forma a cumprir com essa condição, é frequentemente necessário reequilibrar a árvore após uma inserção.

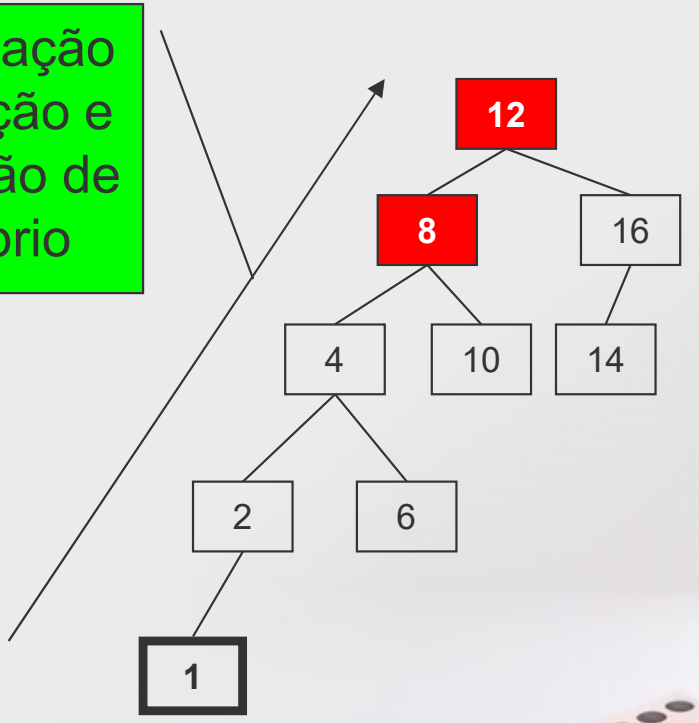
Árvores Red-Black

- A árvore AVL tem uma condição de equilibrio “exigente”:
  - De forma a cumprir com essa condição, é frequentemente necessário reequilibrar a árvore após uma inserção.
  - **insere(1)**

Procura de ponto de inserção



Actualização verificação e reparação de equilibrio



Árvores Red-Black

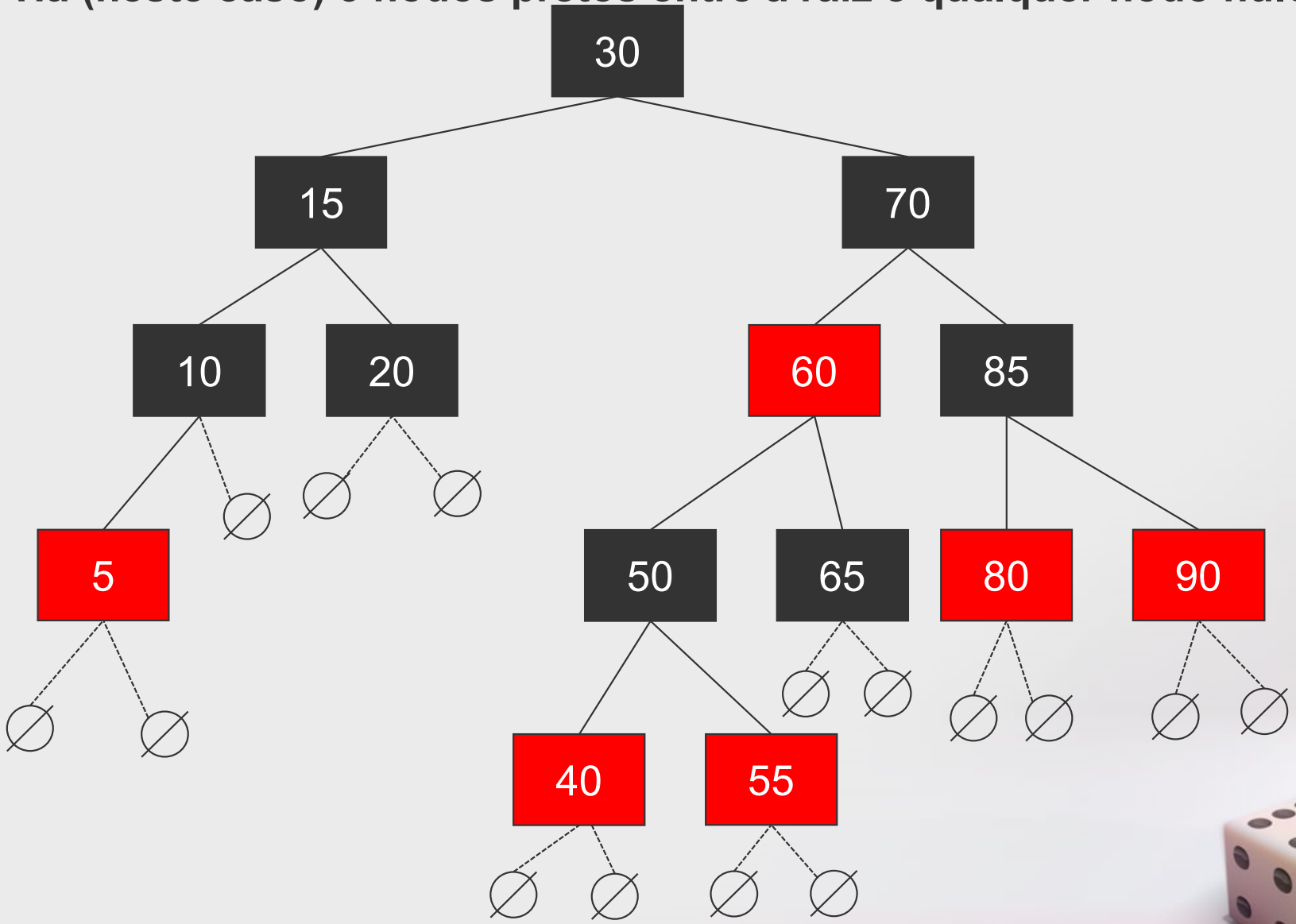
- Adoptando um critério de equilibrio mais relaxado, pode-se reduzir o número de operações de balanceamento necessárias
  - **Desta forma, existe um compromisso entre profundidade e eficiência de inserção.**
- Em comparação com as árvores AVL, as árvoes **red-black** têm uma maior eficiência de inserção, por via da utilização de uma propriedade de equilibrio ligeiramente mais permissiva.

Árvores Red-Black

- **Propriedade de Equilibrio (árvore red-black):**
  1. Todos os nodos são coloridos (vermelhos/pretos)
  2. A raiz é preta.
  3. Se um nodo é vermelho, ambos os descendentes devem ser pretos.
  4. Todos os percursos entre um raiz e qualquer um nodo nulo devem passar pelo mesmo número de nodos pretos

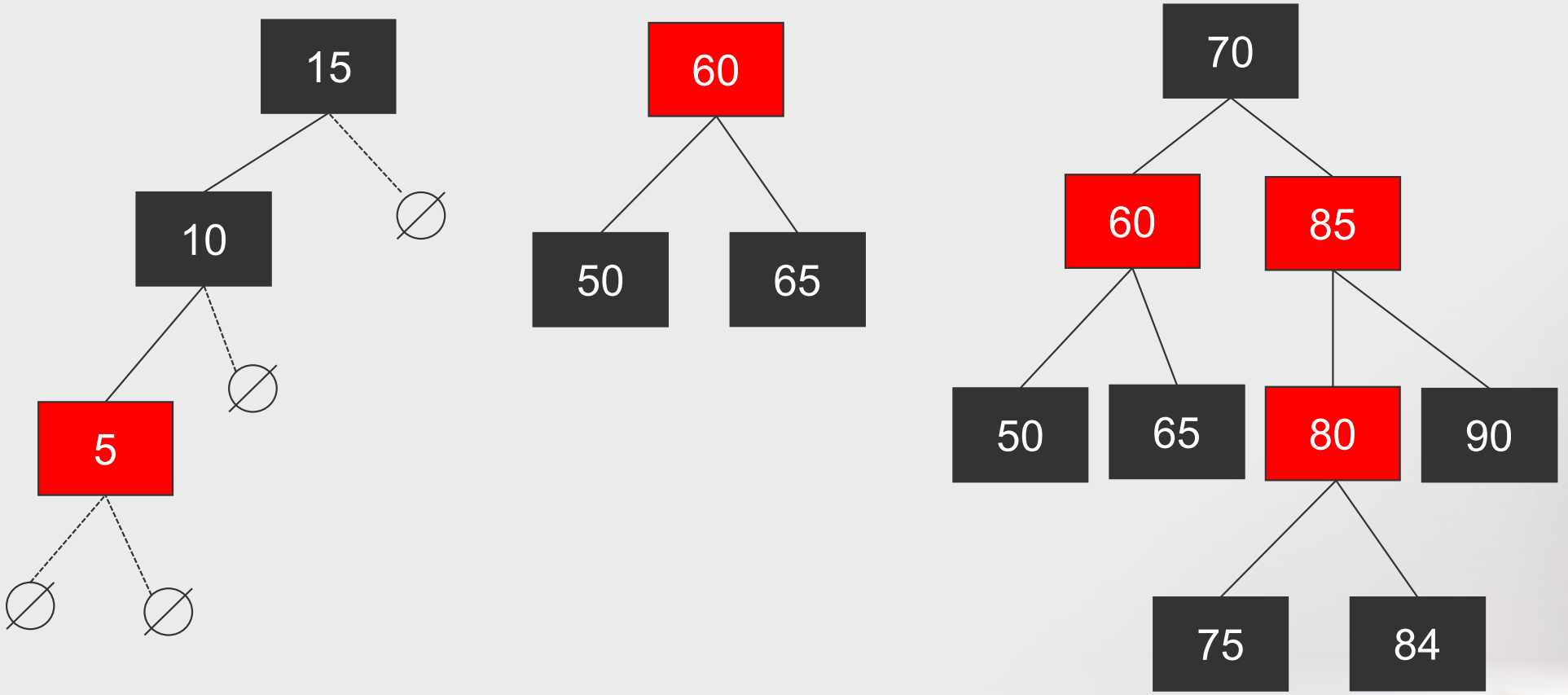
Árvores Red-Black

p.ex: Há (neste caso) 3 nodos pretos entre a raiz e qualquer nodo nulo



Árvores Red-Black

Estas NÃO são árvores RB

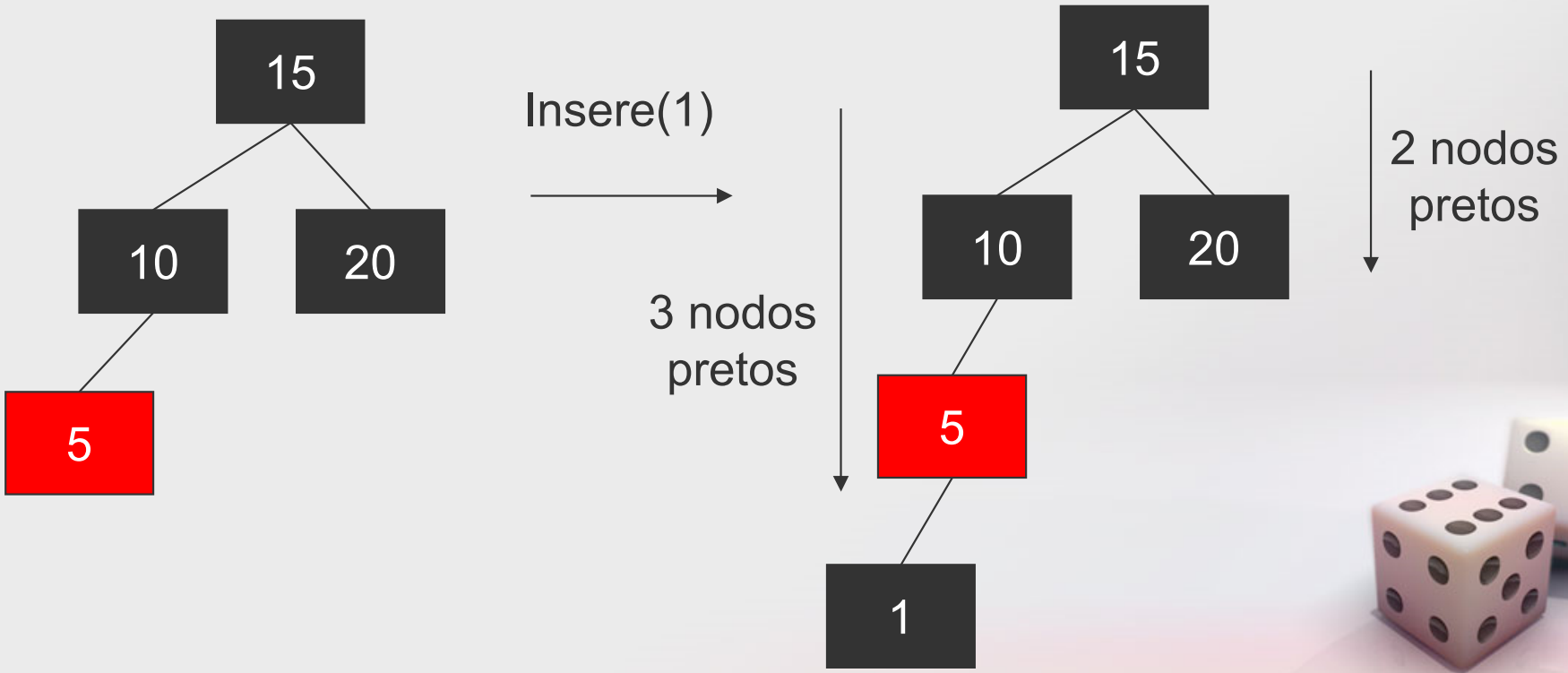


Árvores Red-Black

- Consegue-se demonstrar que, usando esta propriedade de equilibrio:
  - A àrvore tem que conter pelo menos 2^B-1 nodos pretos, onde B é o número de nodos pretos entre a raiz e uma folha.
  - A altura da àrvore é, no máximo, 2 log (N+1)
    - A pesquisa é logaritmica.

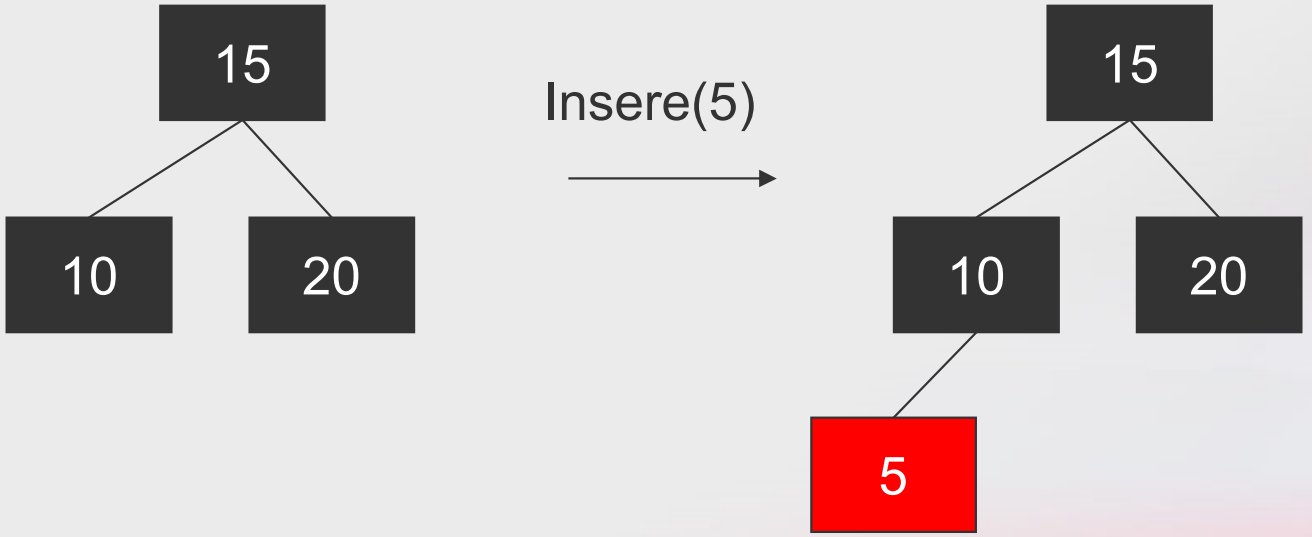
Árvores Red-Black

- Inserção de um nodo:
  - Um nodo é inserido na árvore como uma nova folha, logo, se fosse inserido como sendo preto, o percurso entre a raiz e os descendentes nulos desse nodo atravessaria mais um nodo preto do que os restantes!

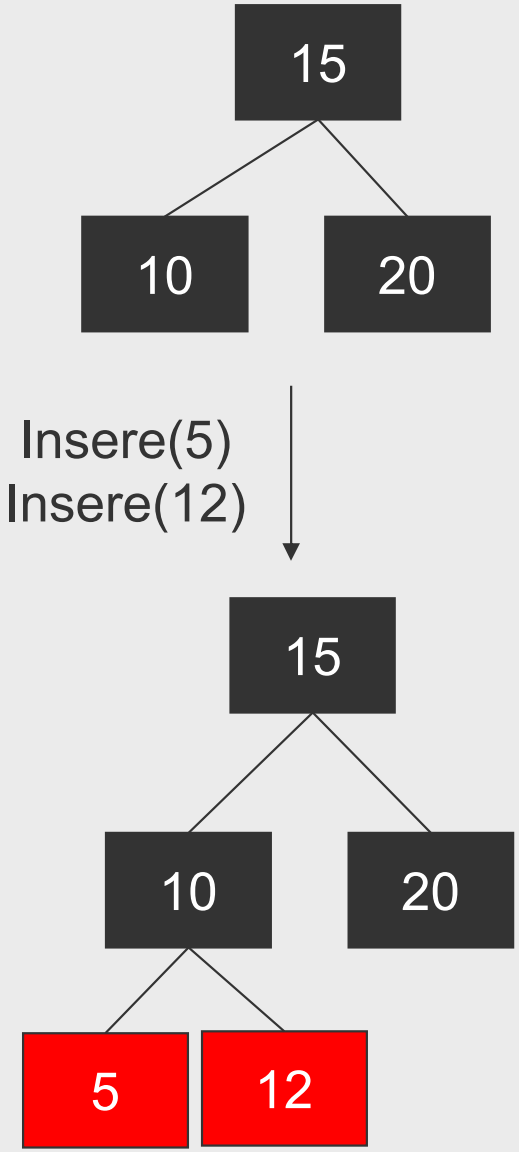


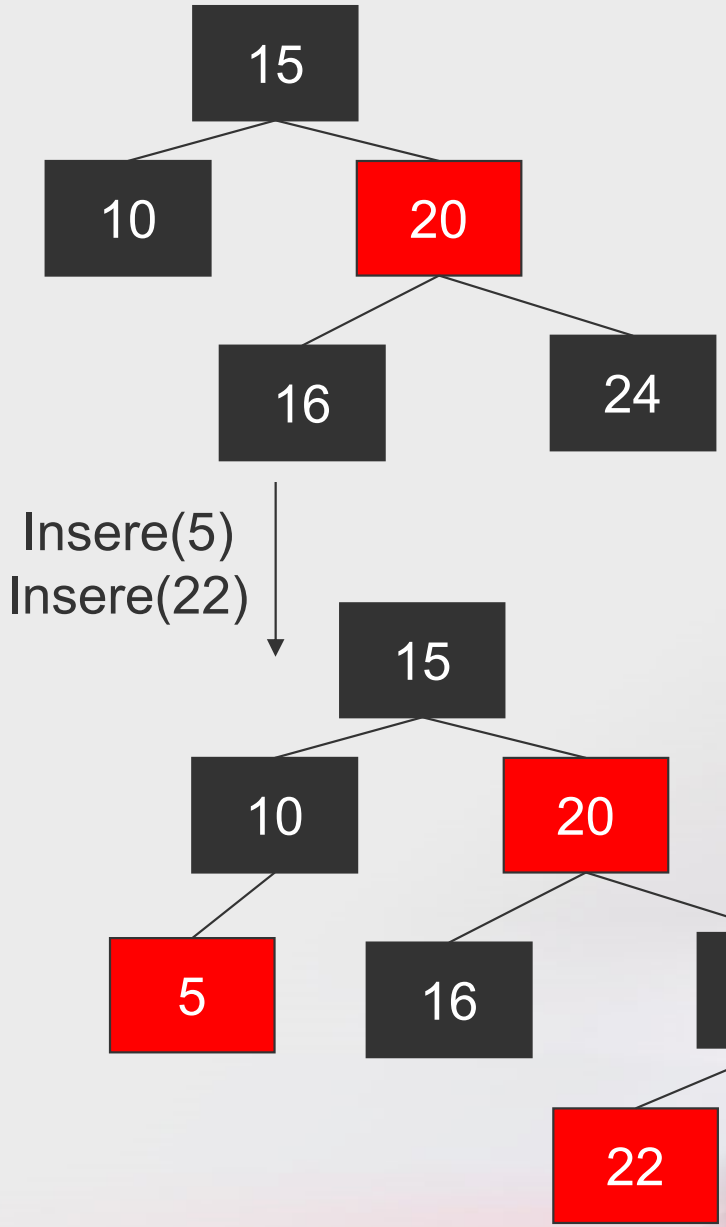
Árvores Red-Black

- Inserção de um nodo:
  - Os novos nodos são inseridos como sendo nodos vermelhos!
  - Caso o antecessor seja um nodo preto?
    - Todas as propriedades de equilíbrio estão verificadas!
    - A inserção termina sem mais verificações ou procedimentos adicionais.



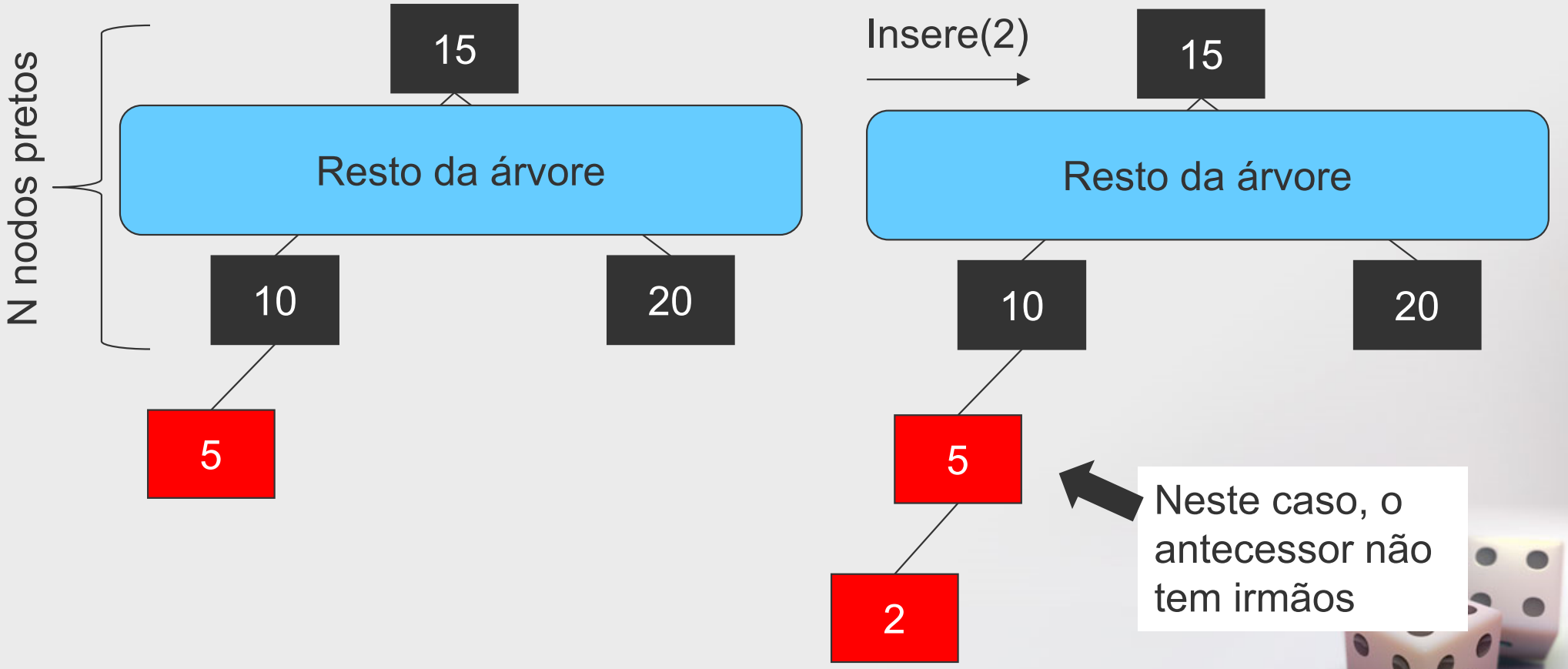
Árvores Red-Black





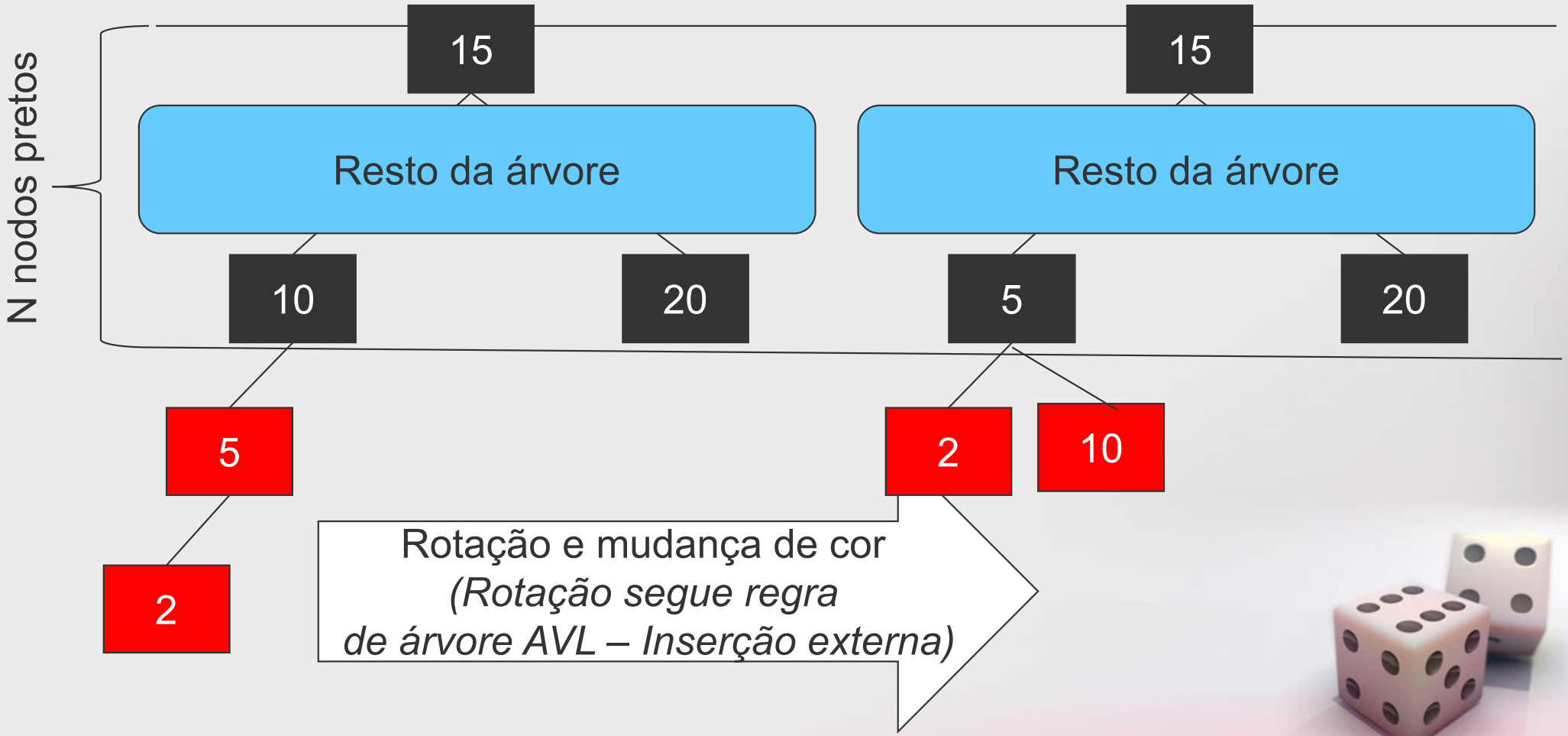
Árvores Red-Black

- Inserção de um nodo sob um nodo vermelho:
  - Há vários casos a considerar.
  - Ter em atenção manutenção de propriedade de equilibrio



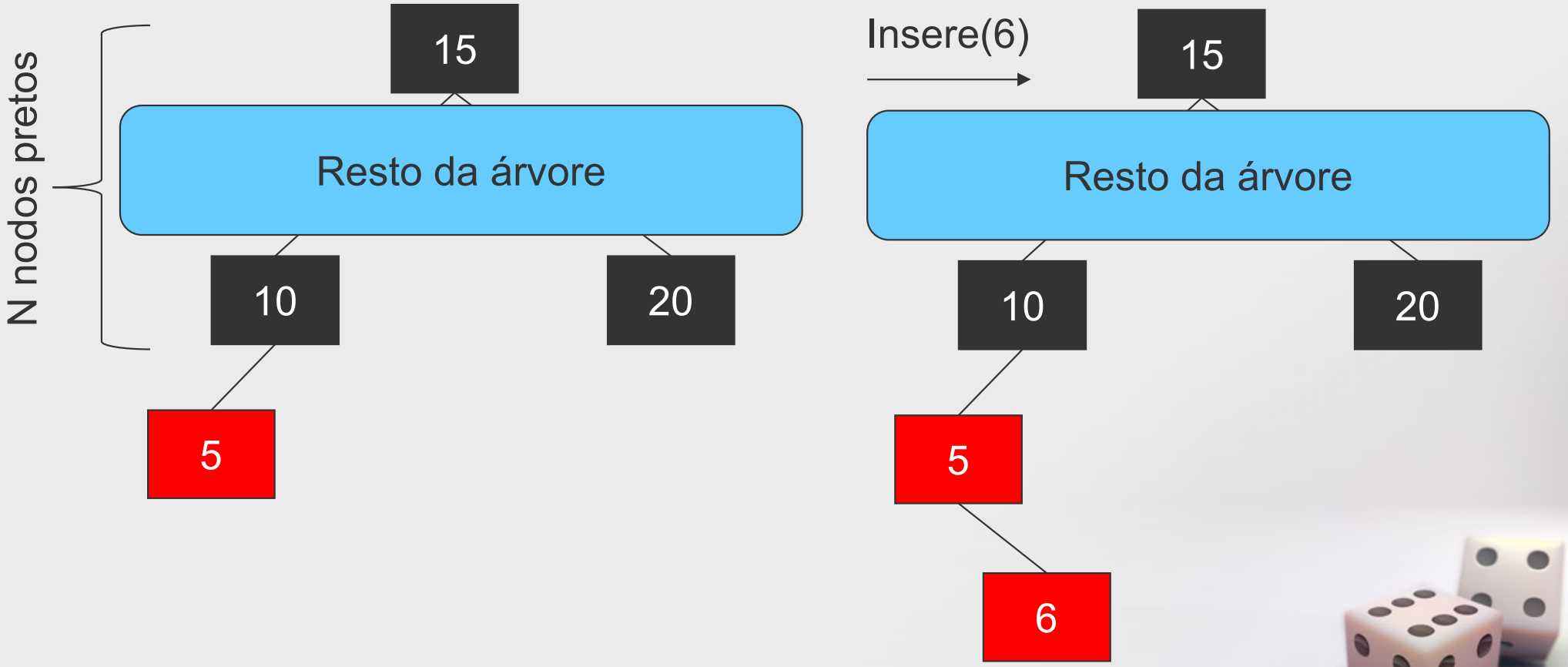
Árvores Red-Black

- Inserção de um nodo sob um nodo vermelho:
  - Há vários casos a considerar.
  - Ter em atenção manutenção de propriedade de equilibrio



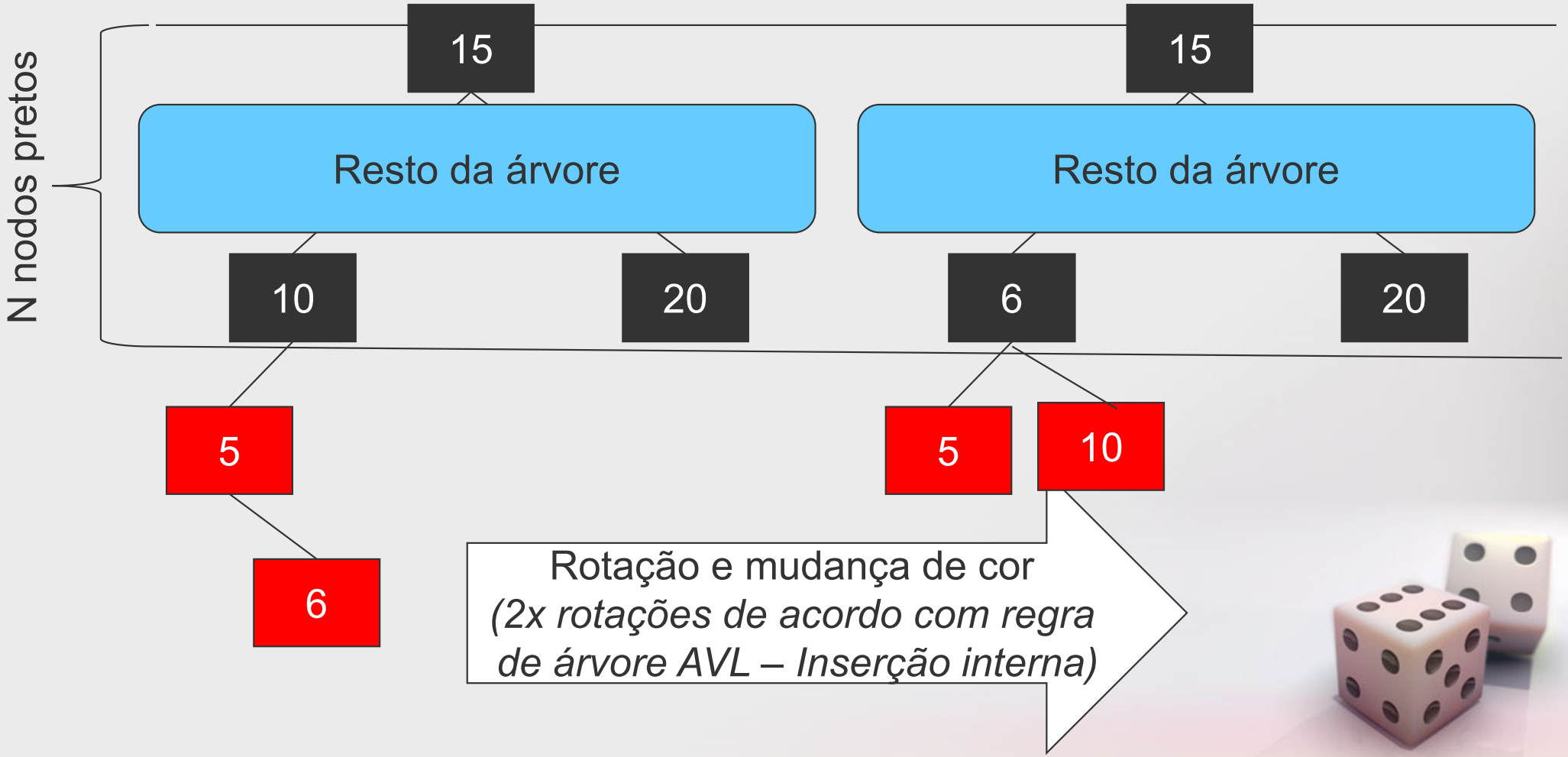
Árvores Red-Black

- Inserção de um nodo sob um nodo vermelho:
  - Há vários casos a considerar.
  - Ter em atenção manutenção de propriedade de equilibrio



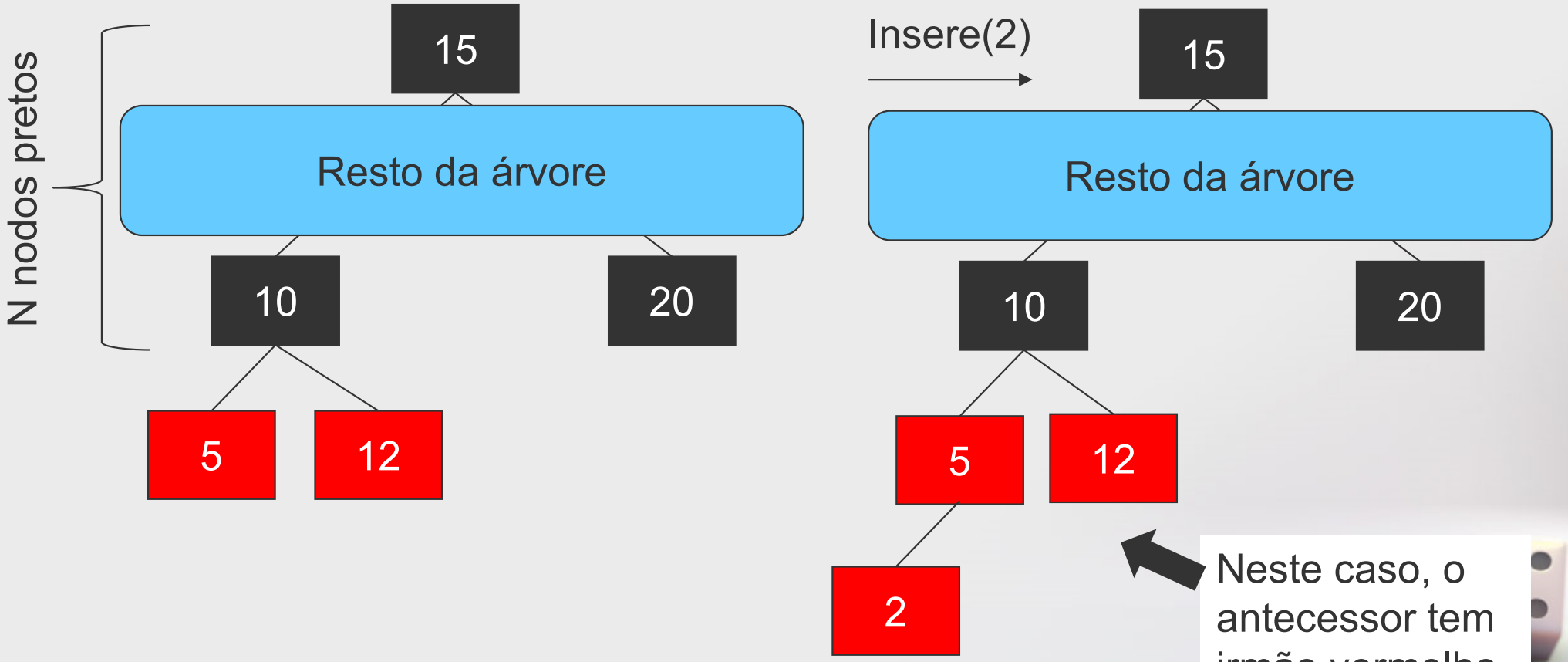
Árvores Red-Black

- Inserção de um nodo sob um nodo vermelho:
  - Há vários casos a considerar.
  - Ter em atenção manutenção de propriedade de equilibrio



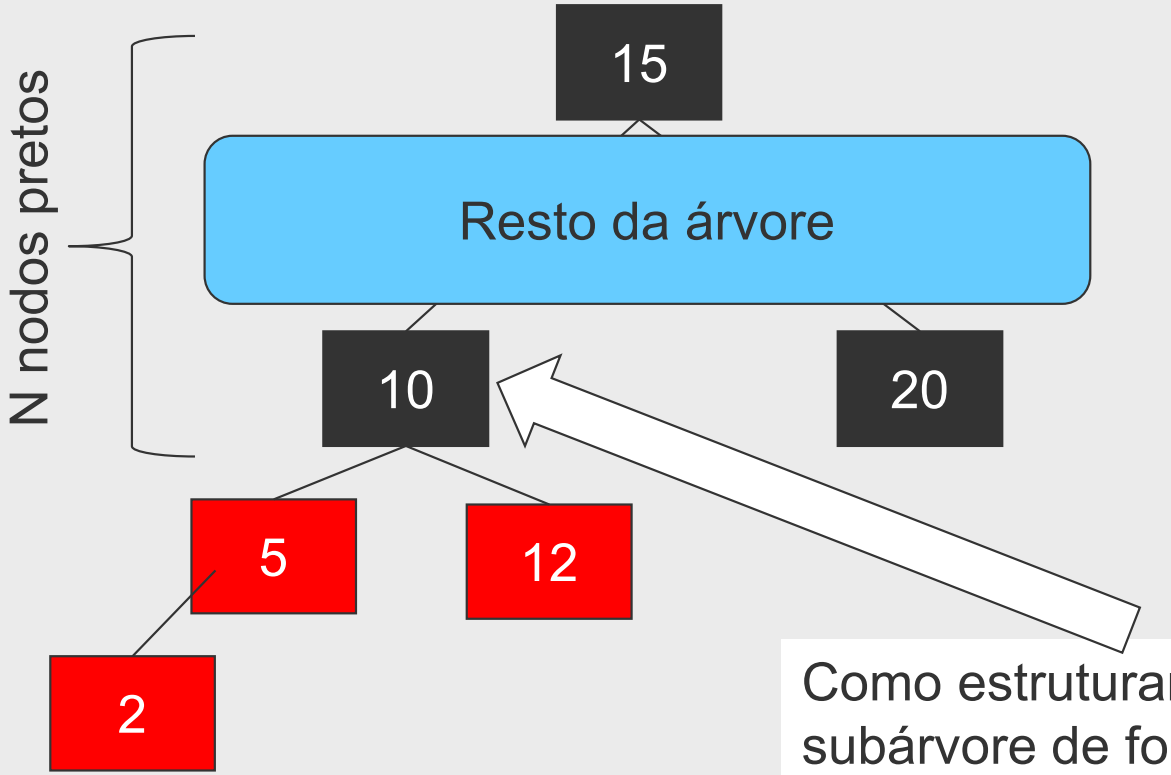
Árvores Red-Black

- Inserção de um nodo sob um nodo vermelho:
  - Há vários casos a considerar.
  - Ter em atenção manutenção de propriedade de equilibrio



Árvores Red-Black

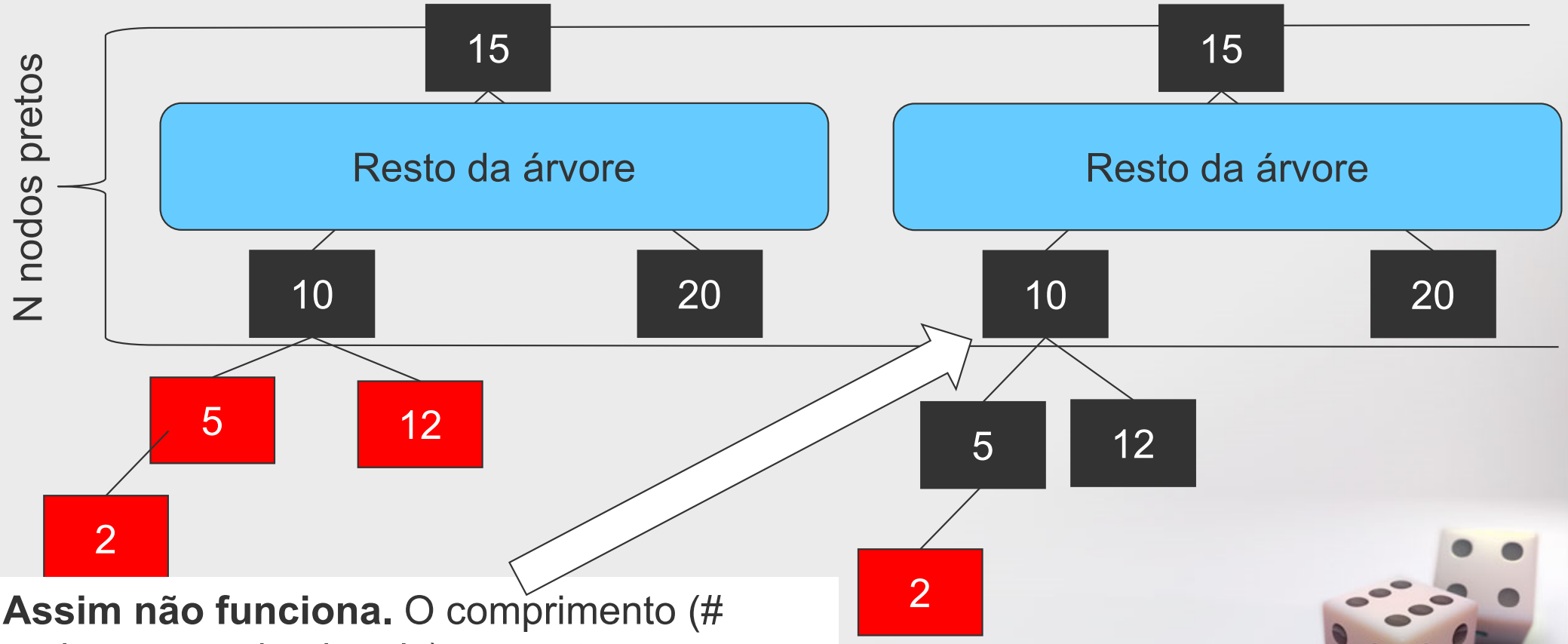
- Inserção de um nodo sob um nodo vermelho:
  - Há vários casos a considerar.
  - Ter em atenção manutenção de propriedade de equilibrio



Como estruturar (rodar, recolorir) esta subárvore de forma a manter o equilibrio de toda a árvore?

Árvores Red-Black

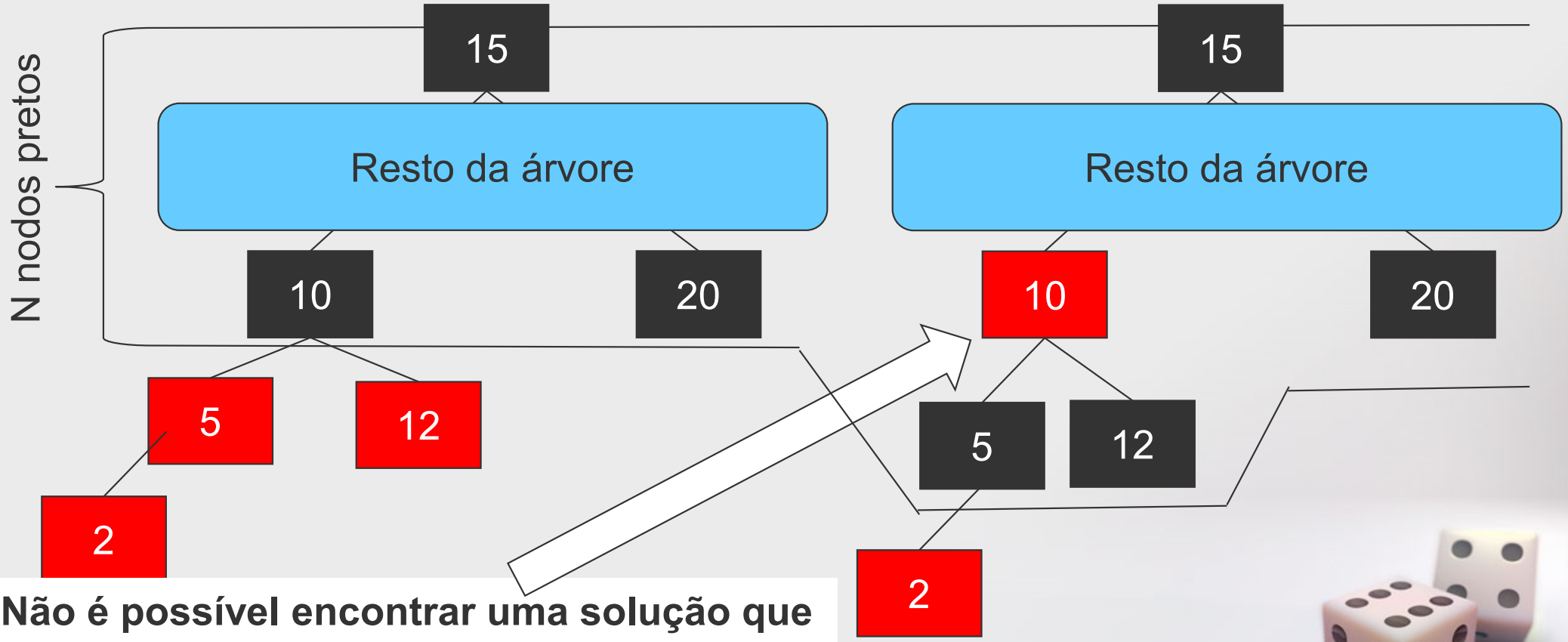
- Inserção de um nodo sob um nodo vermelho:
  - Há vários casos a considerar.
  - Ter em atenção manutenção de propriedade de equilibrio



Assim não funciona. O comprimento (# nodos pretos desde raiz) aumenta apenas nesta parte de árvore, o que a torna desequilibrada

Árvores Red-Black

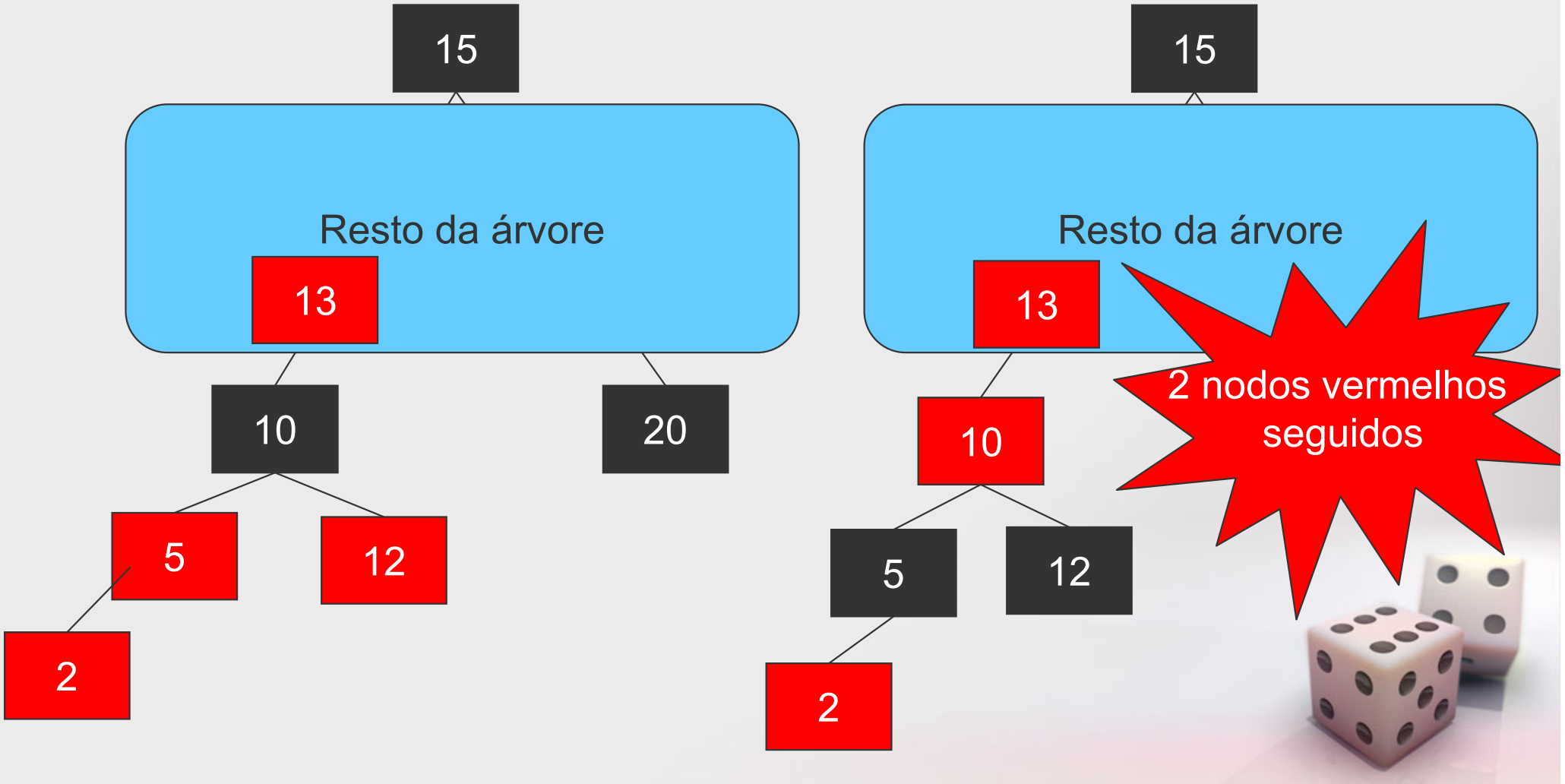
- Inserção de um nodo sob um nodo vermelho:
  - Há vários casos a considerar.
  - Ter em atenção manutenção de propriedade de equilibrio



Não é possível encontrar uma solução que não tenha a raiz desta sub-árvore vermelha. Por exemplo, assim.

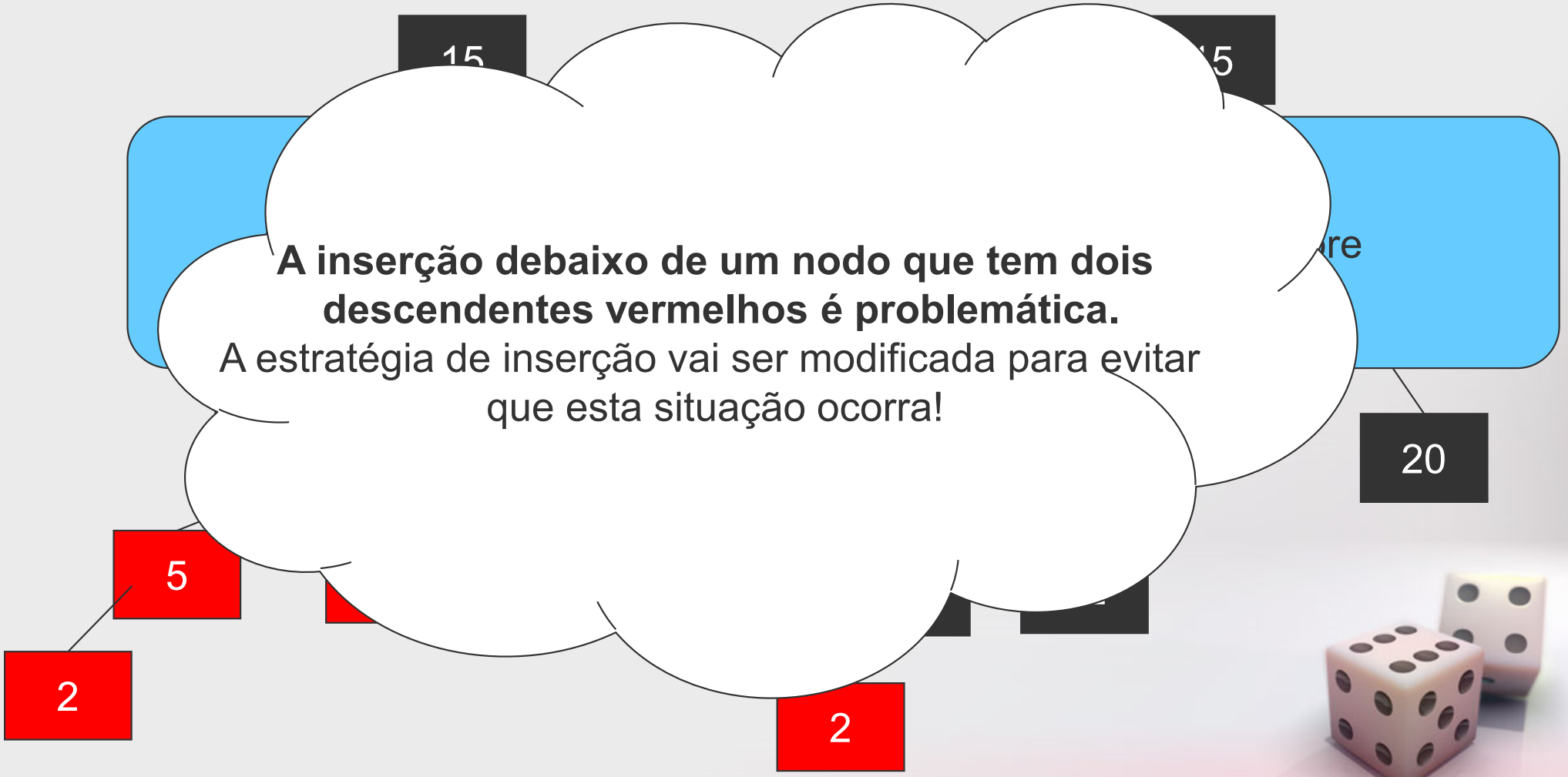
Árvores Red-Black

- Isto é um problema, se o nodo antecessor tb for vermelho!



Árvores Red-Black

- Isto é um problema, se o nodo antecessor tb for vermelho!



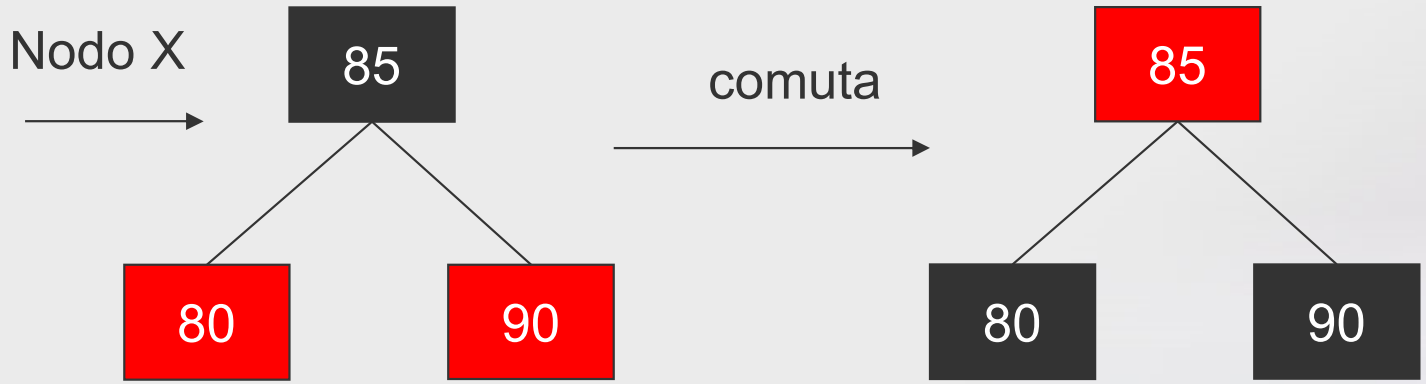
A inserção debaixo de um nodo que tem dois descendentes vermelhos é problemática.
A estratégia de inserção vai ser modificada para evitar que esta situação ocorra!

Árvores Red-Black

- Inserção (*Top-Down*):
  - A ideia é aproveitar a travessia descendente á procura do local de inserção para efectuar modificações na árvore que garantem uma inserção com sucesso.
    - Especificamente, queremos garantir que nunca inserimos debaixo de um nodo que tem um irmão vermelho.
    - Desta forma, nunca se irá verificar o caso problemático!

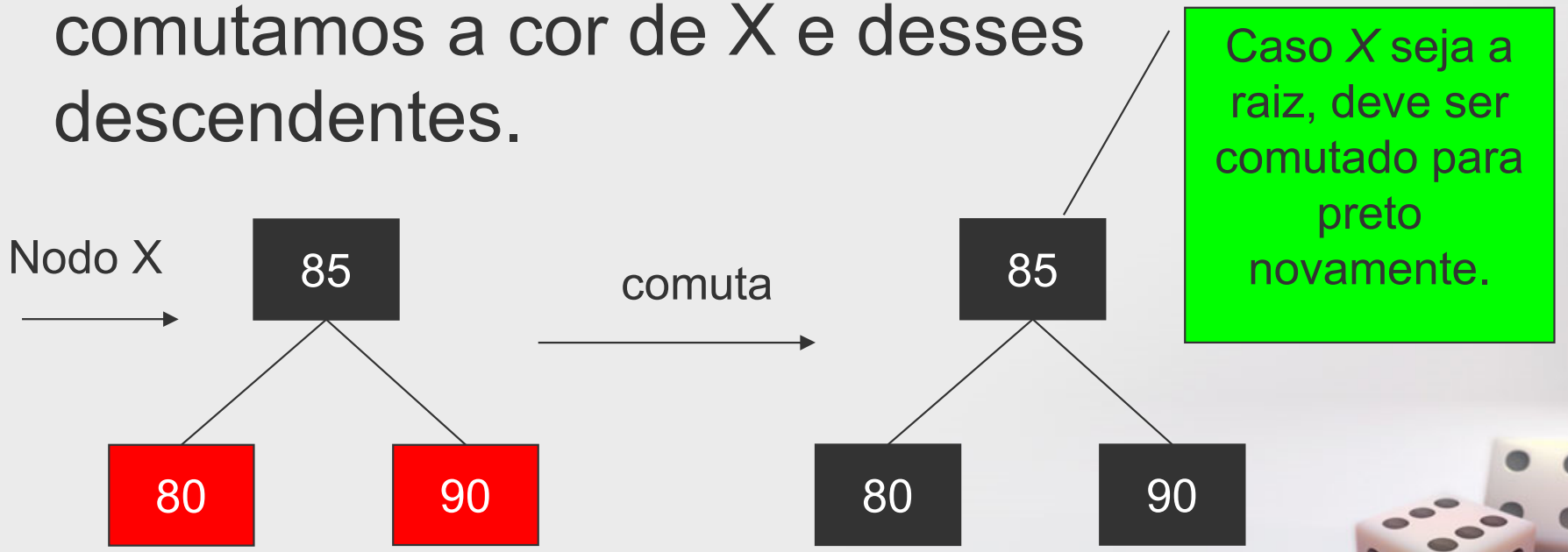
Árvores Red-Black

- Ideia Básica:
  - Na travessia à procura do ponto de inserção, sempre que passamos por um nodo X que tem dois descendentes vermelhos, comutamos a cor de X e desses descendentes.



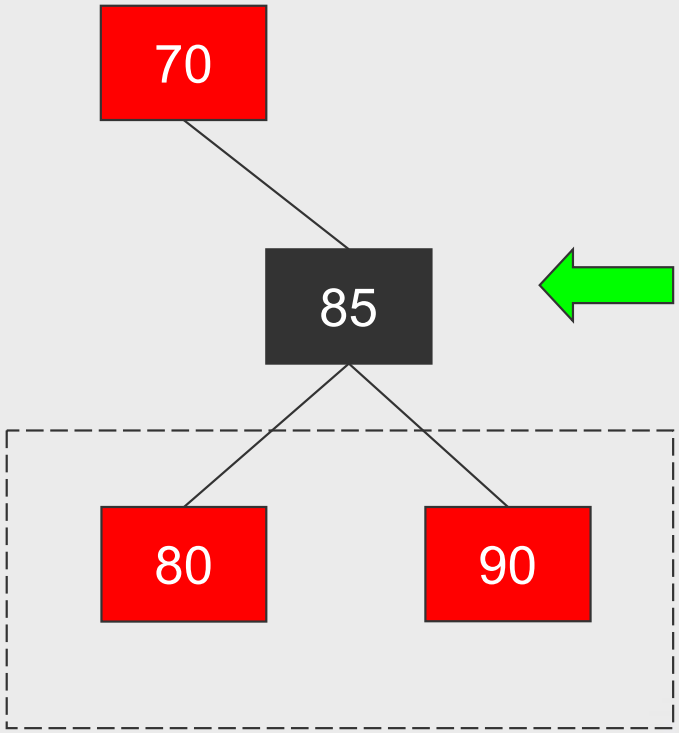
Árvores Red-Black

- Ideia Básica:
  - Na travessia à procura do ponto de inserção, sempre que passamos por um nodo X que tem dois descendentes vermelhos, comutamos a cor de X e desses descendentes.



Árvores Red-Black

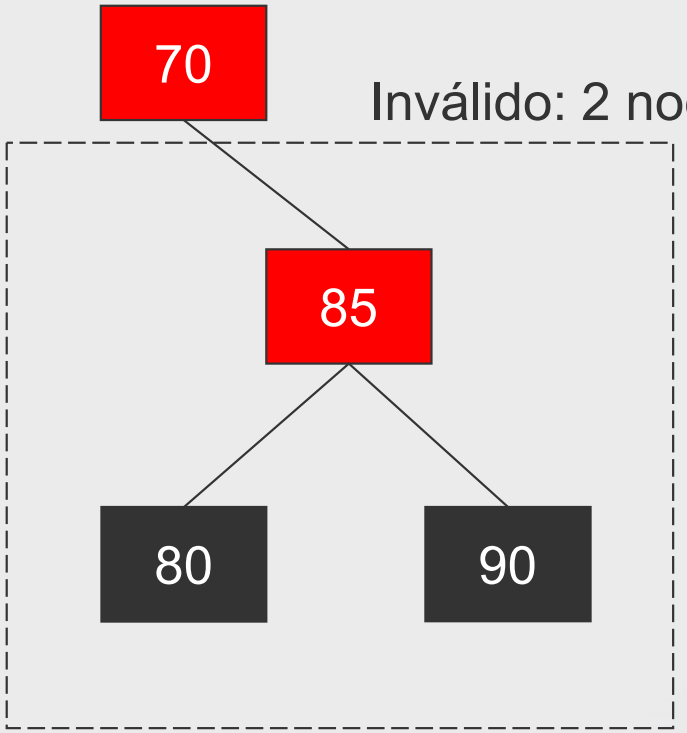
- E se o antecessor de *X* for vermelho?



Encontrei 2 descendentes vermelhos… vou comutar!

Árvores Red-Black

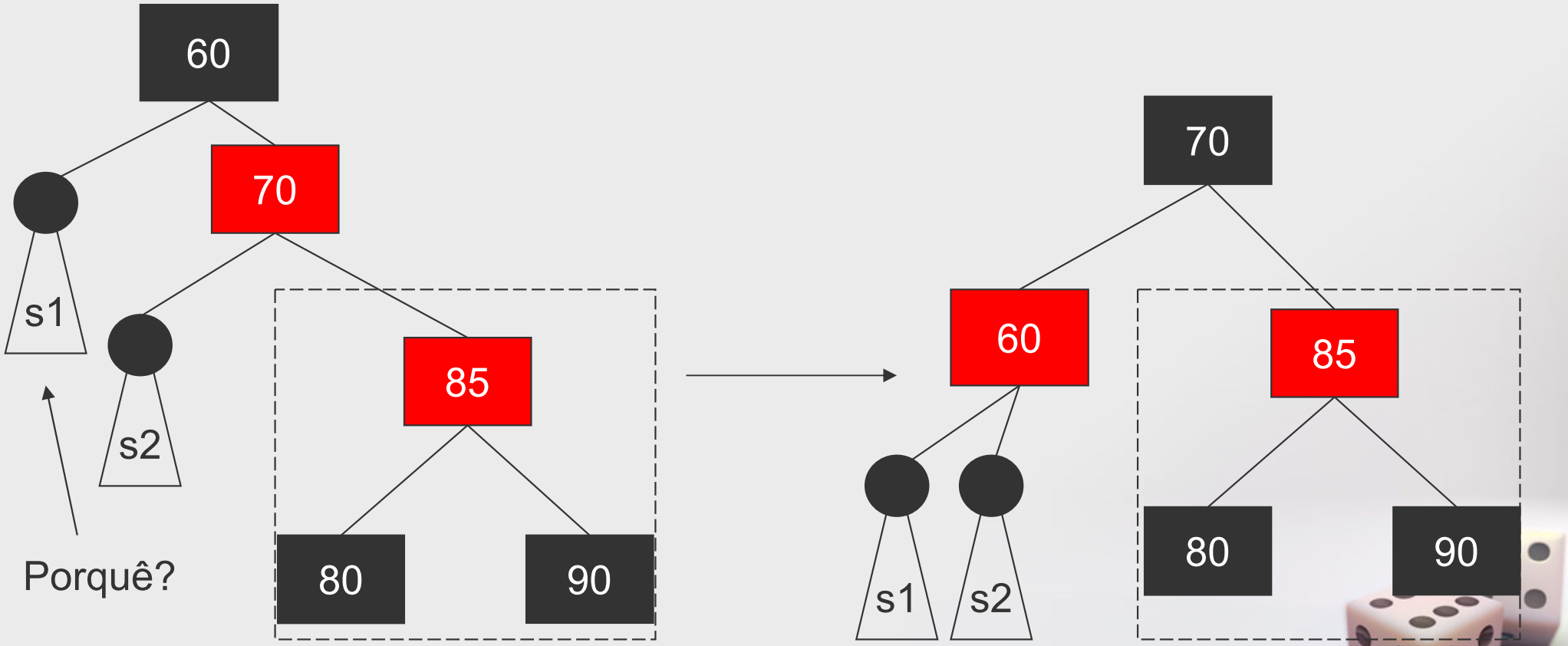
- E se o antecessor de *X* for vermelho?



Inválido: 2 nodos vermelhos consecutivos

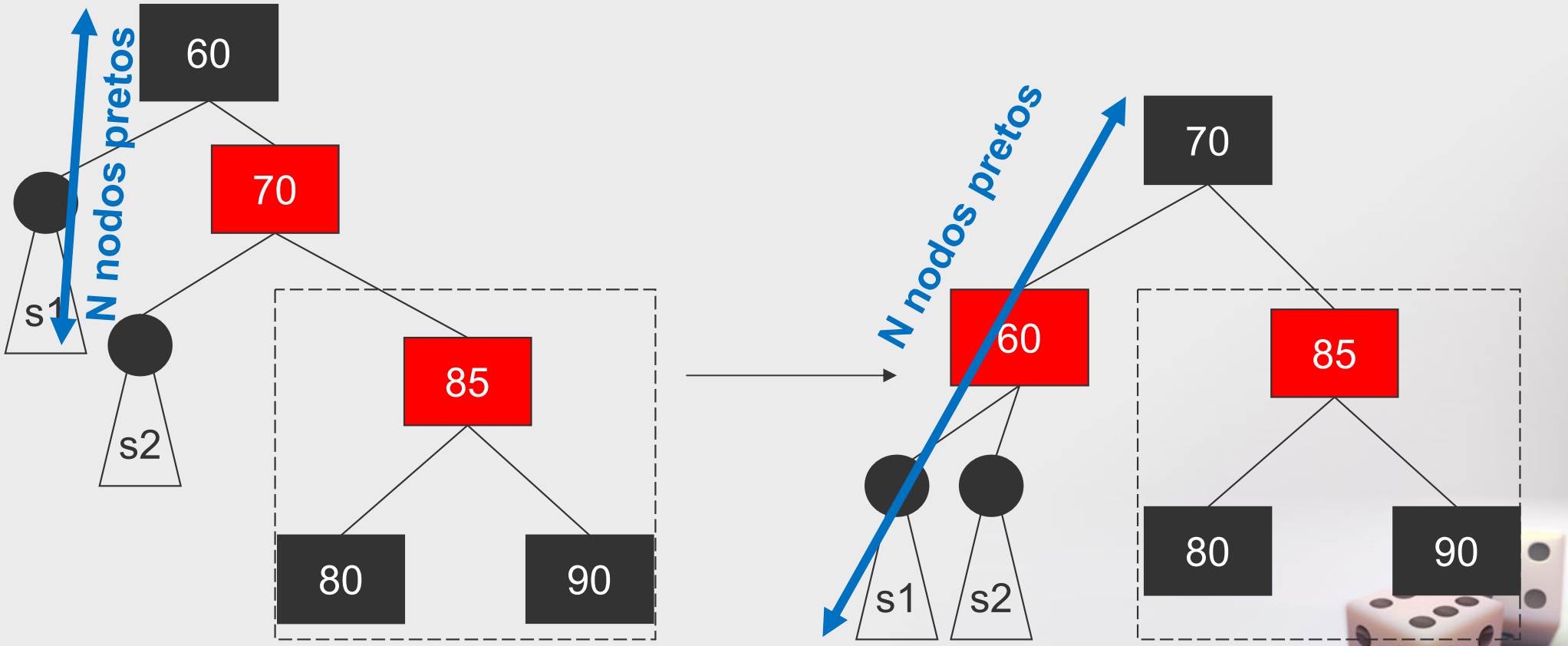
Árvores Red-Black

- E se o antecessor de *X* for vermelho?
  - Pode aplicar-se uma rotação (simples ou dupla)



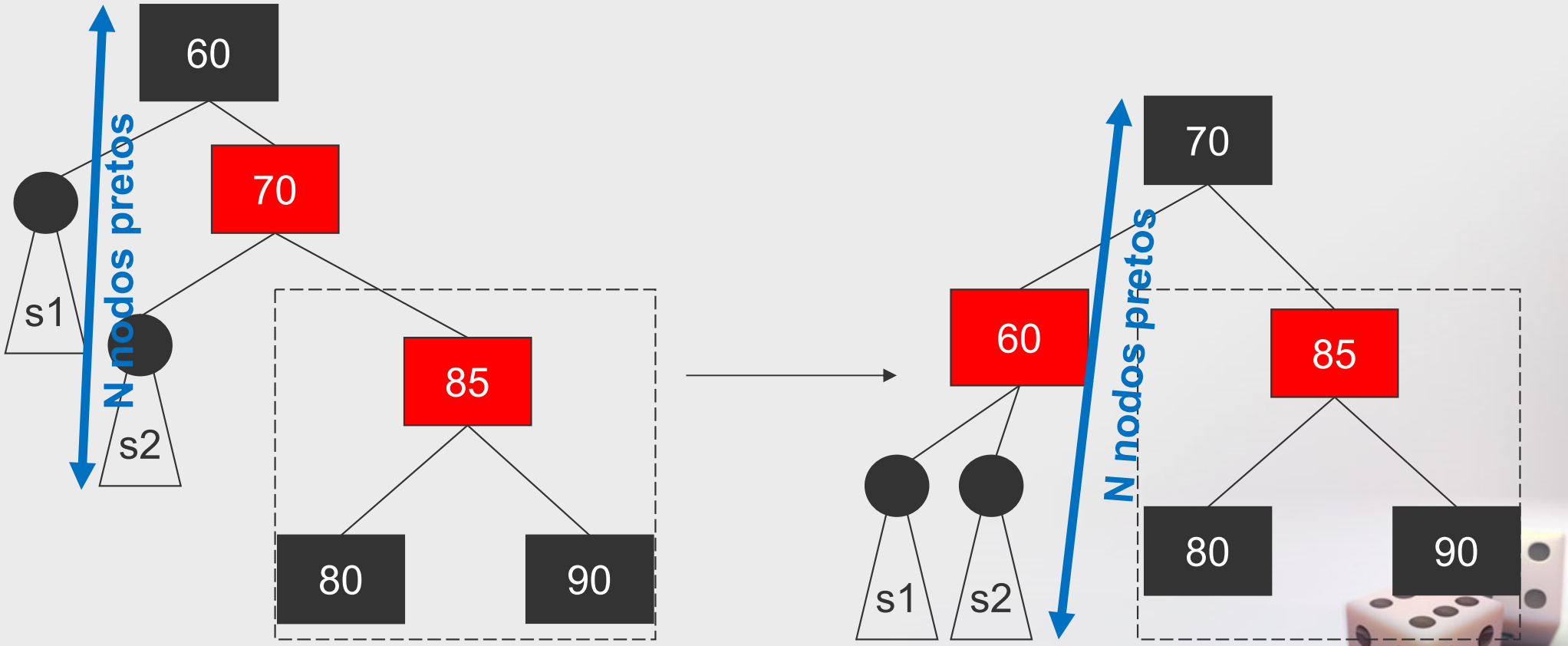
Árvores Red-Black

- Esta rotação preserva o equilibrio.



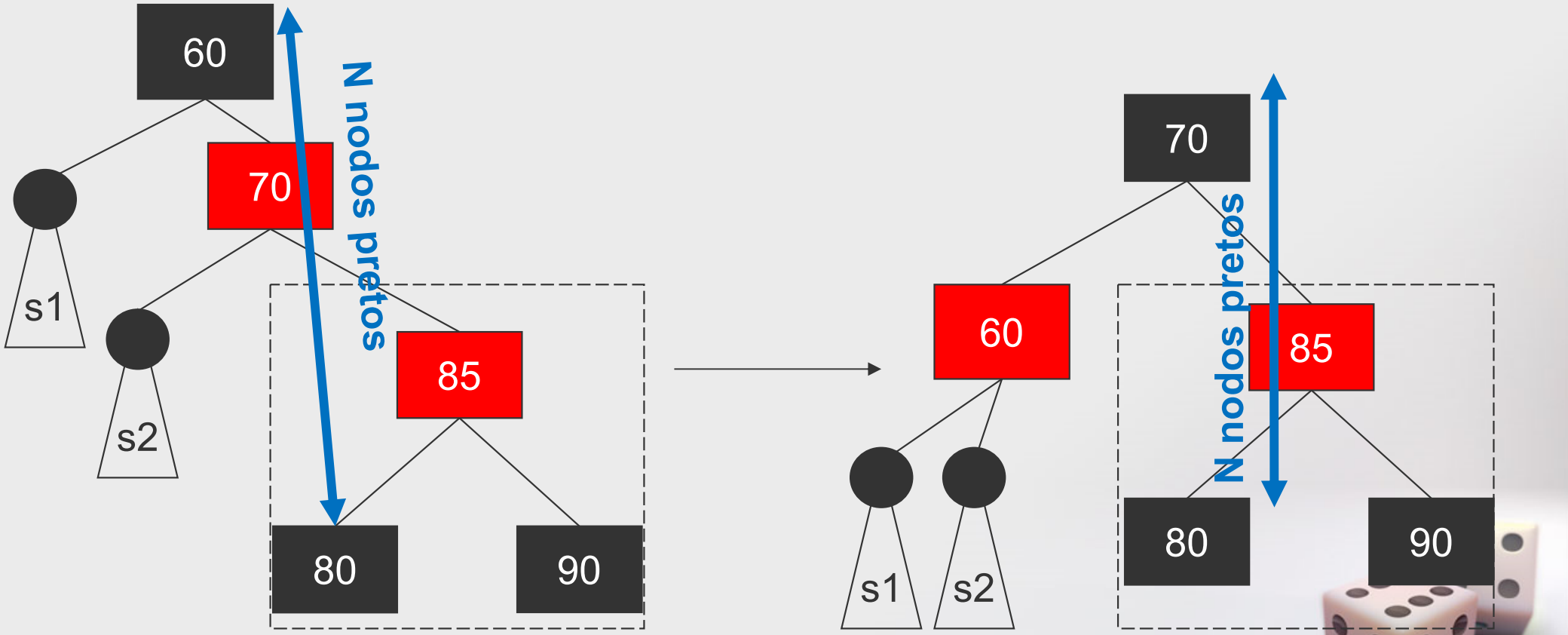
Árvores Red-Black

- Esta rotação preserva o equilibrio.



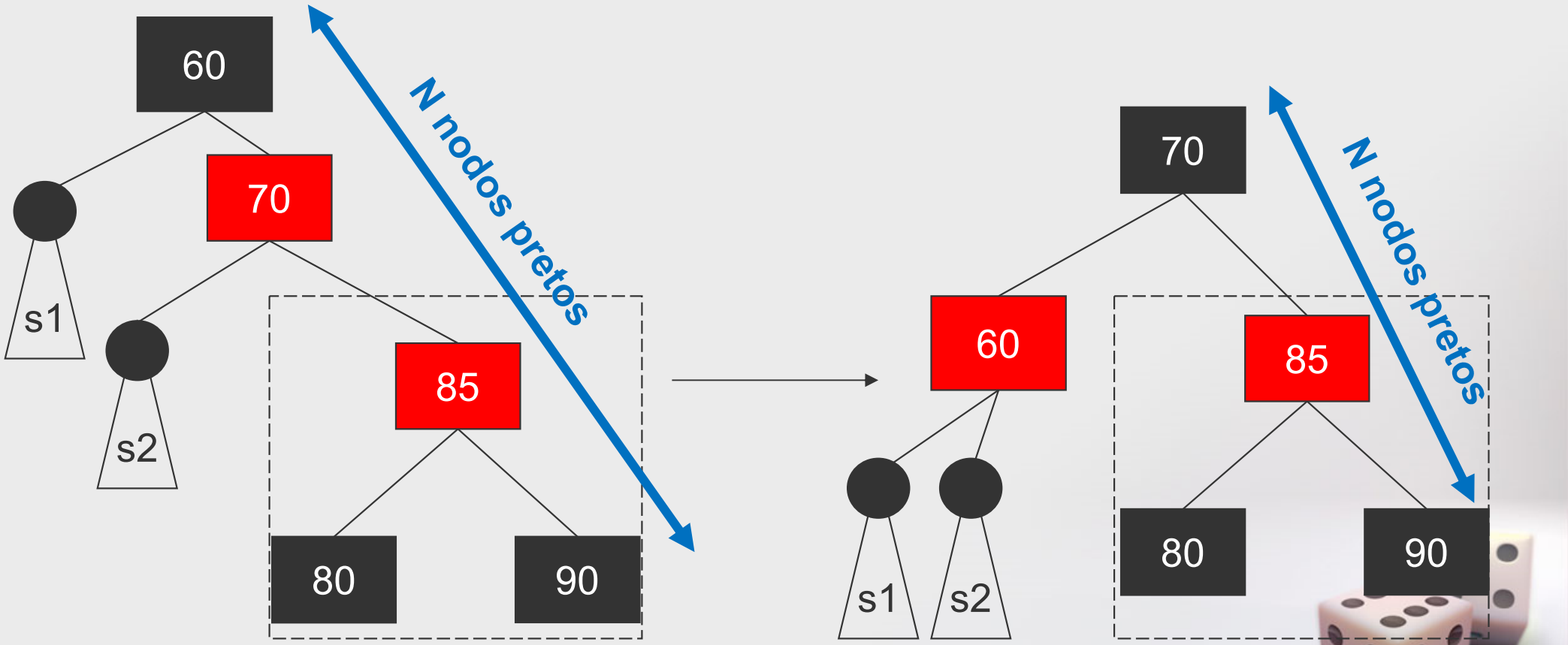
Árvores Red-Black

- Esta rotação preserva o equilibrio.



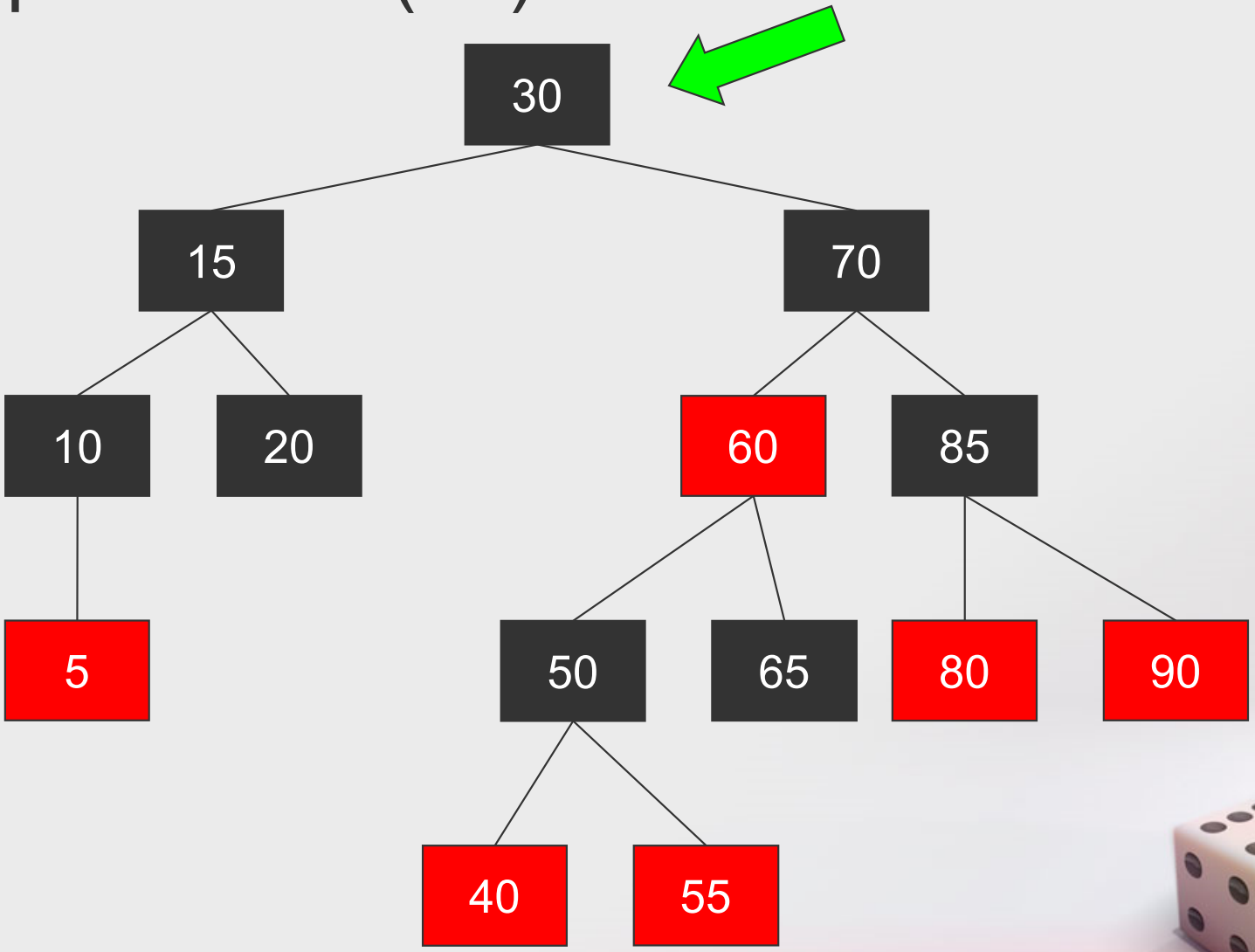
Árvores Red-Black

- Esta rotação preserva o equilibrio.



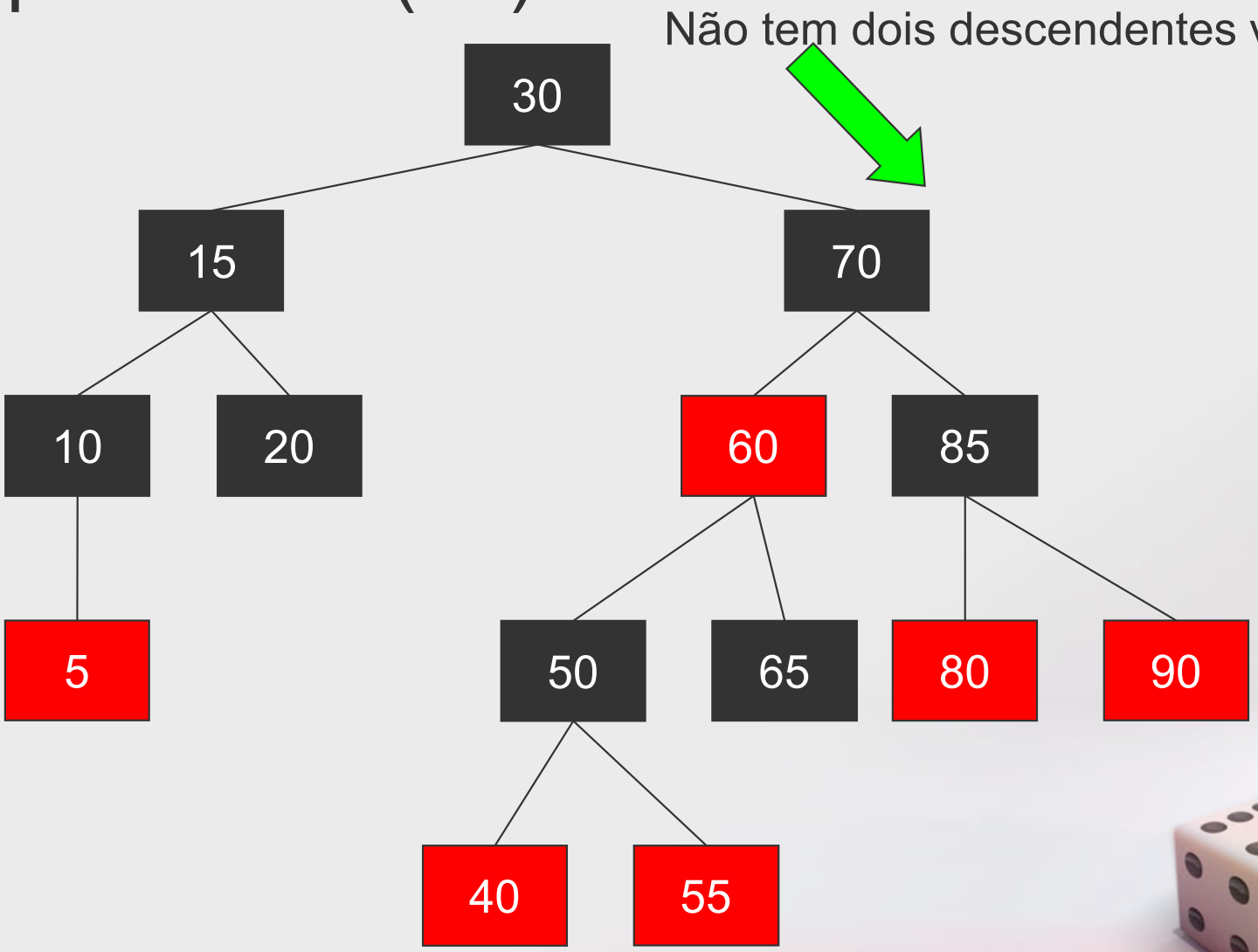
Árvores Red-Black

- Exemplo: Insere(45)



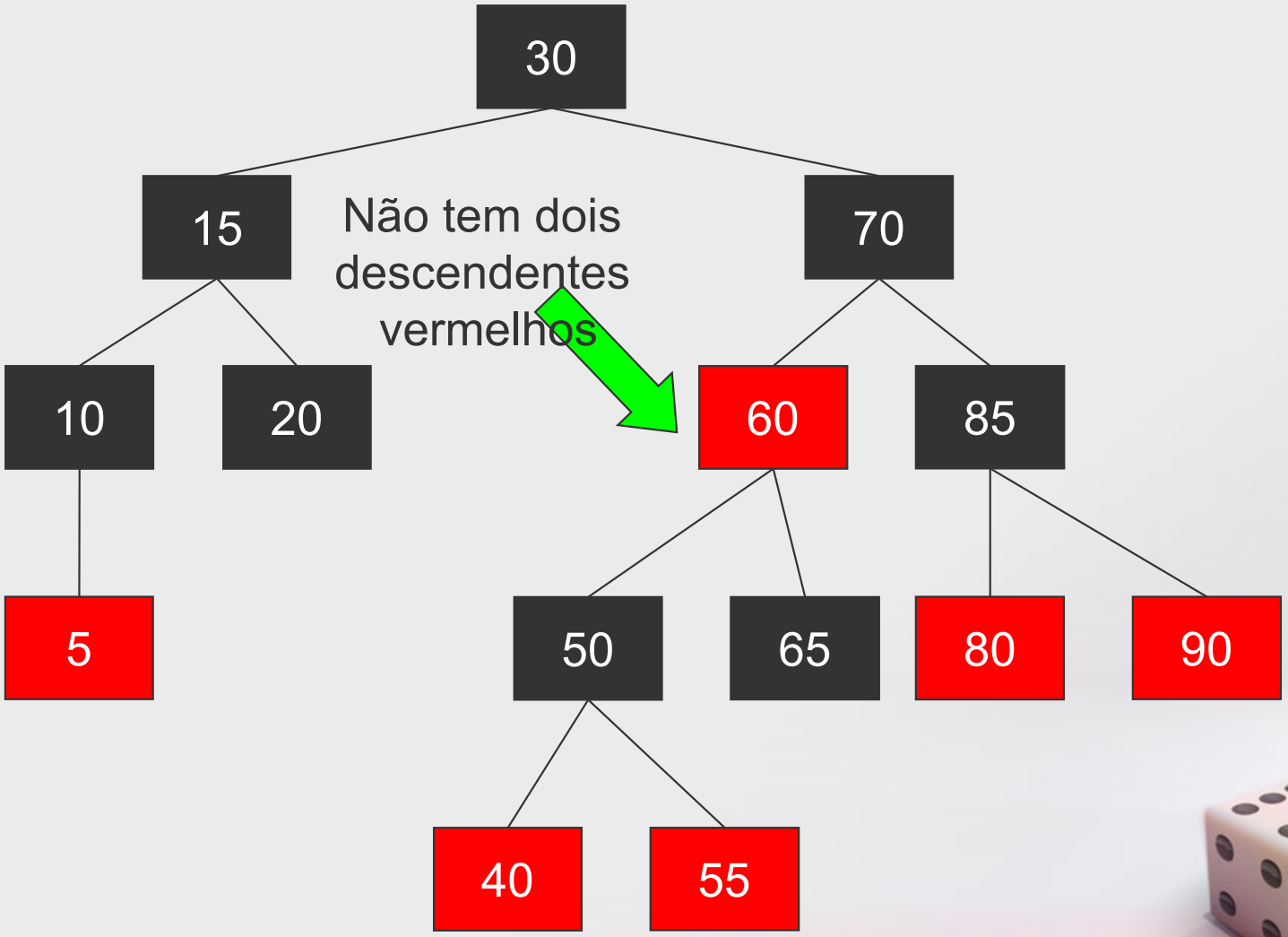
Árvores Red-Black

- Exemplo: Insere(45)



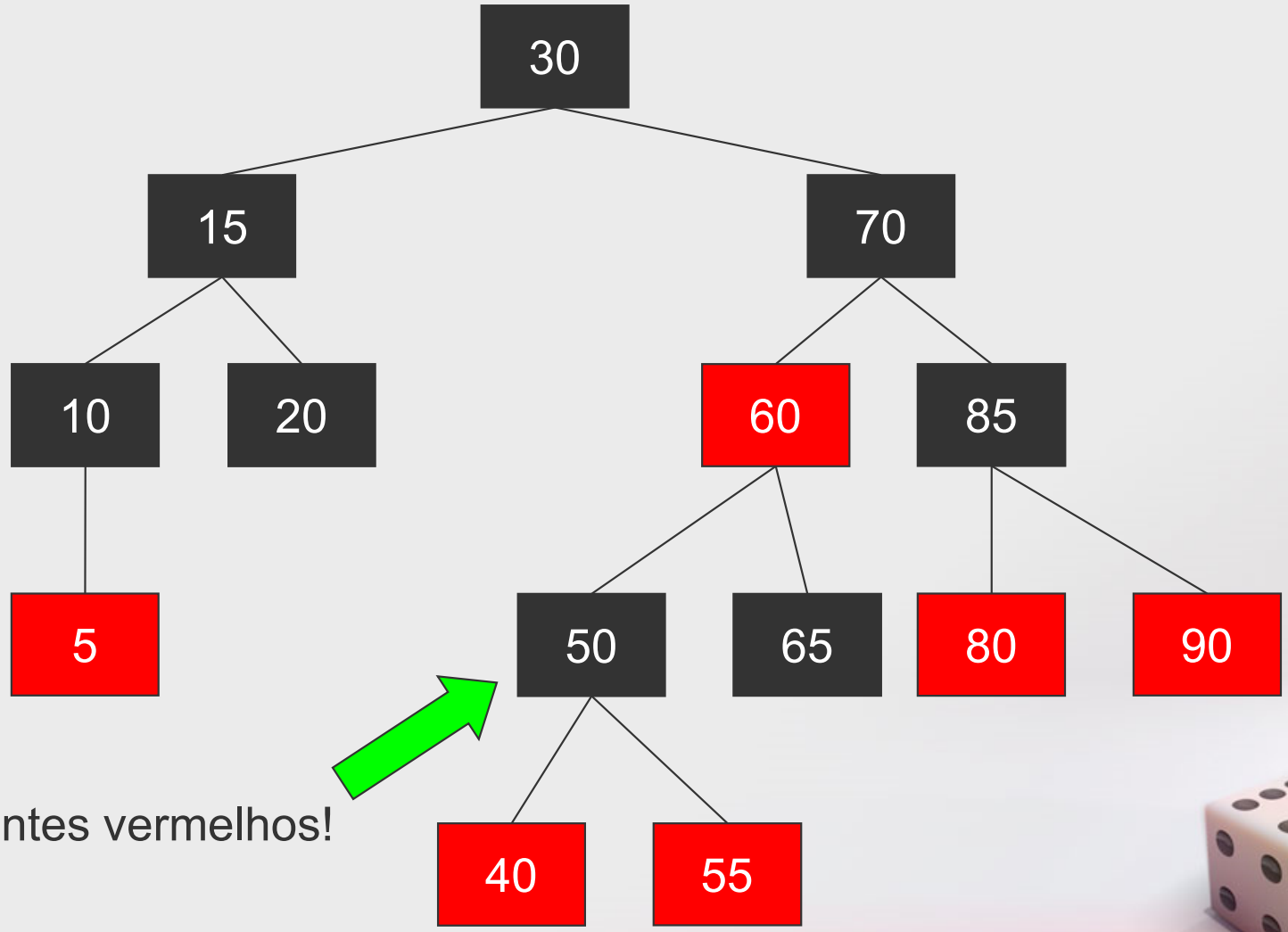
Árvores Red-Black

- Exemplo: Insere(45)



Árvores Red-Black

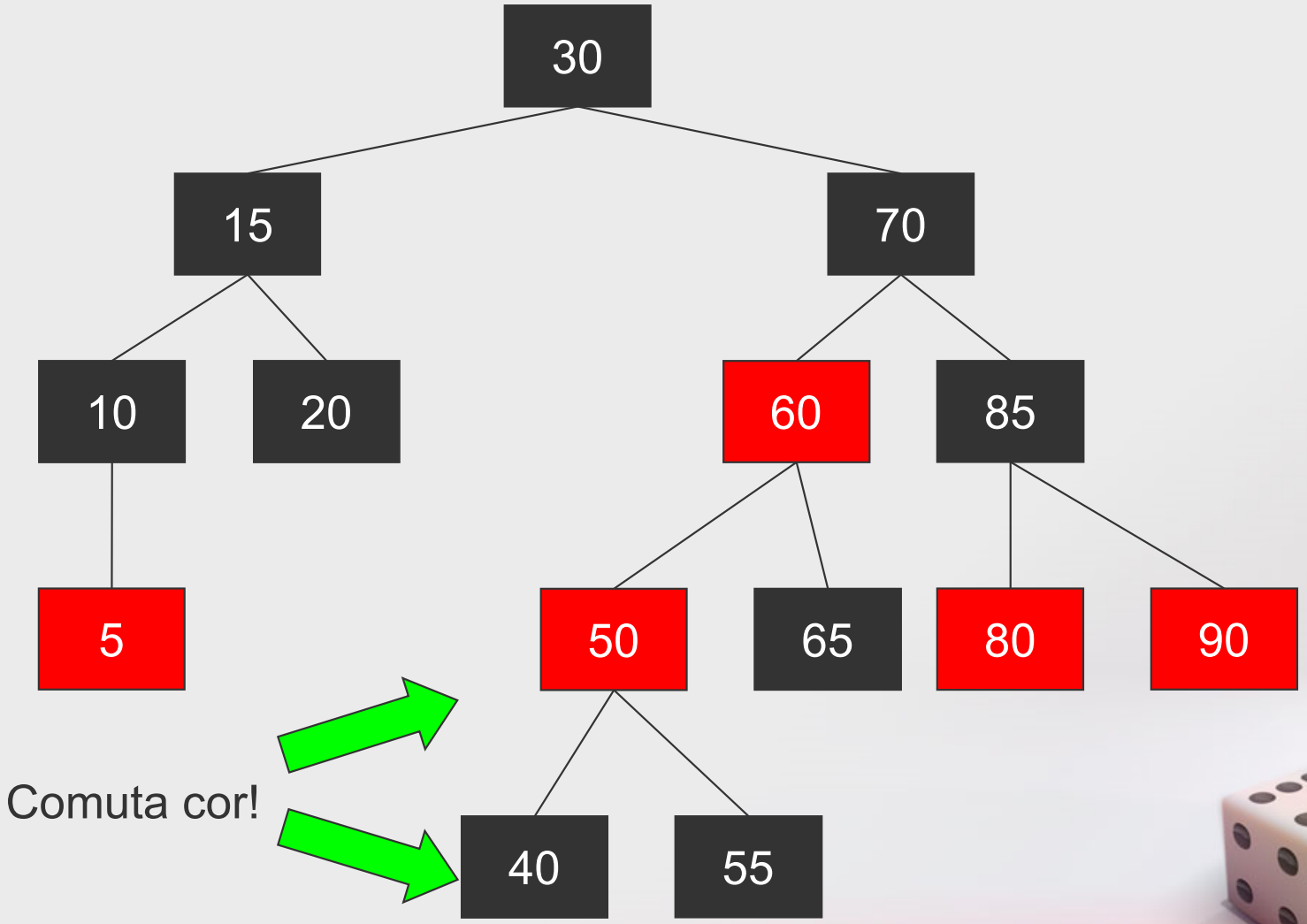
- Exemplo: Insere(45)



Dois descendentes vermelhos!

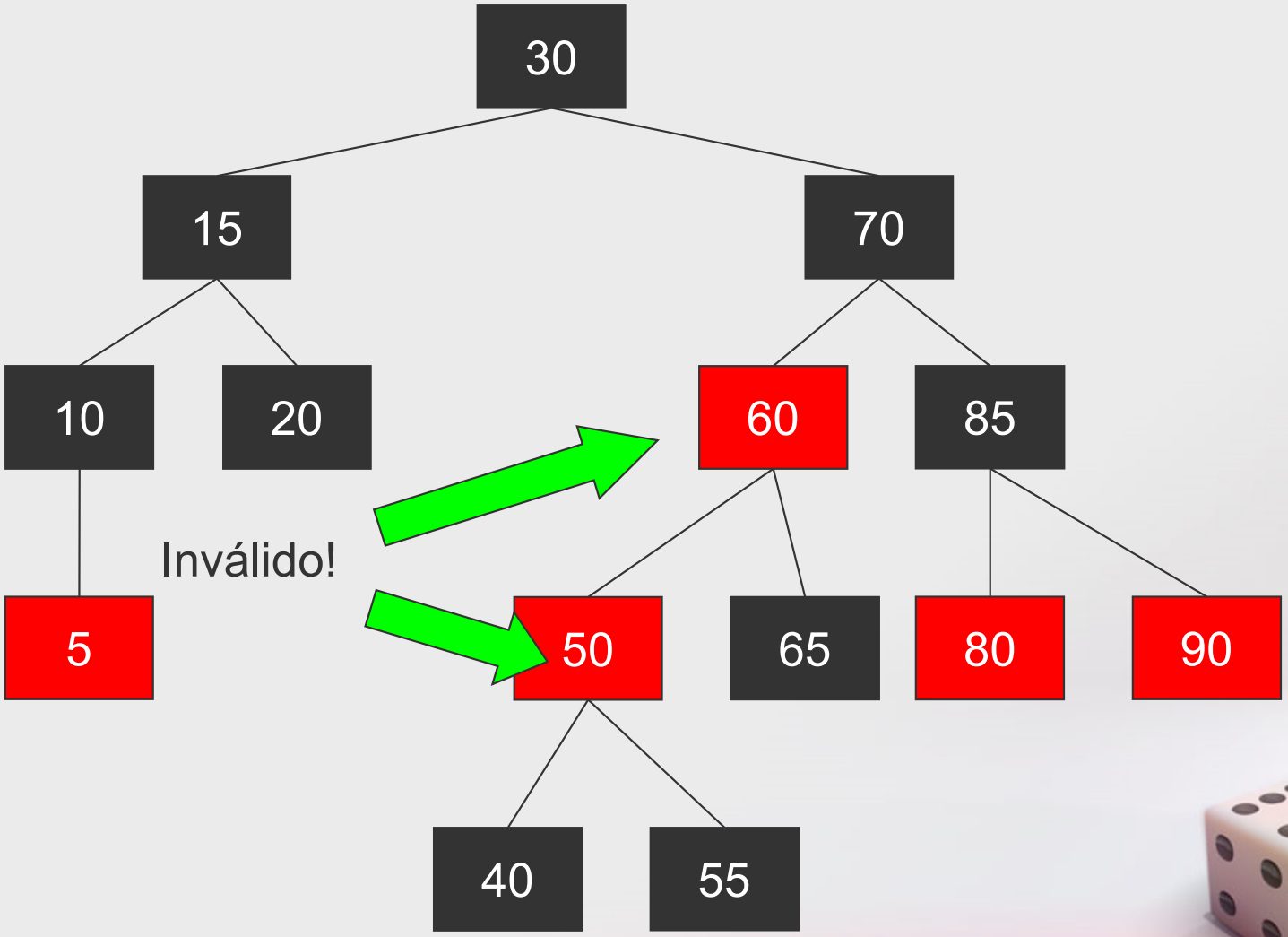
Árvores Red-Black

- Exemplo: Insere(45)



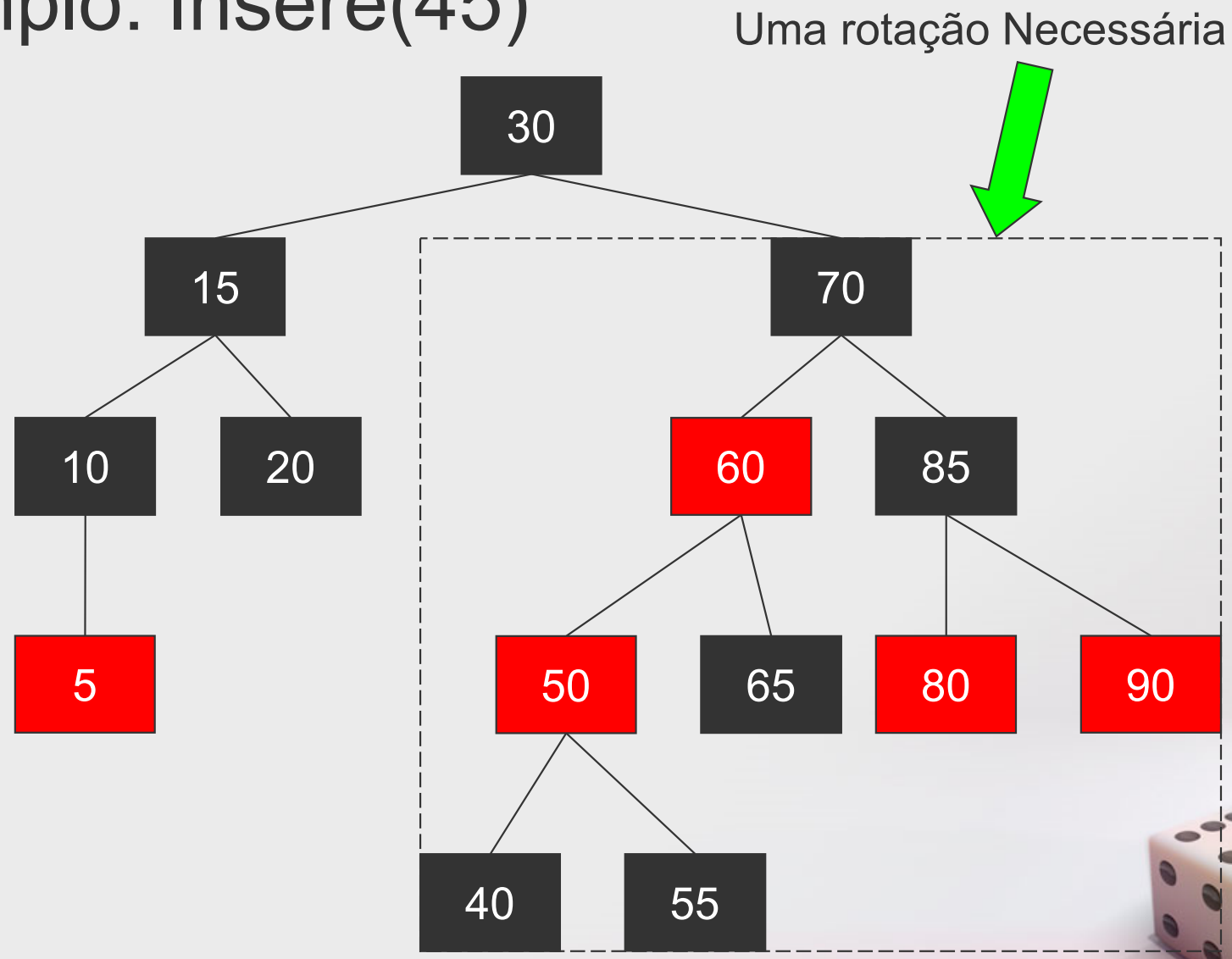
Árvores Red-Black

- Exemplo: Insere(45)



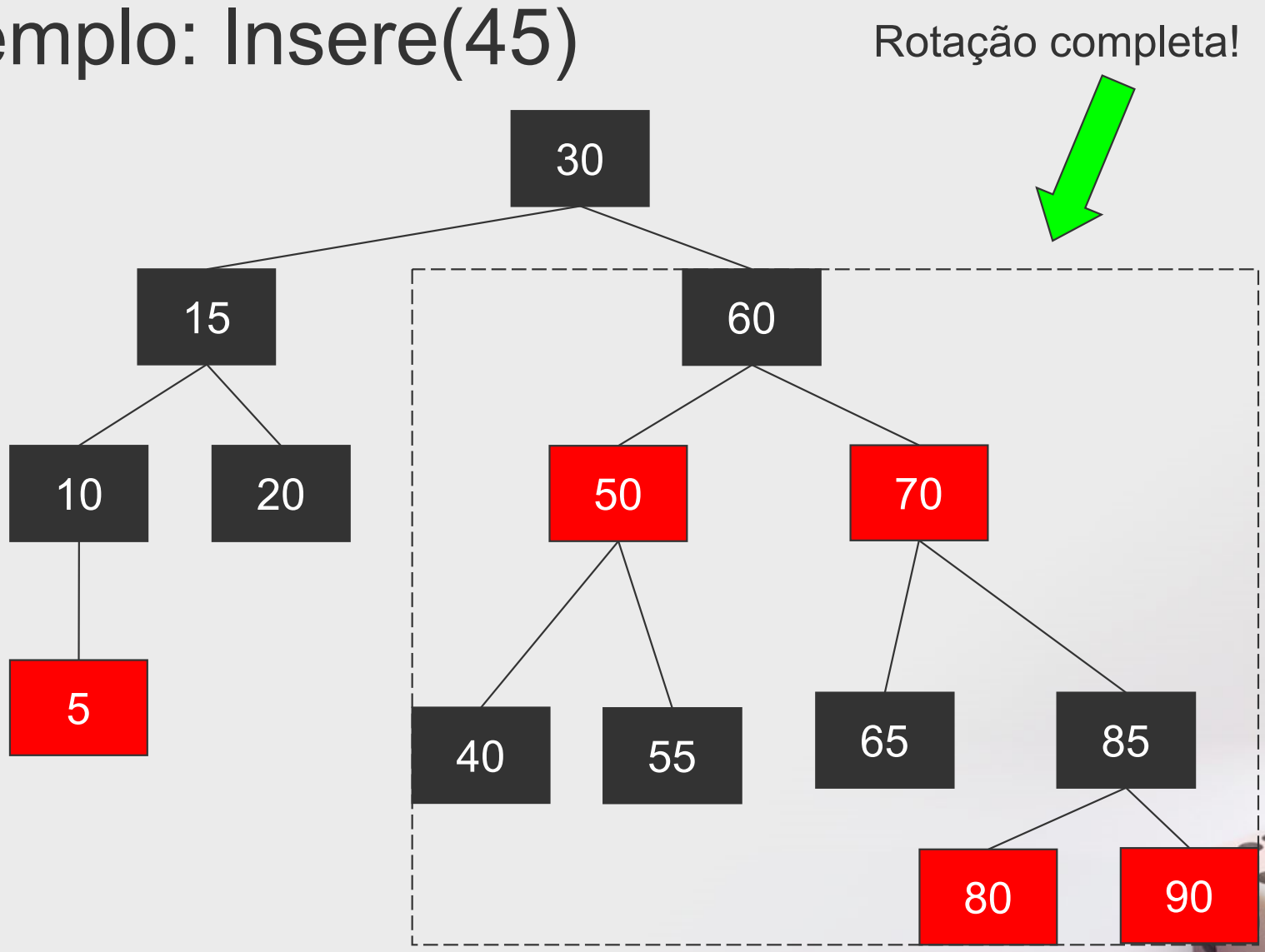
Árvores Red-Black

- Exemplo: Insere(45)



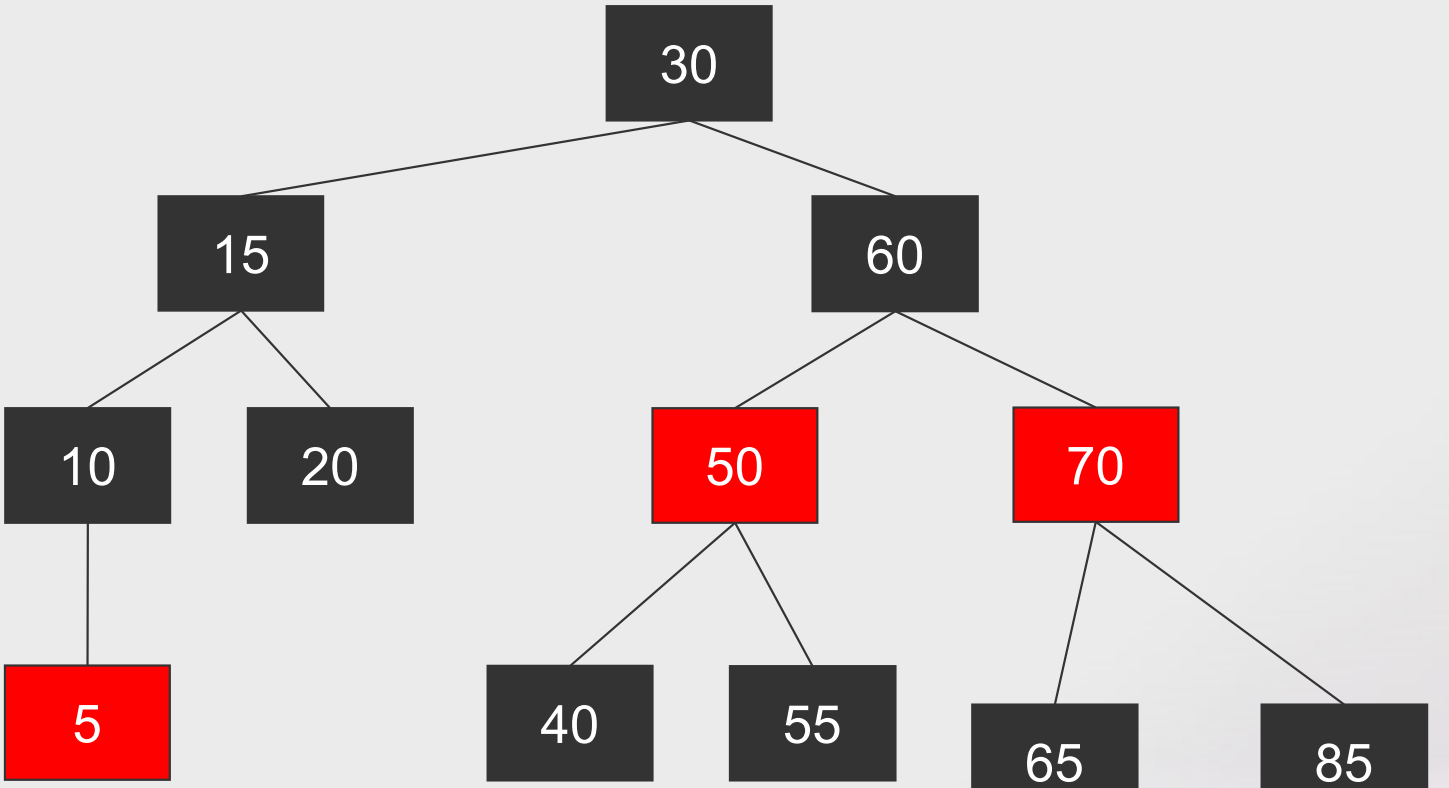
Árvores Red-Black

- Exemplo: Insere(45)



Árvores Red-Black

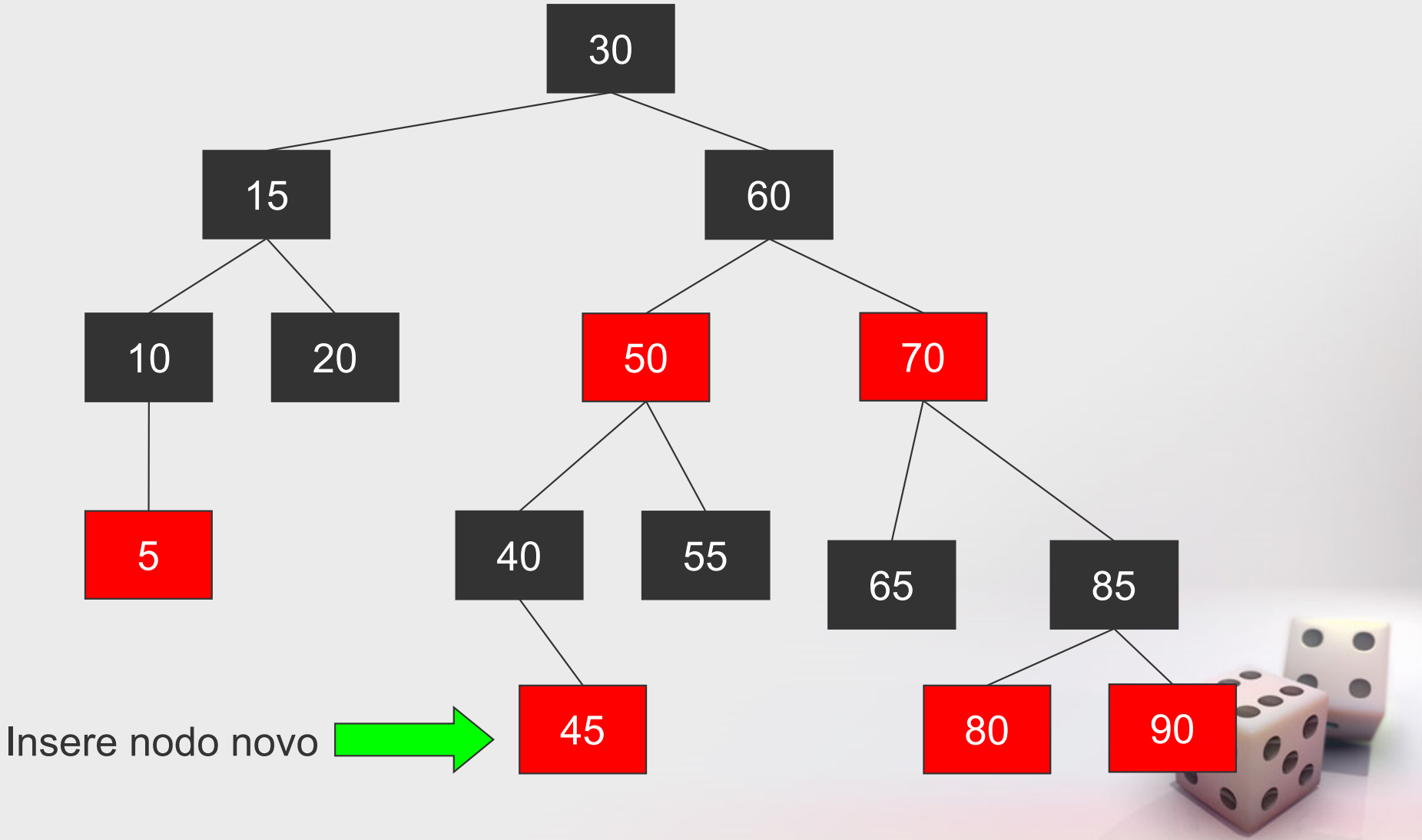
- Exemplo: Insere(45)



Continua a procurar e encontra local de inserção

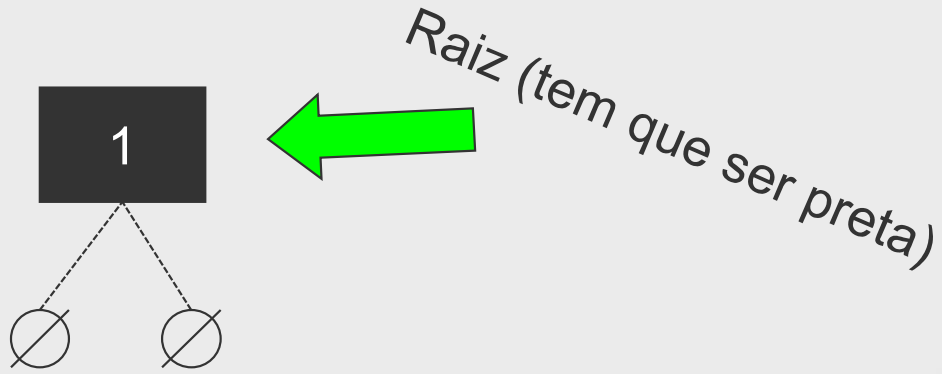
Árvores Red-Black

- Exemplo: Insere(45)



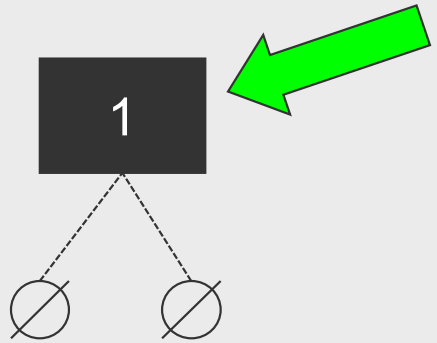
Árvores Red-Black

- Exemplo: Inserção de sequência
- Insere(1)



Árvores Red-Black

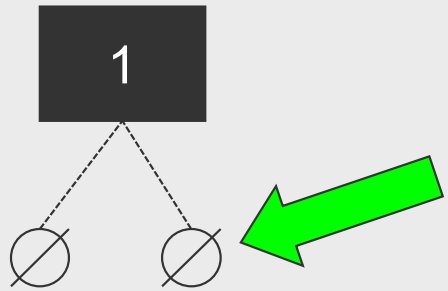
- Exemplo: Inserção de sequência
- Insere(2)



Descendentes não são RR
Não faz nada

Árvores Red-Black

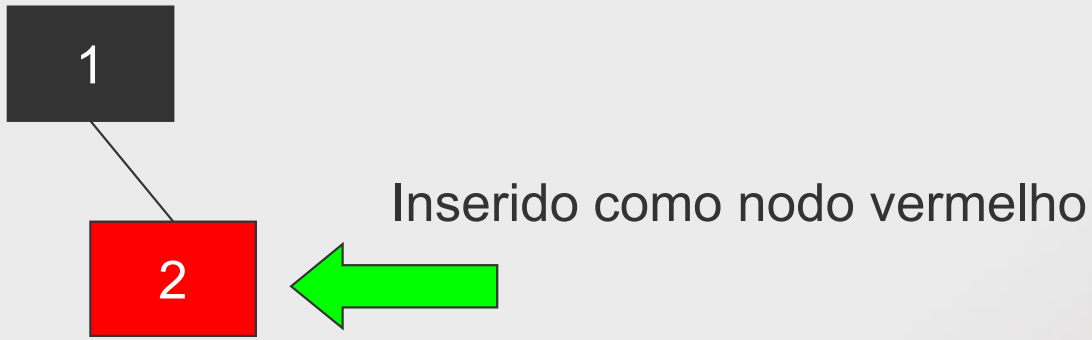
- Exemplo: Inserção de sequência
- Insere(2)



Local de inserção

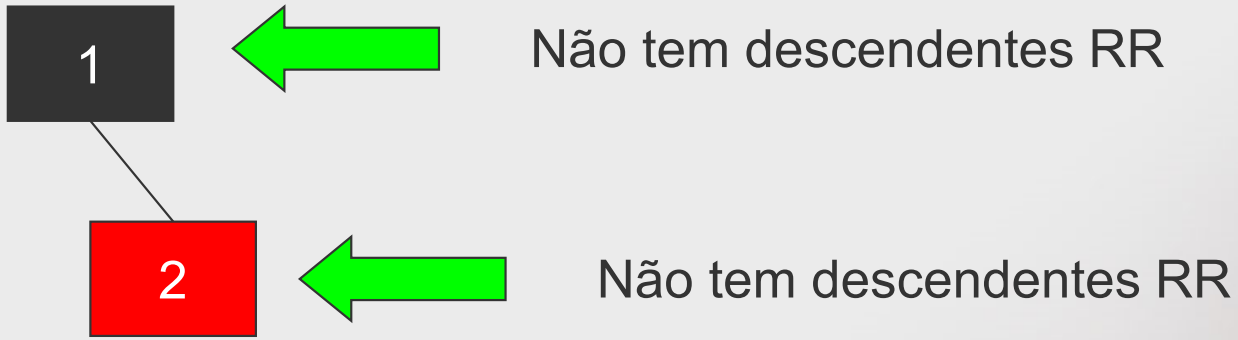
Árvores Red-Black

- Exemplo: Inserção de sequência
- Insere(2)



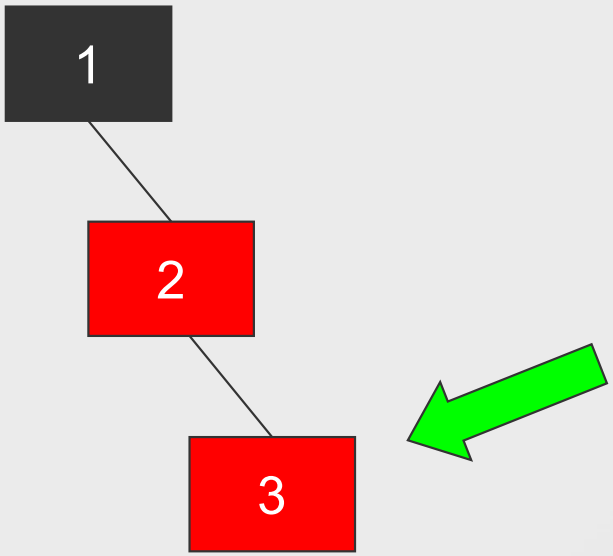
Árvores Red-Black

- Exemplo: Inserção de sequência
- Insere(3)



Árvores Red-Black

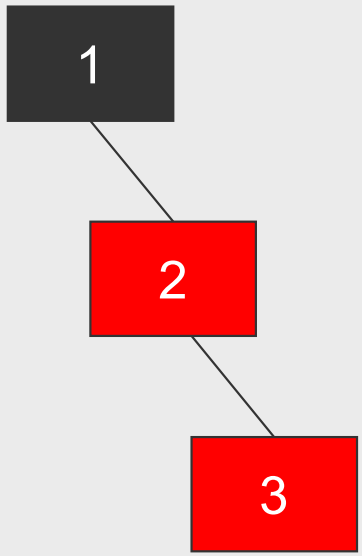
- Exemplo: Inserção de sequência
- Insere(3)



Novo nodo inserido como vermelho

Árvores Red-Black

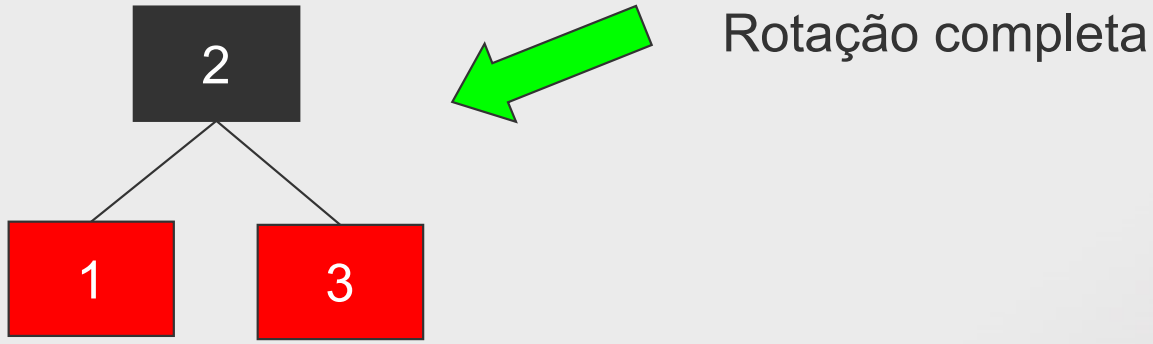
- Exemplo: Inserção de sequência
- Insere(3)



Dois nodos vermelhos consecutivos!
Rotação necessária

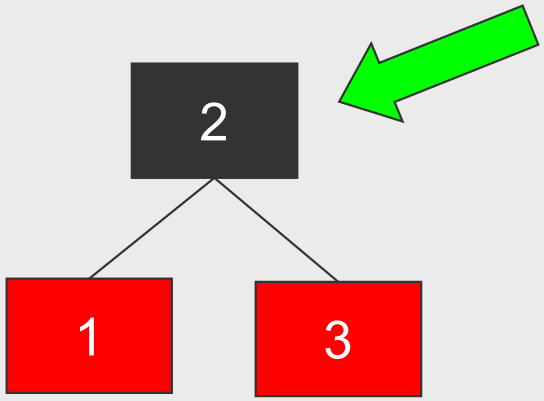
Árvores Red-Black

- Exemplo: Inserção de sequência
- Insere(3)



Árvores Red-Black

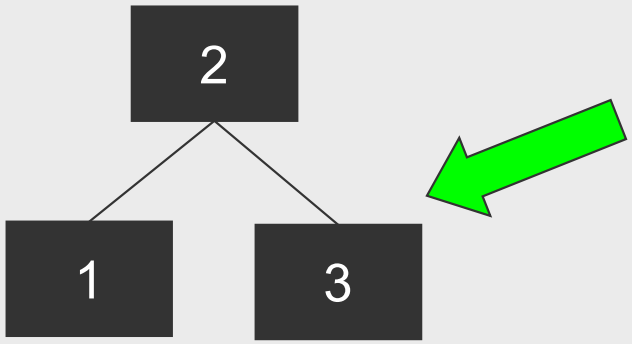
- Exemplo: Inserção de sequência
- Insere(4)



Dois descendentes RR!
É necessário comutar.

Árvores Red-Black

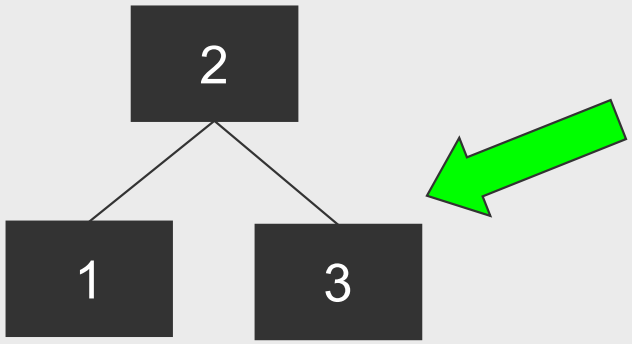
- Exemplo: Inserção de sequência
- Insere(4)



Cores comutadas!.

Árvores Red-Black

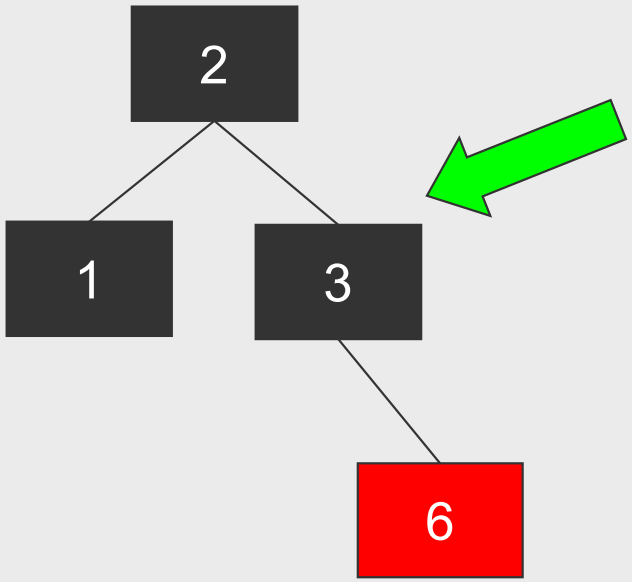
- Exemplo: Inserção de sequência
- Insere(4)



Continua pesquisa, não são necessárias mais comutações

Árvores Red-Black

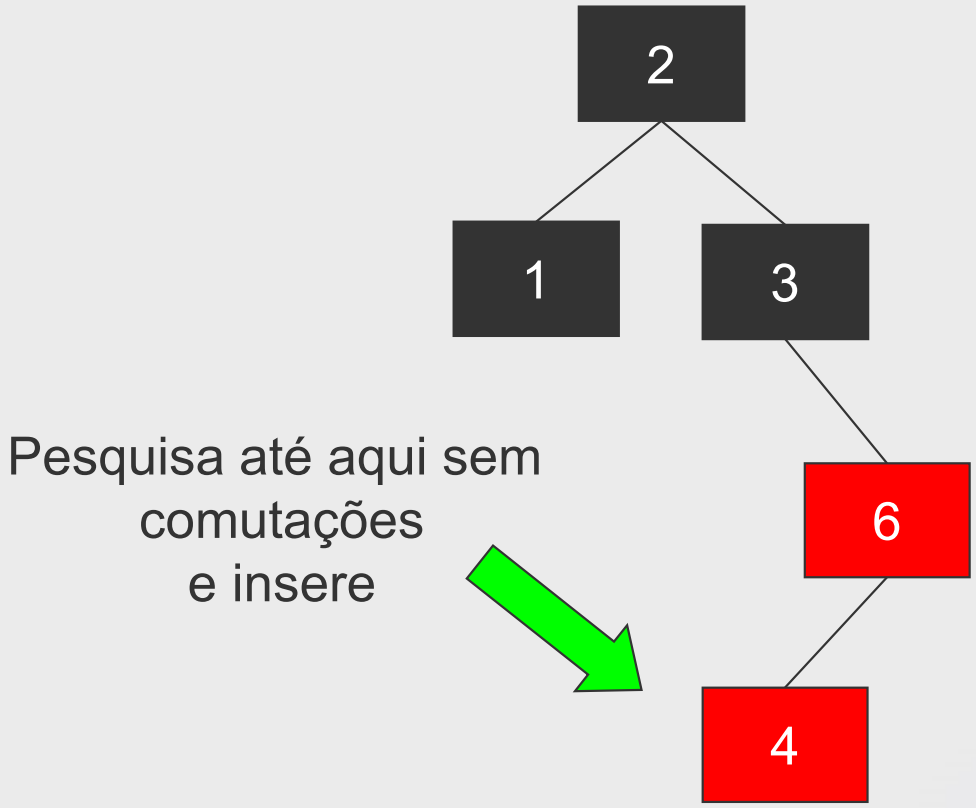
- Exemplo: Inserção de sequência
- Insere(6)



Continua pesquisa, não são necessárias mais comutações

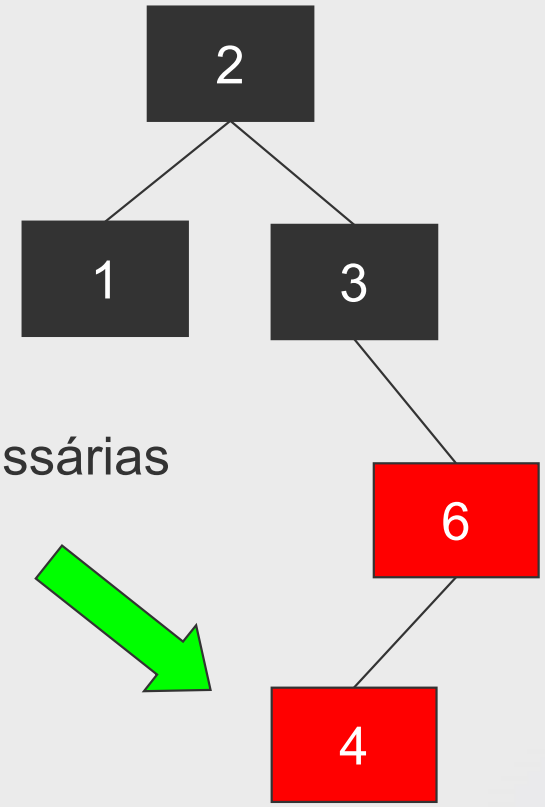
Árvores Red-Black

- Exemplo: Inserção de sequência
- Insere(4)



Árvores Red-Black

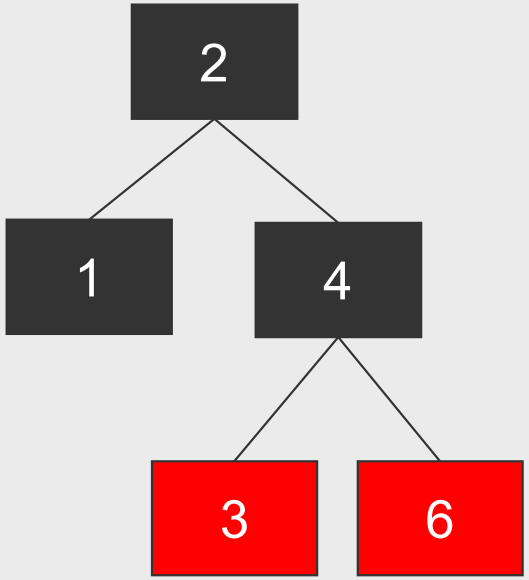
- Exemplo: Inserção de sequência
- Insere(4)



Duas rotações necessárias

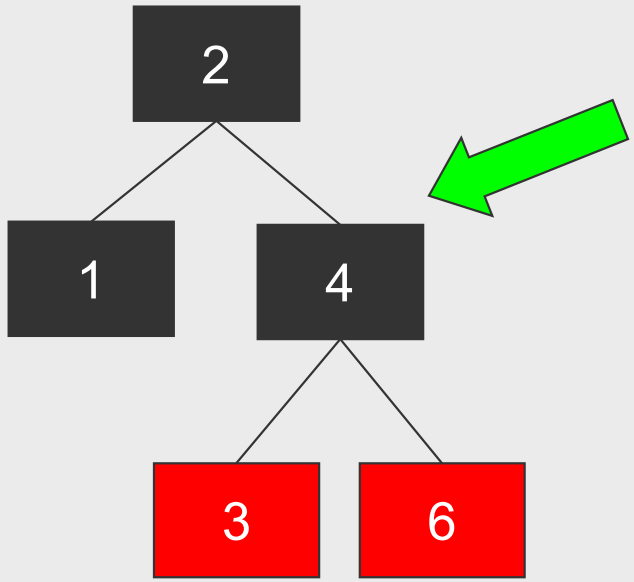
Árvores Red-Black

- Exemplo: Inserção de sequência
- Insere(4)



Árvores Red-Black

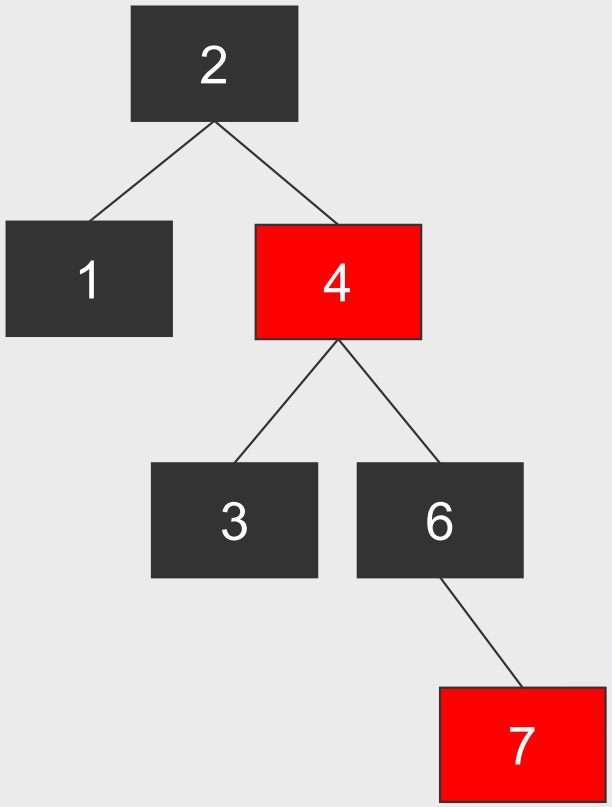
- Exemplo: Inserção de sequência
- Insere(7)



Comutação necessária

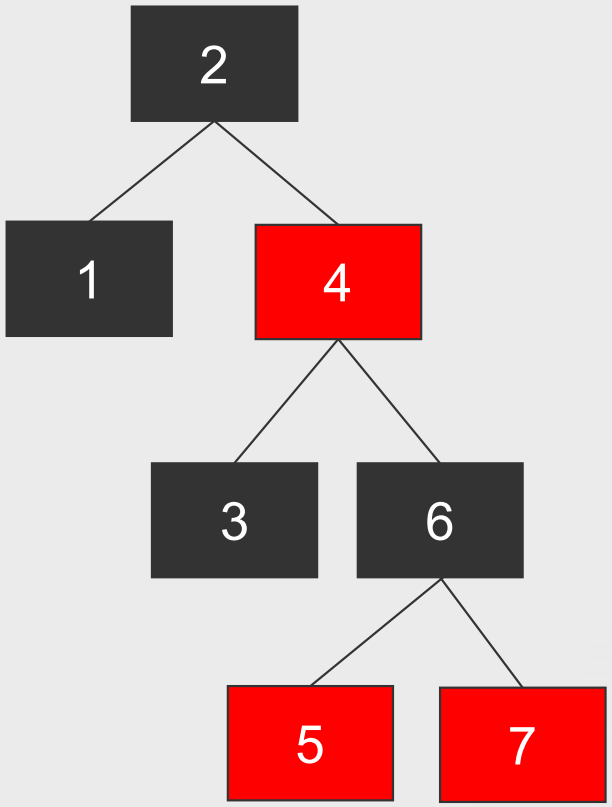
Árvores Red-Black

- Exemplo: Inserção de sequência
- Insere(7)



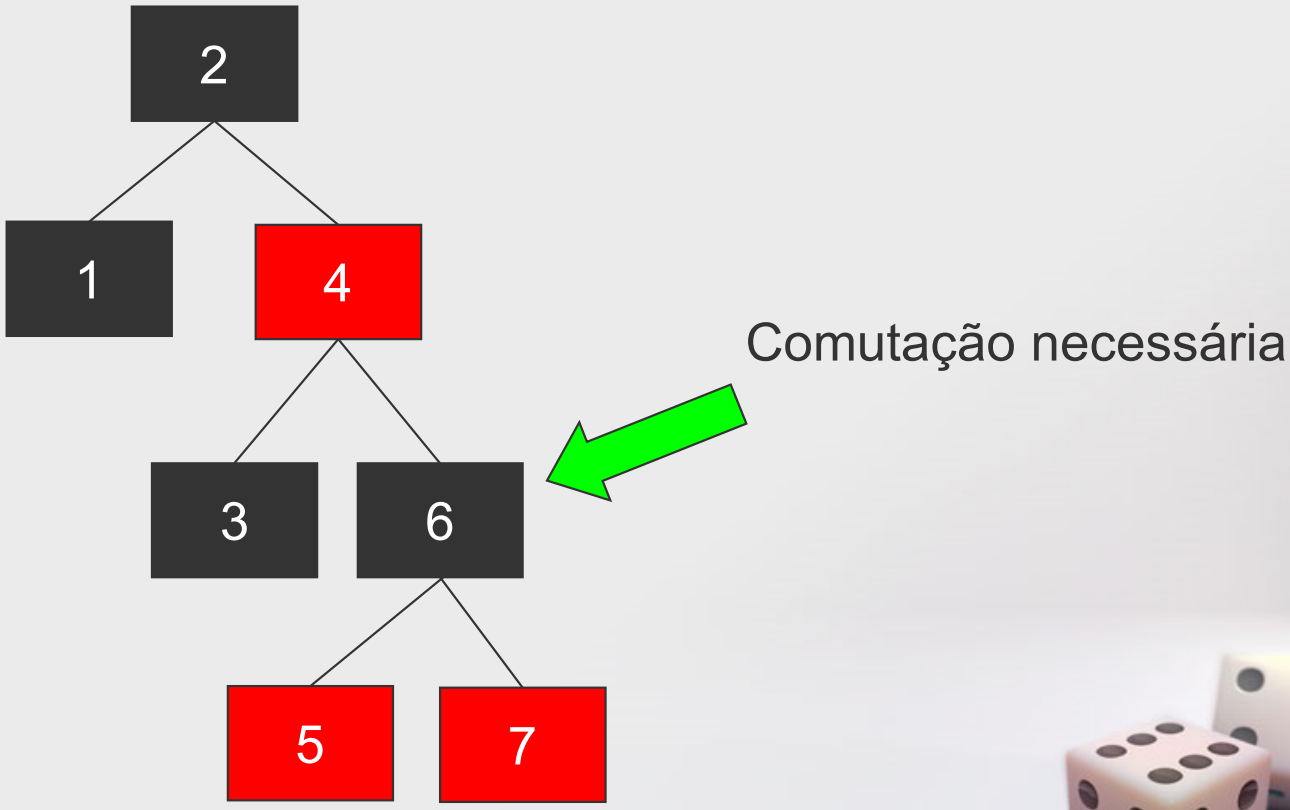
Árvores Red-Black

- Exemplo: Inserção de sequência
- Insere(5)



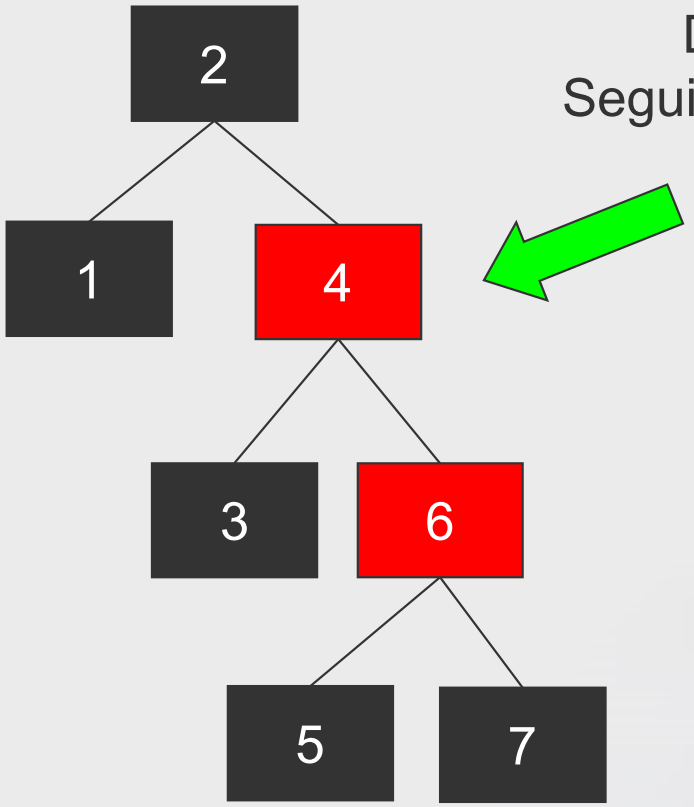
Árvores Red-Black

- Exemplo: Inserção de sequência
- Insere(8)



Árvores Red-Black

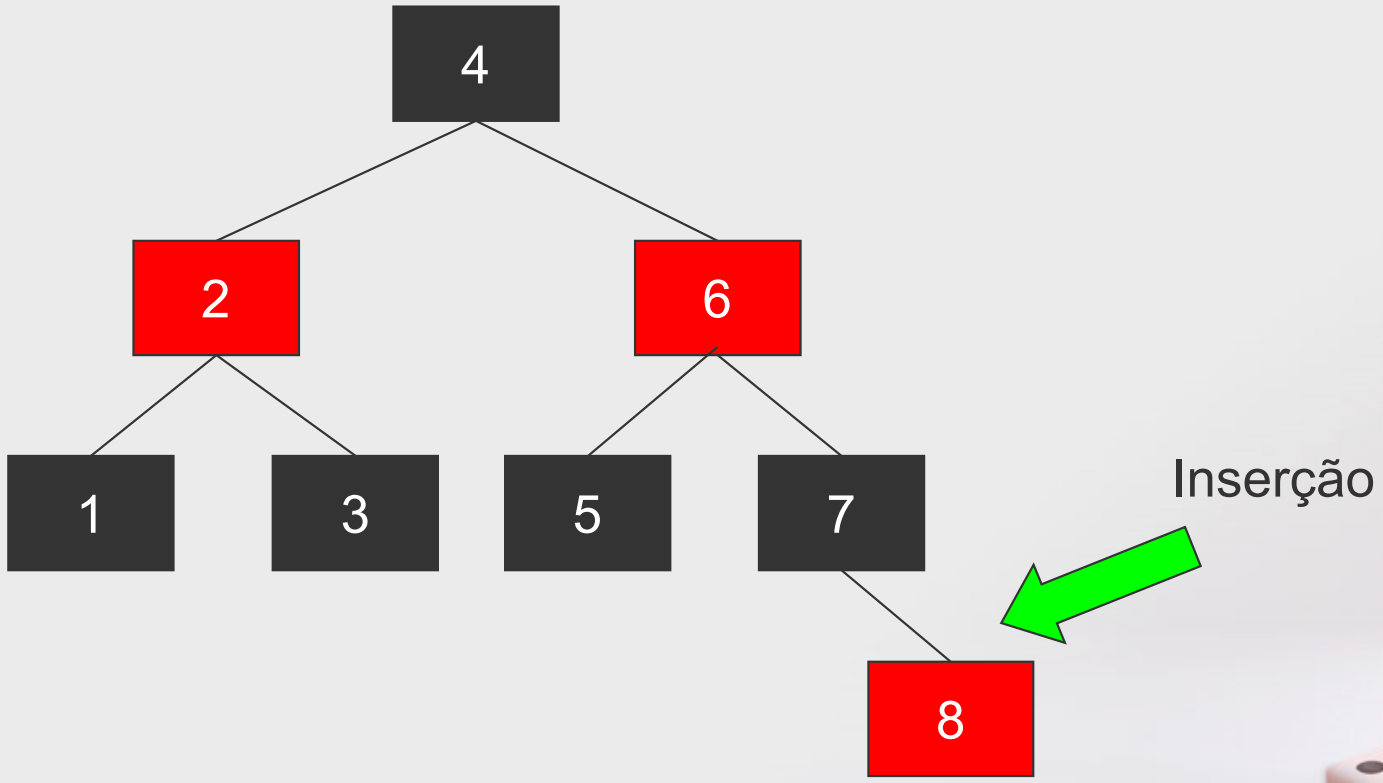
- Exemplo: Inserção de sequência
- Insere(8)



Dois nodos vermelhos Seguidos. Rotação (entre 2 e 4) e recoloração necessária…

Árvores Red-Black

- Exemplo: Inserção de sequência
- Insere(8)



Árvores AVL vs. Árvores Red-Black

- As àrvores AVL têm um profundidade ligeiramente menor, em termos teóricos:
  - AVL: 1,44 log(n+1)
  - Red-Black: 2 log(n+1)
- Apesar disso, experimentalmente verifica-se que o número médio de nodos percorrido é, em média, similar.
- A inserção em árvores Red-Black é mais eficiente.
  - Em termos práticos, as rotações são pouco frequentes.

Árvores Red-Black

- Implementação:
  - A implementação é complicada não só pela simetria como também pelos inúmeros casos especiais:
    - Raiz – Entre outros pormenores, não tem antecessor.
    - Sub-árvores vazias.
  - Para simplificar a implementação, são usadas duas *sentinelas*.
    - Uma sentinela é um valor (ou nodo) especial que é inserido numa estrutura de dados para simplificar algumas operações.

Árvore Red-Black

- Sentinelas:
  - **NullNode**: Existe um (único) nódo que representa um nodo nulo. Este nodo é sempre preto. Logo, não há *links* nulos.
  - **Header**: Pseudo-raiz (raiz da raiz). Tem um valor de -∞ e o seu descendente direito é a raiz da àrvore.

Árvore Red-Black

```
private static class RedBlackNode<AnyType>
{
// Construtores
RedBlackNode( AnyType theElement )
{
   this( theElement, null, null );
}
RedBlackNode( AnyType theElement,
RedBlackNode<AnyType> lt,
RedBlackNode<AnyType> rt )
{
   element = theElement;
   left = lt; right = rt; color = RedBlackTree.BLACK;
}
AnyType element;
RedBlackNode<AnyType> left;
RedBlackNode<AnyType> right;
int                   color;
}
```

Árvore Red-Black

```
public class RedBlackTree
<AnyType extends Comparable<? super AnyType>>
 {
    public RedBlackTree( ) { …}
    public void insert( AnyType item ) {…}
    public void remove( AnyType x ) {…}
    public AnyType findMin( )  {… }
    public AnyType findMax( ) {…}
    public AnyType find( AnyType x ) {…}
    public void makeEmpty( ) { header.right = nullNode; }
    public boolean isEmpty( )  return header.right == nullNode; }
    public void printTree( ) { printTree( header.right ); }

    …
}
```

Interface Externo

Árvore Red-Black

```
public class RedBlackTree
<AnyType extends Comparable<? super AnyType>>
 {
…
    private void printTree( RedBlackNode<AnyType> t ) {…}
    private final int
    compare( AnyType item, RedBlackNode<AnyType> t ) {…}
    private void handleReorient( AnyType item ){…}
    private RedBlackNode<AnyType>
    rotate( AnyType item, RedBlackNode<AnyType> parent ) { …}
    private static <AnyType>   RedBlackNode<AnyType>
    rotateWith_LeftChild(RedBlackNode<AnyType> k2 ) { …}
    private static <AnyType> RedBlackNode<AnyType>
    rotateWithRightChild( RedBlackNode<AnyType> kl ) {…}
}
```

Interface Interno

Árvore Red-Black

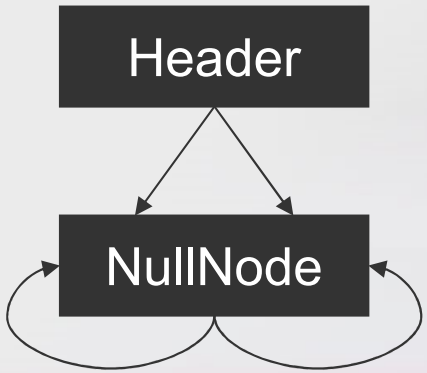
- Construção do NullNode

```
nullNode = new RedBlackNode<AnyType>(null);
nullNode.left = nullNode.right = nullNode;
```

- Construção do Header

```
header = new RedBlackNode<AnyType>( null );
header.left = header.right = nullNode;
```

Árvore vazia =

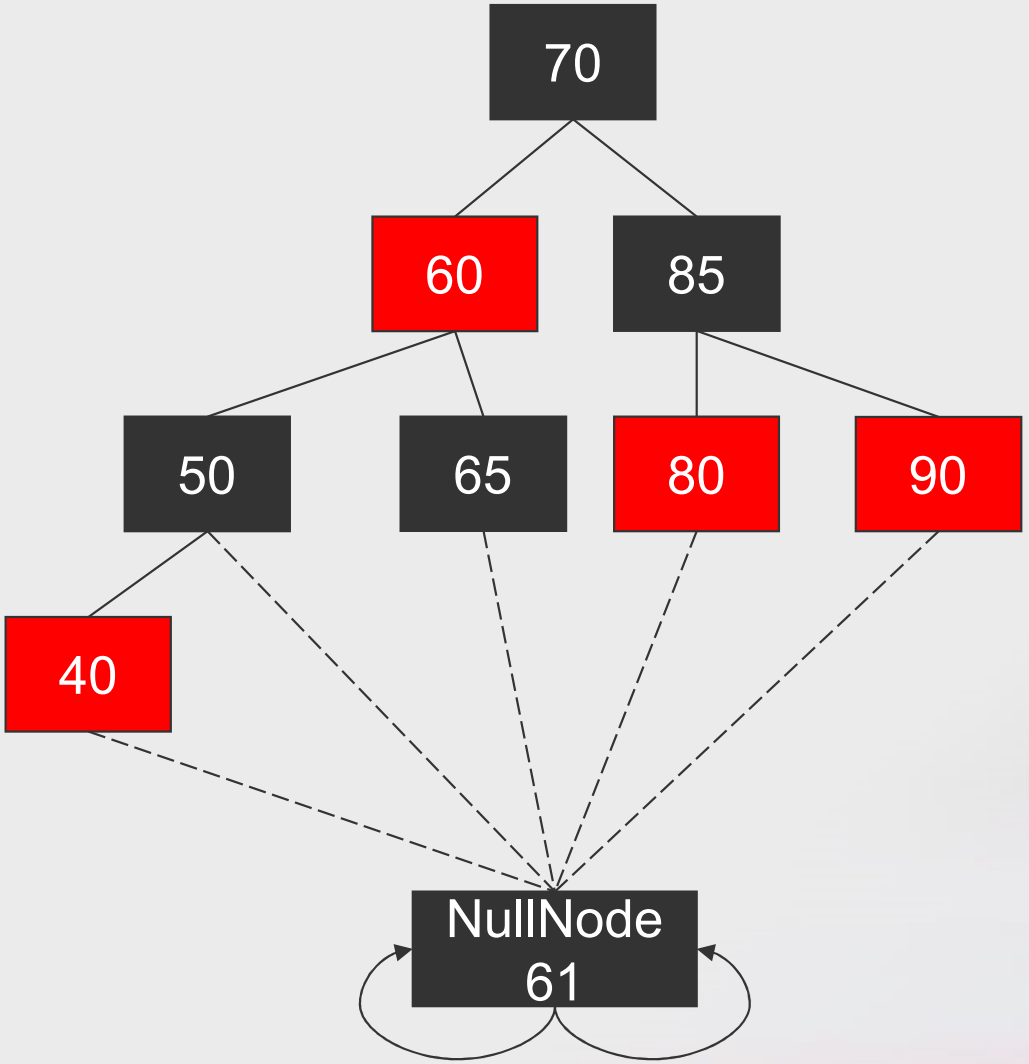


Árvore Red-Black

- Pesquisa com sentinelas
  - No inicio da pesquisa, o nodo nulo é inicializado com o valor a pesquisar!
    - Desta forma, há a garantia que a pesquisa tem sucesso.
    - Evita-se um teste adicional (i,e: verificação de nulo) em cada nodo atravessado.

Árvore Red-Black

- Pesquisa(61)



Árvore Red-Black

```
public AnyType find( AnyType x )
{
nullNode.element = x;
current = header.right;
for(;;)
{
    if( x.compareTo( current.element ) < 0 )
          current = current.left;
    else
          if( x.compareTo( current.element ) > 0 )
                  current = current.right;
          else
                  if( current != nullNode )
                          return current.element;
                  else
                          return null;
```

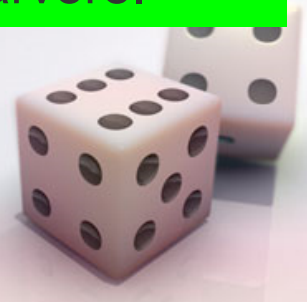
`current.left` e `current.right` nunca serão null, por causa da sentinela

Árvore Red-Black

```
public AnyType find( AnyType x )
{
nullNode.element = x;
current = header.right;
for(;;)
{
    if( x.compareTo( current.element ) < 0 )
          current = current.left;
    else
          if( x.compareTo( current.element ) > 0 )
                  current = current.right;
          else
                  if( current != nullNode )
                          return current.element;
                  else
                          return null;
```

`current.left` e `current.right` nunca serão null, por causa da sentinela

Basta fazer este teste quando o valor pretendido é encontrado para saber se existe ou não na àrvore.



Árvores Red-Black

```
private final int compare
( AnyType item, RedBlackNode<AnyType> t )
{
   if( t == header )
       return 1;
   else
       return item.compareTo( t.element );
 }
```

O `header` deverá ser sempre menor do que qualquer outro nodo.

Caso não esteja em causa o `header`, compara normalmente

Árvore Red-Black

- Rotina de Inserção (interna):
  - **Usa 4 referências**
    - **current** – ponto actual de inserção
    - **parent** – antecessor de current
    - **grand** – antecessor de parent
    - **great** – antecessor de grand

Árvore Red-Black

```
public void insert( AnyType item )
{
 current = parent = grand = header;
 nullNode.element = item;
 while( compare( item, current ) != 0 )
       {
       great = grand; grand = parent; parent = current;
       current = compare( item, current ) < 0 ?  current.left : current.right;
       // Verifica se dois descendentes são vermelhos; ajusta árvore se isso se verificar
       if( current.left.color == RED && current.right.color == RED )
                handleReorient( item );
       }
// Inserção falha se valor já existe
if( current != nullNode )
       throw new DuplicateItemException( item.toString( ) );
current = new RedBlackNode<AnyType>( item, nullNode, nullNode );
// liga a ascendente
if( compare( item, parent ) < 0 )
       parent.left = current;
else
       parent, right = current;
handleReorient( item );
}
```

Por causa de `header`, não é necessário lidar como se fossem casos especiais as inserções próximas da raiz

Árvore Red-Black

```
public void insert( AnyType item )
{
 current = parent = grand = header;
 nullNode.element = item;
 while( compare( item, current ) != 0 )
       {
       great = grand; grand = parent; parent = current;
       current = compare( item, current ) < 0 ?  current.left : current.right;
       // Verifica se dois descendentes são vermelhos; ajusta árvore se isso se verificar
       if( current.left.color == RED && current.right.color == RED )
                handleReorient( item );
       }
// Inserção falha se valor já existe
if( current != nullNode )
       throw new DuplicateItemException( item.toString( ) );
current = new RedBlackNode<AnyType>( item, nullNode, nullNode );
// liga a ascendente
if( compare( item, parent ) < 0 )
       parent.left = current;
else
       parent, right = current;
handleReorient( item );
}
```

Pesquisa ponto de inserção

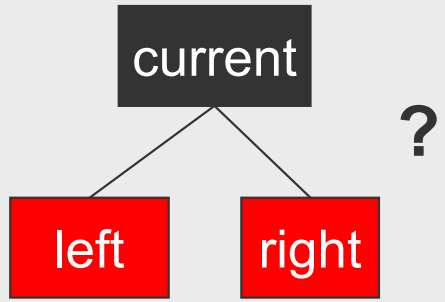
Árvore Red-Black

```
public void insert( AnyType item )
{
 current = parent = grand = header;
 nullNode.element = item;
 while( compare( item, current ) != 0 )
       {
       great = grand; grand = parent; parent = current;
       current = compare( item, current ) < 0 ?  current.left : current.right;
       // Verifica se dois descendentes são vermelhos; ajusta árvore se isso se verificar
       if( current.left.color == RED && current.right.color == RED )
                handleReorient( item );
       }
// Inserção falha se valor já existe
if( current != nullNode )
       throw new DuplicateItemException( item.toString( ) );
current = new RedBlackNode<AnyType>( item, nullNode, nullNode );
// liga a ascendente
if( compare( item, parent ) < 0 )
       parent.left = current;
else
       parent, right = current;
handleReorient( item );
}
```

Vai para subárvore direita ou esquerda conforme o necessário…

Árvore Red-Black

```
public void insert( AnyType item )
{
 current = parent = grand = header;
 nullNode.element = item;
 while( compare( item, current ) != 0 )
       {
       great = grand; grand = parent; parent = current;
       current = compare( item, current ) < 0 ?  current.left : current.right;
       // Verifica se dois descendentes são vermelhos; ajusta árvore se isso se verificar
       if( current.left.color == RED && current.right.color == RED )
                handleReorient( item );
       }
// Inserção falha se valor já existe
if( current != nullNode )
       throw new DuplicateItemException( item.toString( ) );
current = new RedBlackNode<AnyType>( item, nullNode, nullNode );
// liga a ascendente
if( compare( item, parent ) < 0 )
       parent.left = current;
else
       parent, right = current;
handleReorient( item );
}
```



Caso encontre um nodo com dois descentes vermelhos, corrige.

Árvore Red-Black

```
public void insert( AnyType item )
{
 current = parent = grand = header;
 nullNode.element = item;
 while( compare( item, current ) != 0 )
       {
       great = grand; grand = parent; parent = current;
       current = compare( item, current ) < 0 ?  current.left : current.right;
       // Verifica se dois descendentes são vermelhos; ajusta árvore se isso se verificar
       if( current.left.color == RED && current.right.color == RED )
                handleReorient( item );
       }
// Inserção falha se valor já existe
if( current != nullNode )
       throw new DuplicateItemException( item.toString( ) );
current = new RedBlackNode<AnyType>( item, nullNode, nullNode );
// liga a ascendente
if( compare( item, parent ) < 0 )
       parent.left = current;
else
       parent, right = current;
handleReorient( item );
}
```

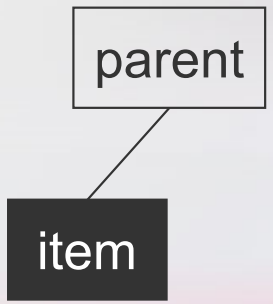
Caso o novo elemento já exista, gera excepção!

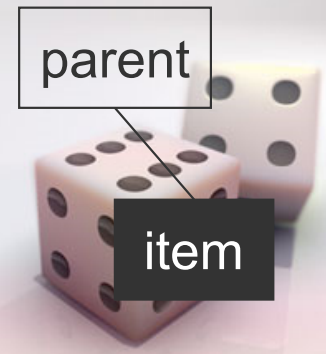
Árvore Red-Black

```
public void insert( AnyType item )
{
 current = parent = grand = header;
 nullNode.element = item;
 while( compare( item, current ) != 0 )
       {
       great = grand; grand = parent; parent = current;
       current = compare( item, current ) < 0 ?  current.left : current.right;
       // Verifica se dois descendentes são vermelhos; ajusta árvore se isso se verificar
       if( current.left.color == RED && current.right.color == RED )
                handleReorient( item );
       }
// Inserção falha se valor já existe
if( current != nullNode )
       throw new DuplicateItemException( item.toString( ) );
current = new RedBlackNode<AnyType>( item, nullNode, nullNode );
// liga a ascendente
if( compare( item, parent ) < 0 )
       parent.left = current;
else
       parent, right = current;
handleReorient( item );
}
```

Insere novo elemento em `left` ou `right` de antecessor, conforme seja maior ou menor do que este?

Item<parent? item>parent?



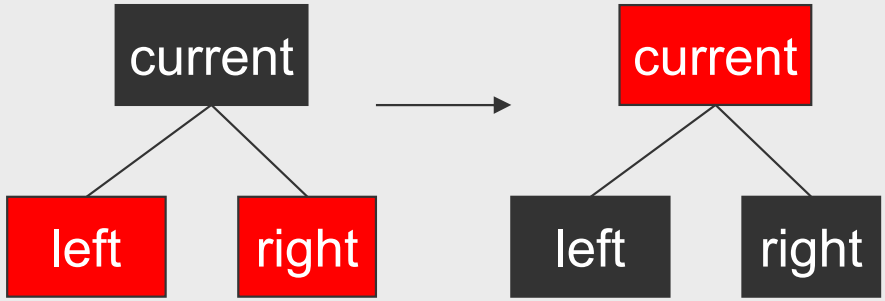


Árvore Red-Black

```
private void handleReorient( AnyType item )
{
      // Inverte cores
      current.color = RED;
      current.left.color = BLACK;
      current.right.color = BLACK;
      if( parent.color == RED ) // é necessário rodar
                {
                grand.color = RED;
                if( ( compare( item, grand ) < 0 ) != ( compare( item, parent ) < 0 ) )
                          parent = rotate( item, grand ); // inicia dupla rotação
                current = rotate( item, great );
                current.color = BLACK;
                }
      header.right.color = BLACK; // torna raiz preta
}
```

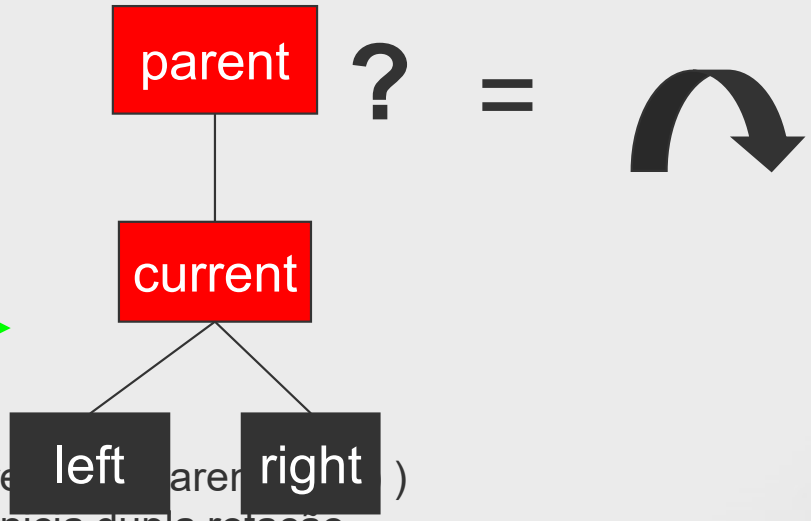
Árvore Red-Black

```
private void handleReorient( AnyType item )
{
    // Inverte cores
    current.color = RED;
    current.left.color = BLACK;
    current.right.color = BLACK;
    if( parent.color == RED ) // é necessário rodar
            {
            grand.color = RED;
            if( ( compare( item, grand ) < 0 ) != ( compare( item, parent ) < 0 ) )
                    parent = rotate( item, grand ); // inicia dupla rotação
            current = rotate( item, great );
            current.color = BLACK;
            }
    header.right.color = BLACK; // torna raiz preta
}
```



Árvore Red-Black

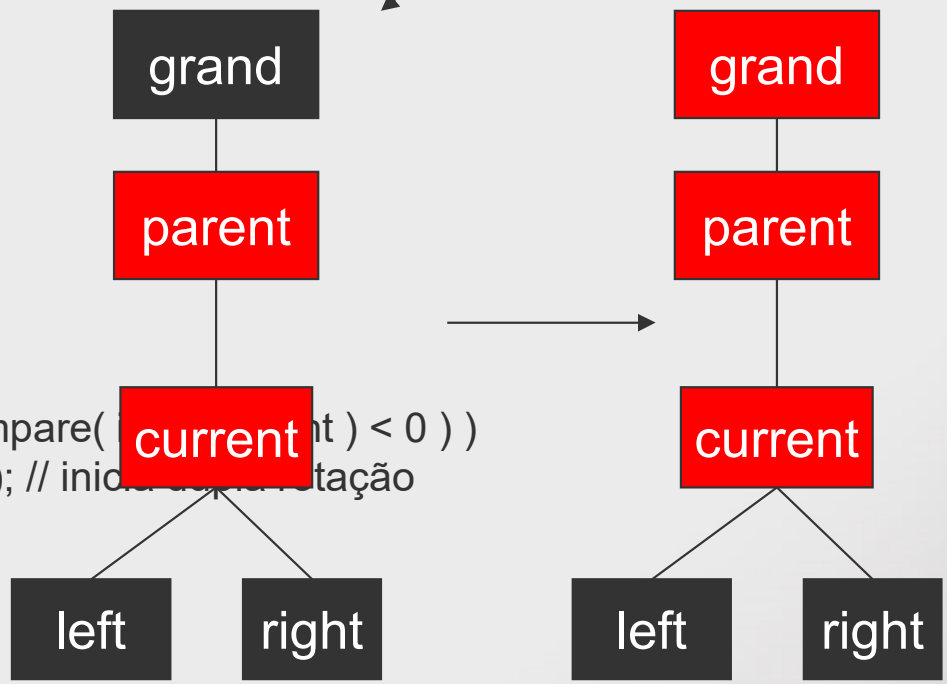
```
private void handleReorient( AnyType item )
{
      // Inverte cores
      current.color = RED;
      current.left.color = BLACK;
      current.right.color = BLACK;
      if( parent.color == RED ) // é necessário rodar
                {
                grand.color = RED;
                if( ( compare( item, grand ) < 0 ) != ( compare
                          parent = rotate( item, grand ); // inicia dupla rotação
                current = rotate( item, great );
                current.color = BLACK;
                }
      header.right.color = BLACK; // torna raiz preta
}
```



Árvore Red-Black

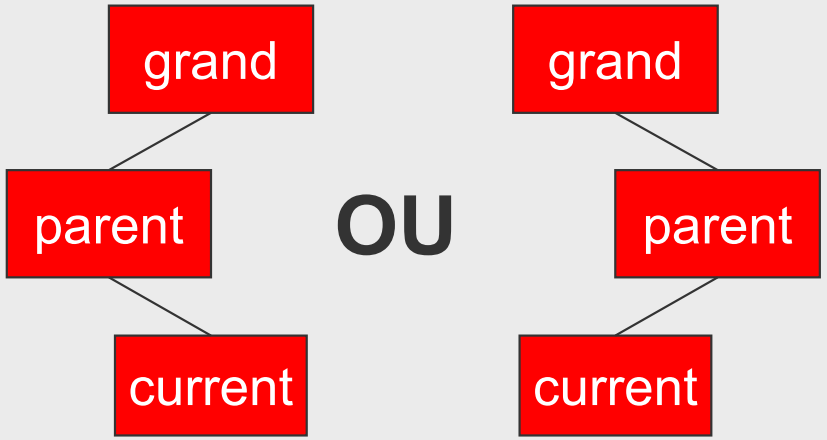
```
private void handleReorient( AnyType item )
{
      // Inverte cores
      current.color = RED;
      current.left.color = BLACK;
      current.right.color = BLACK;
      if( parent.color == RED ) // é necessário rodar
            {
            grand.color = RED;
            if( ( compare( item, grand ) < 0 ) != ( compare( item, parent ) < 0 ) )
                  parent = rotate( item, grand ); // inicia dupla rotação
            current = rotate( item, great );
            current.color = BLACK;
            }
      header.right.color = BLACK; // torna raiz preta
}
```

Será sempre preto (porquê)?



Árvore Red-Black

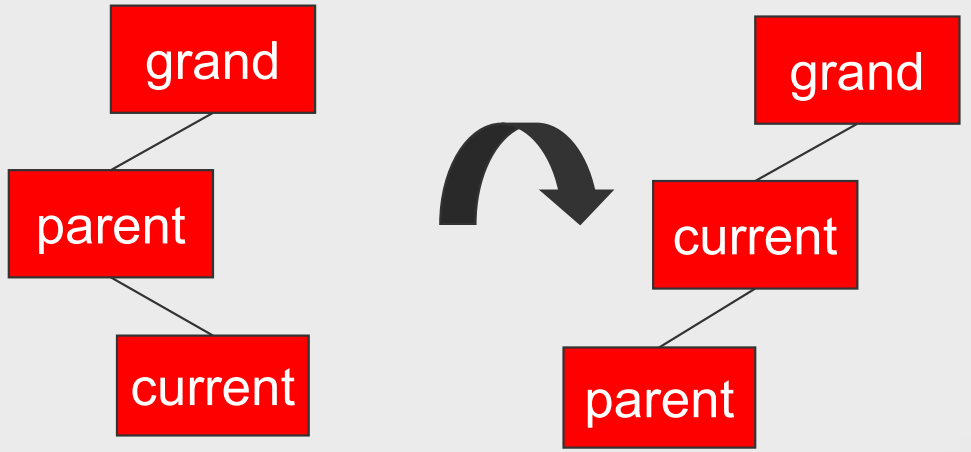
```
private void handleReorient( AnyType item )
{
      // Inverte cores
      current.color = RED;
      current.left.color = BLACK;
      current.right.color = BLACK;
      if( parent.color == RED ) // é necessário rodar
                {
                grand.color = RED;
                if( ( compare( item, grand ) < 0 ) != ( compare( item, parent ) < 0 ) )
                          parent = rotate( item, grand ); // inicia dupla rotação
                current = rotate( item, great );
                current.color = BLACK;
                }
      header.right.color = BLACK; // torna raiz preta
}
```



verifica se é necessária uma dupla rotação!

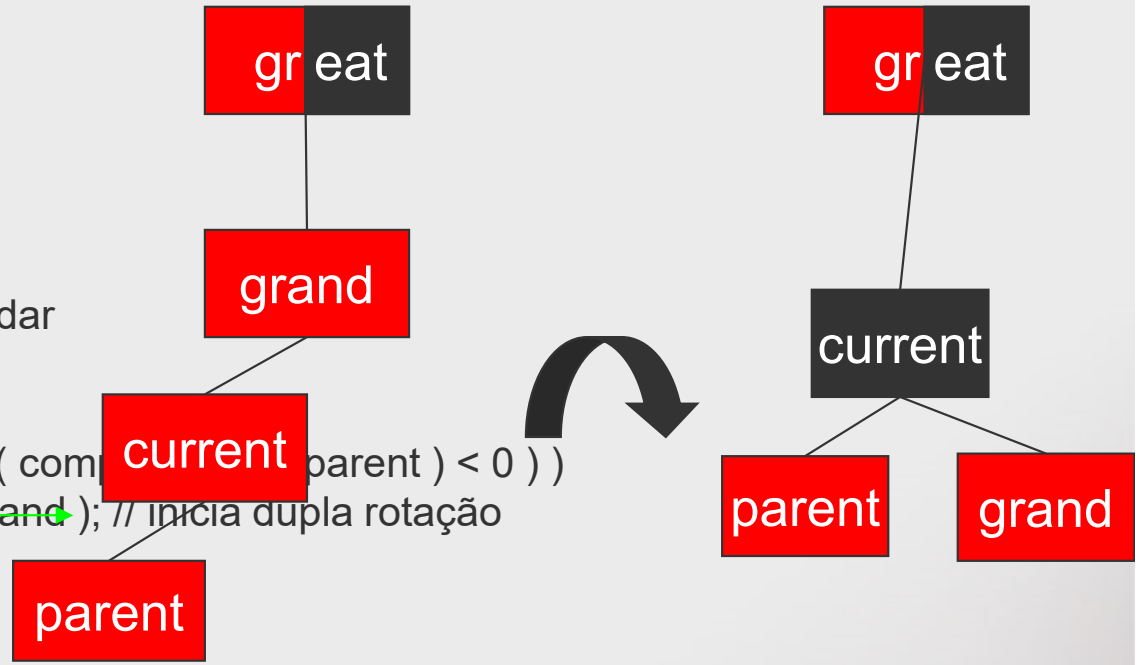
Árvore Red-Black

```
private void handleReorient( AnyType item )
{
      // Inverte cores
      current.color = RED;
      current.left.color = BLACK;
      current.right.color = BLACK;
      if( parent.color == RED ) // é necessário rodar
                {
                grand.color = RED;
                if( ( compare( item, grand ) < 0 ) != ( compare( item, parent ) < 0 ) )
                          parent = rotate( item, grand ); // inicia dupla rotação
                current = rotate( item, great );
                current.color = BLACK;
                }
      header.right.color = BLACK; // torna raiz preta
}
```



Árvore Red-Black

```
private void handleReorient( AnyType item )
{
      // Inverte cores
      current.color = RED;
      current.left.color = BLACK;
      current.right.color = BLACK;
      if( parent.color == RED ) // é necessário rodar
            {
            grand.color = RED;
            if( ( compare( item, grand ) < 0 ) != ( com                parent ) < 0 ) )
                  parent = rotate( item, grand ); // inicia dupla rotação
            current = rotate( item, great );
            current.color = BLACK;
            }
      header.right.color = BLACK; // torna raiz preta
}
```

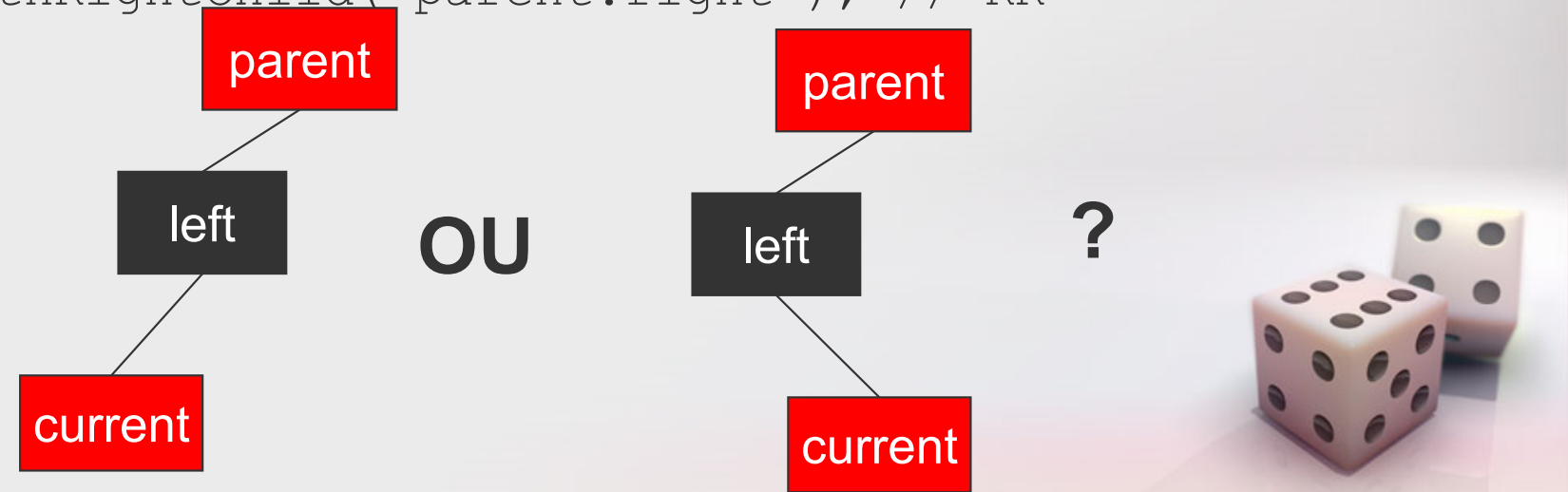


Árvore Red-Black

```
private RedBlackNode<AnyType> rotate
( AnyType item,
RedBlackNode<AnyType> parent )
{
if( compare( item, parent ) < 0 )
   return parent.left = compare( item, parent.left )<0 ?
   rotateWithLeftChild( parent.left ) : // LL
   rotateWithRightChild( parent.left ) ; // LR
else
   return parent.right = compare( item, parent.right )< 0 ?
   rotateWithLeftChild( parent.right ) : // RL
   rotateWithRightChild( parent.right ); // RR
}
```

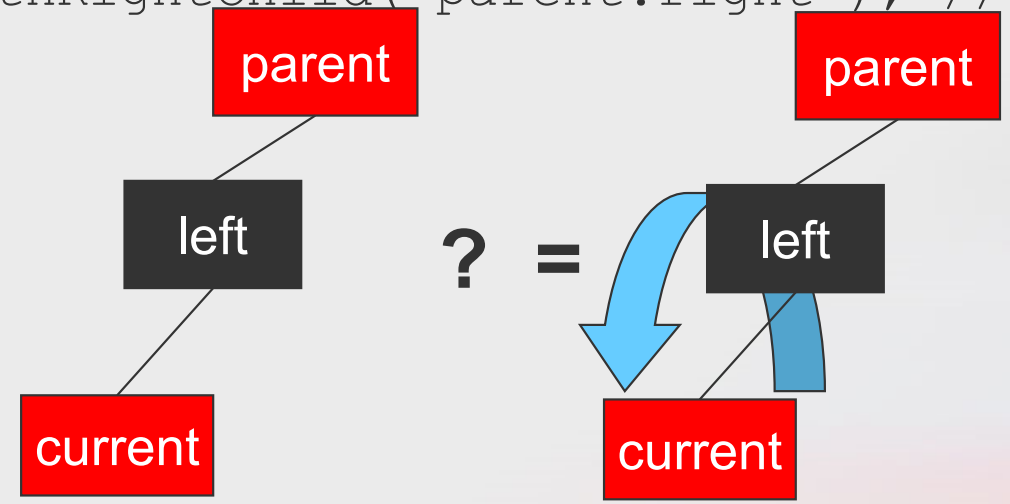
Árvore Red-Black

```
private RedBlackNode<AnyType> rotate
( AnyType item,
RedBlackNode<AnyType> parent )
{
if( compare( item, parent ) < 0 )
   return parent.left = compare( item, parent.left )<0 ?
   rotateWithLeftChild( parent.left ) : // LL
   rotateWithRightChild( parent.left ) ; // LR
else
   return parent.right = compare( item, parent.right )< 0 ?
   rotateWithLeftChild( parent.right ) : // RL
   rotateWithRightChild( parent.right ); // RR
}
```



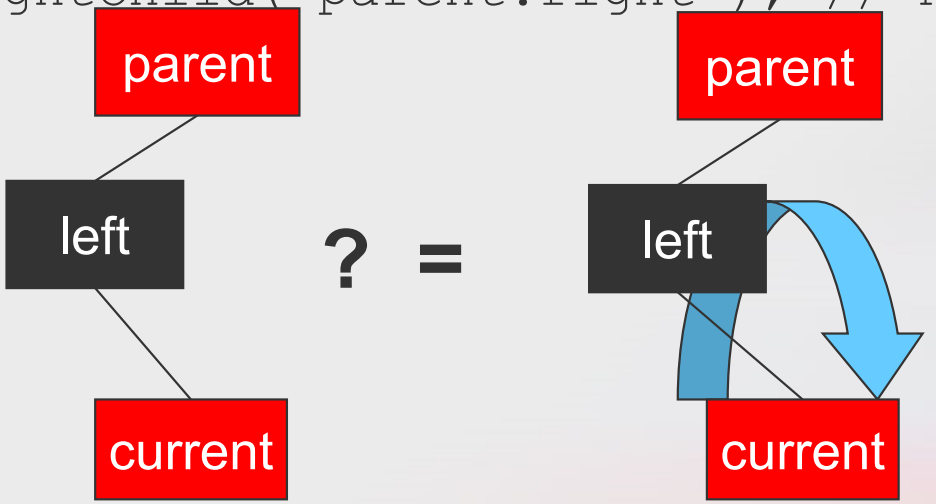
Árvore Red-Black

```
private RedBlackNode<AnyType> rotate
( AnyType item,
RedBlackNode<AnyType> parent )
{
if( compare( item, parent ) < 0 )
   return parent.left = compare( item, parent.left )<0 ?
   rotateWithLeftChild( parent.left ) : // LL
   rotateWithRightChild( parent.left ) ; // LR
else
   return parent.right = compare( item, parent.right )< 0 ?
   rotateWithLeftChild( parent.right ) : // RL
   rotateWithRightChild( parent.right ); // RR
}
```



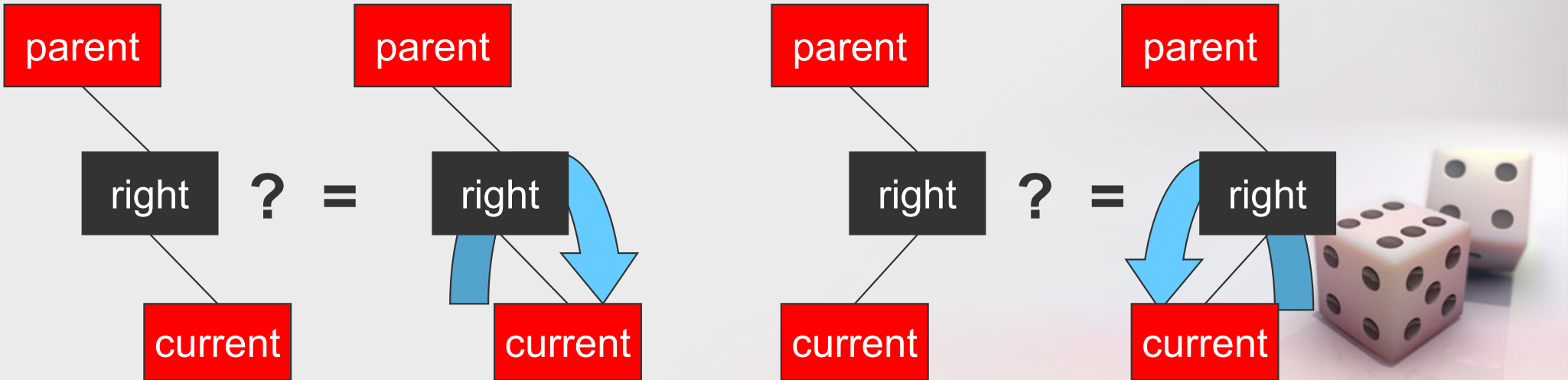
Árvore Red-Black

```
private RedBlackNode<AnyType> rotate
( AnyType item,
RedBlackNode<AnyType> parent )
{
if( compare( item, parent ) < 0 )
   return parent.left = compare( item, parent.left )<0 ?
   rotateWithLeftChild( parent.left ) : // LL
   rotateWithRightChild( parent.left ) ; // LR
else
   return parent.right = compare( item, parent.right )< 0 ?
   rotateWithLeftChild( parent.right ) : // RL
   rotateWithRightChild( parent.right ); // RR
}
```



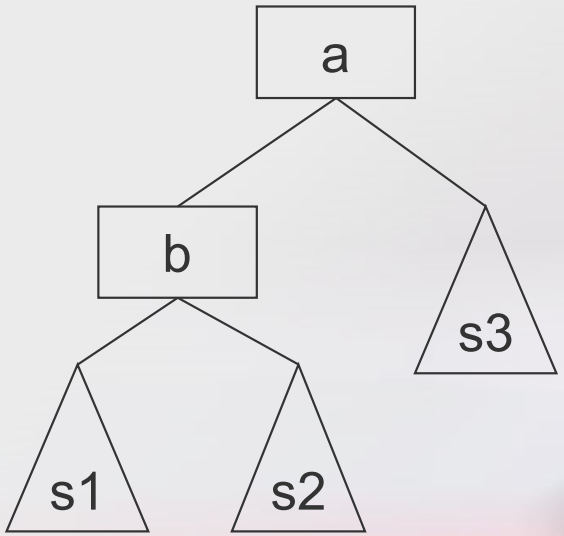
Árvore Red-Black

```
private RedBlackNode<AnyType> rotate
( AnyType item,
RedBlackNode<AnyType> parent )
{
if( compare( item, parent ) < 0 )
   return parent.left = compare( item, parent.left )<0 ?
   rotateWithLeftChild( parent.left ) : // LL
   rotateWithRightChild( parent.left ) ; // LR
else
   return parent.right = compare( item, parent.right )< 0 ?
   rotateWithLeftChild( parent.right ) : // RL
   rotateWithRightChild( parent.right ); // RR
```



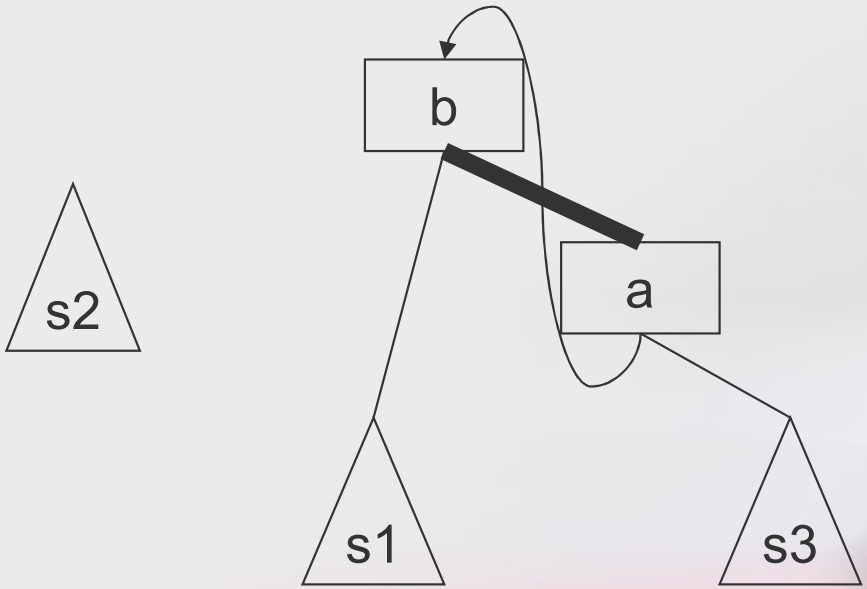
Árvore Red-Black

```
RedBlackNode<AnyType>
rotateWithlLeftChild( RedBlackNode<AnyType> a)
{
RedBlackNode<AnyType> b=a.left;
RedBlackNode<AnyType> s2=b.right;
b.right=a;
a.left=s2;
return b;
}
```



Árvore Red-Black

```
RedBlackNode<AnyType>
rotateWithlLeftChild( RedBlackNode<AnyType> a)
{
RedBlackNode<AnyType> b=a.left;
RedBlackNode<AnyType> s2=b.right;
b.right=a;
a.left=s2;
return b;
}
```



Árvore Red-Black

```
RedBlackNode<AnyType>
rotateWithlLeftChild( RedBlackNode<AnyType> a)
{
RedBlackNode<AnyType> b=a.left;
RedBlackNode<AnyType> s2=b.right;
b.right=a;
a.left=s2;
return b;
}
```