NOVA SCHOOL OF SCIENCE & TECHNOLOGY

NOVALINCS LABORATORY FOR COMPUTER SCIENCE AND INFORMATICS

ALGORITMOS E ESTRUTURAS DE DADOS 2023/2024
ÁRVORES BINÁRIAS PERFEITAMENTE EQUILIBRADAS E EQUILIBRADAS

Armanda Rodrigues

8 de novembro de 2023

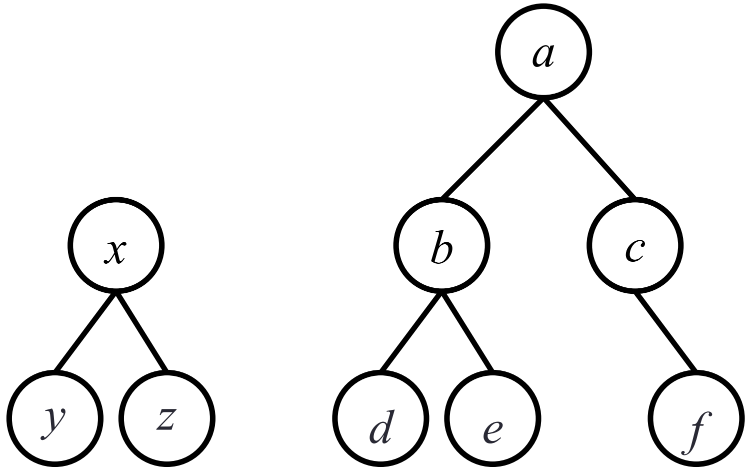
Árvore Binária Perfeitamente Equilibrada

- Uma árvore binária é Perfeitamente Equilibrada se, todo o nó X da mesma verifica a seguinte propriedade:

$$\left| \begin{array}{c} \text{NúmeroDeNós (subárvore esquerda de } X \text{)} \\ - \\ \text{NúmeroDeNós (subárvore direita de } X \text{)} \end{array} \right| \leq 1$$

Altura máxima de uma Arvore Binária Perfeitamente Equilibrada com n nós:

$$H(n) = \lceil \log(n+1) \rceil$$

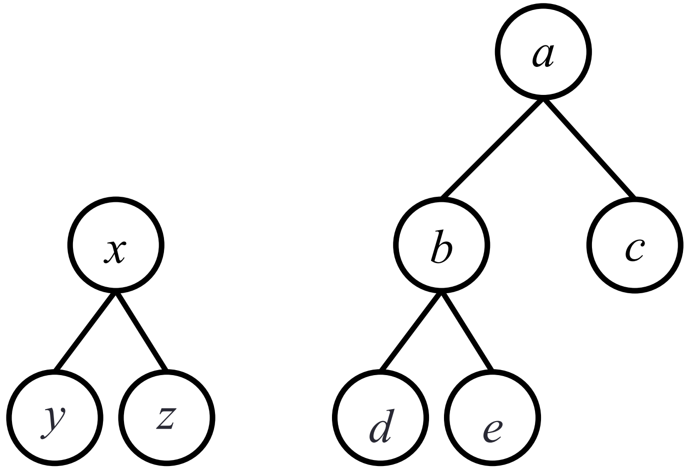


Árvore Binária Equilibrada

- Uma árvore binária é Equilibrada se, todo o nó X da mesma verifica a seguinte propriedade:

$$\left|\begin{array}{c}\text{Altura (subárvore esquerda de } X \text{)} \\ - \\ \text{Altura (subárvore direita de } X \text{)}\end{array}\right| \leq 1$$

Proposição: Uma árvore binária perfeitamente equilibrada é uma árvore binária equilibrada.

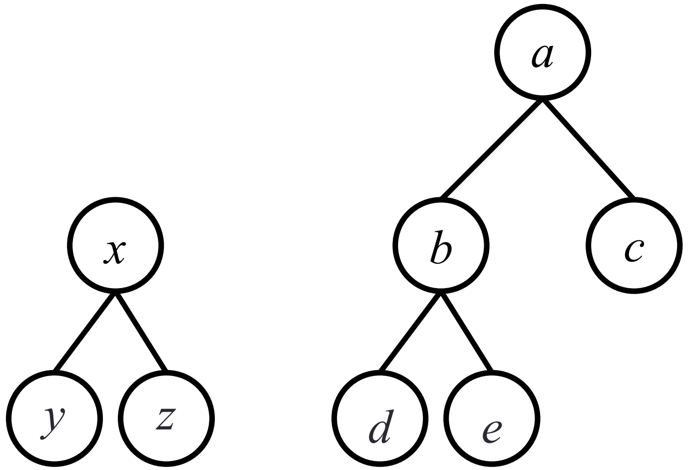


Árvore Binária Equilibrada (2)

- Altura Máxima de uma Árvore Binária Equilibrada com $\boldsymbol{n}$ nós:

$$h \leq \frac{\log n + \log\sqrt{5}}{\log \phi} - 2 \approx \boxed{1.44 \log n - 0.33}$$

Com: $\phi = \dfrac{1+\sqrt{5}}{2} \approx 1.6180$



ALGORITMOS E ESTRUTURAS DE DADOS 2023/2024
ÁRVORES AVL

Armanda Rodrigues

8 de novembro de 2023

Árvore AVL (Adelson-Velskii and Landis – 1962)

- Uma árvore AVL é uma árvore binária de pesquisa equilibrada
  - A diferença entre as alturas das subárvores esquerda e direita deve ser no máximo 1
  - A verificação desta condição deverá ser de fácil manutenção

- Uma forma de implementar a condição é guardar, em cada nó, informação sobre a diferença existente entre as alturas das subárvores do nó.

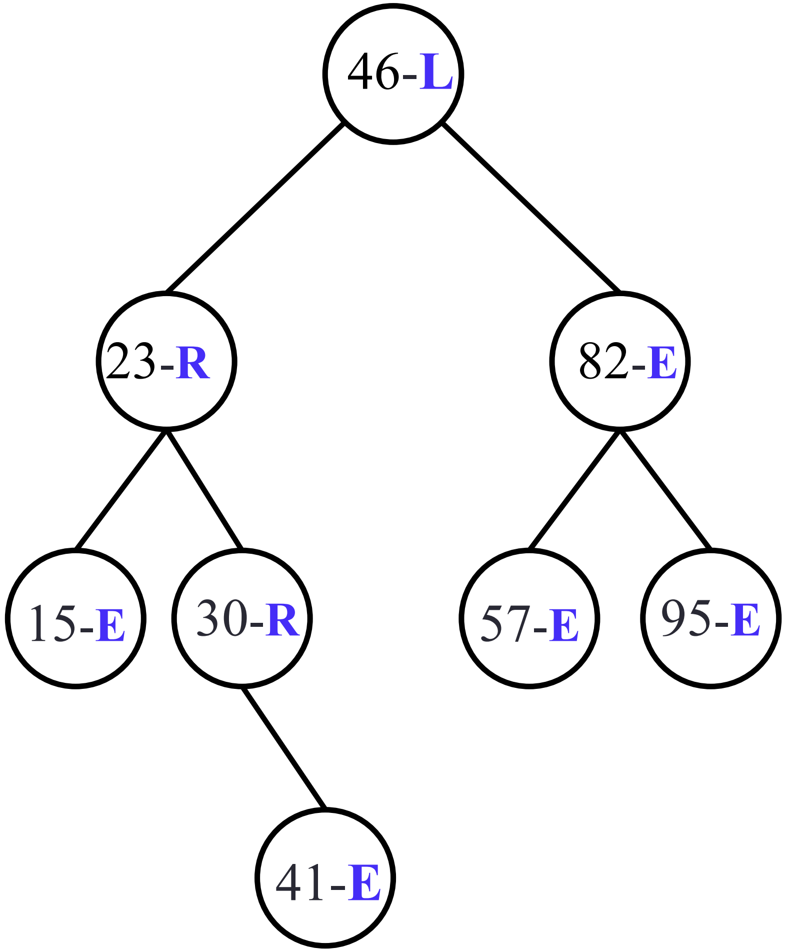
Árvore AVL (Adelson-Velskii and Landis – 1962)

Notação:

L – a subárvore esquerda tem maior altura

E – as duas subárvores têm altura igual

R - a subárvore direita tem maior altura



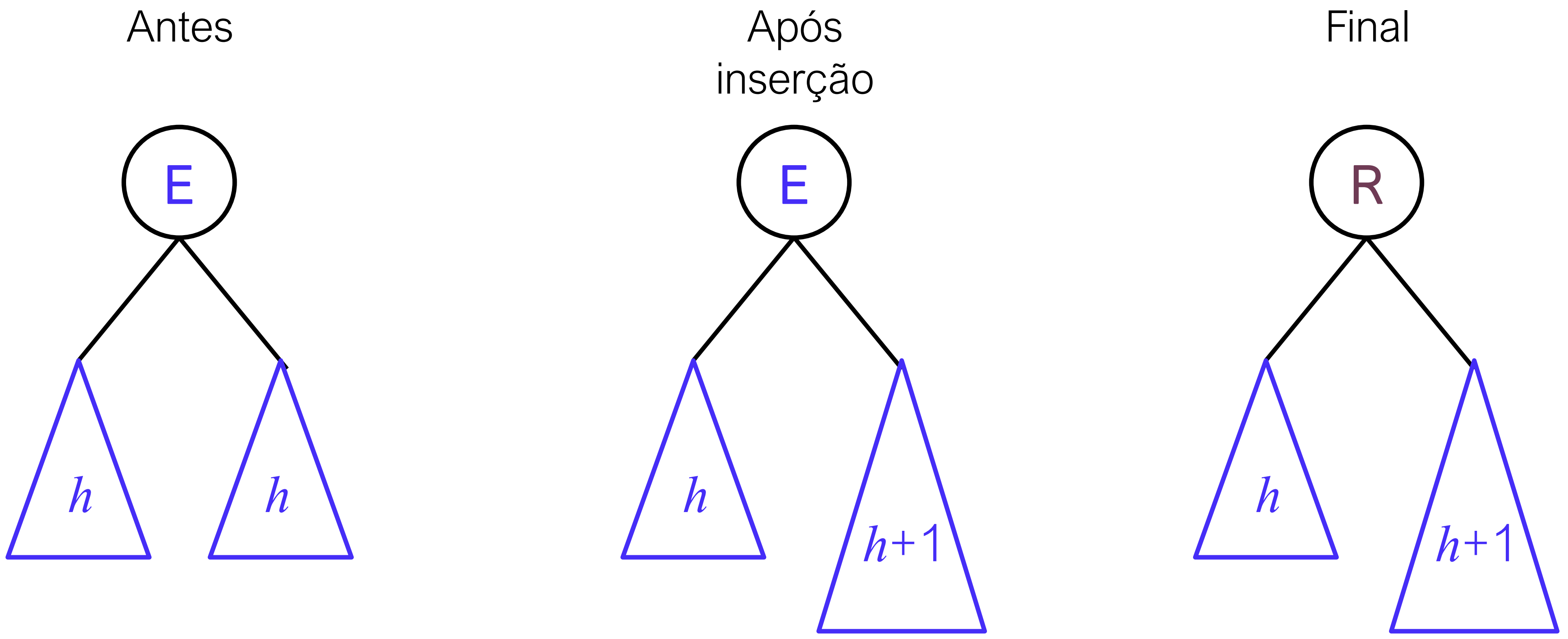
Inserção e Remoção

- Insere-se qualquer entrada como folha.
- Reduz-se qualquer remoção a uma remoção de um nó com, no máximo, um filho.
- Após a inserção e a remoção do nó, reorganiza-se a árvore percorrendo, no pior caso, todos os nós do caminho
  - Que começa no pai do nó inserido ou removido e
  - Que termina na raiz da árvore,

  alterando o fator de equilíbrio do nó ou efetuando rotações, simples ou duplas, à esquerda ou à direita.

Inserção

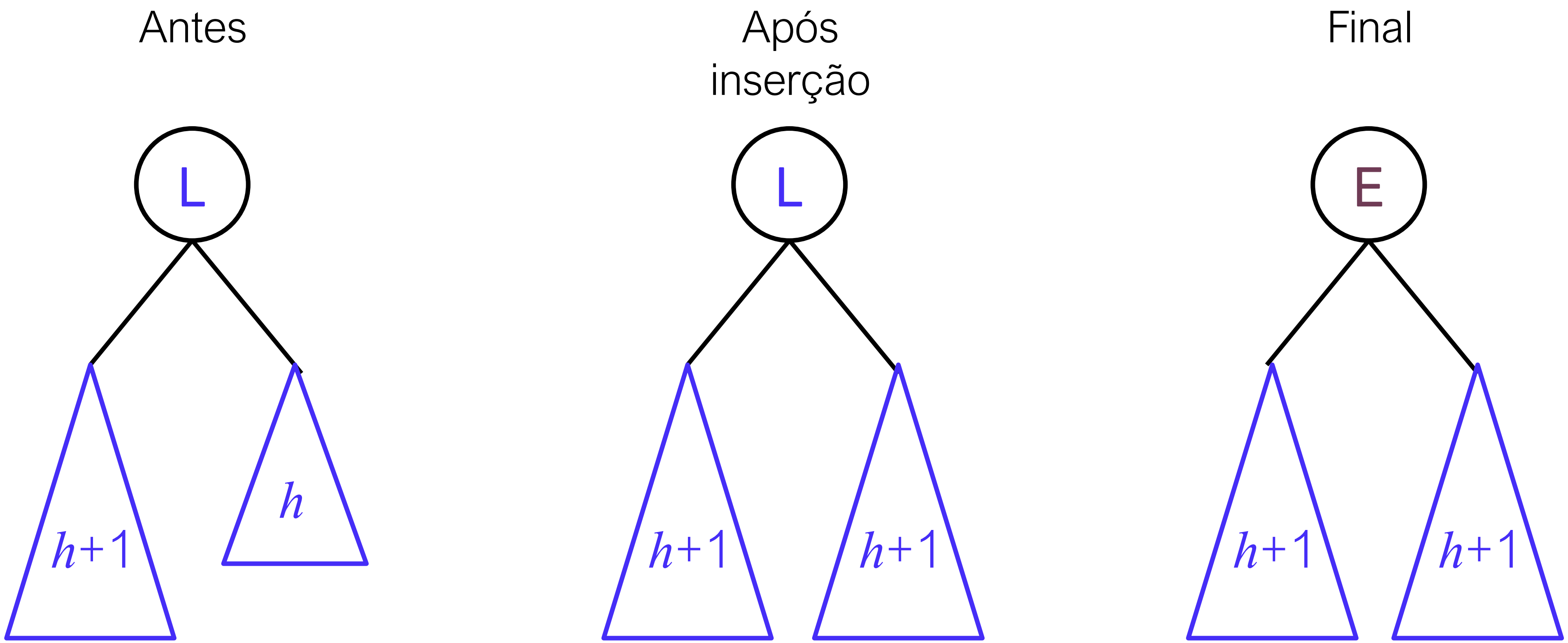
Nó E - Subárvore Direita Cresceu



A árvore cresceu

Inserção

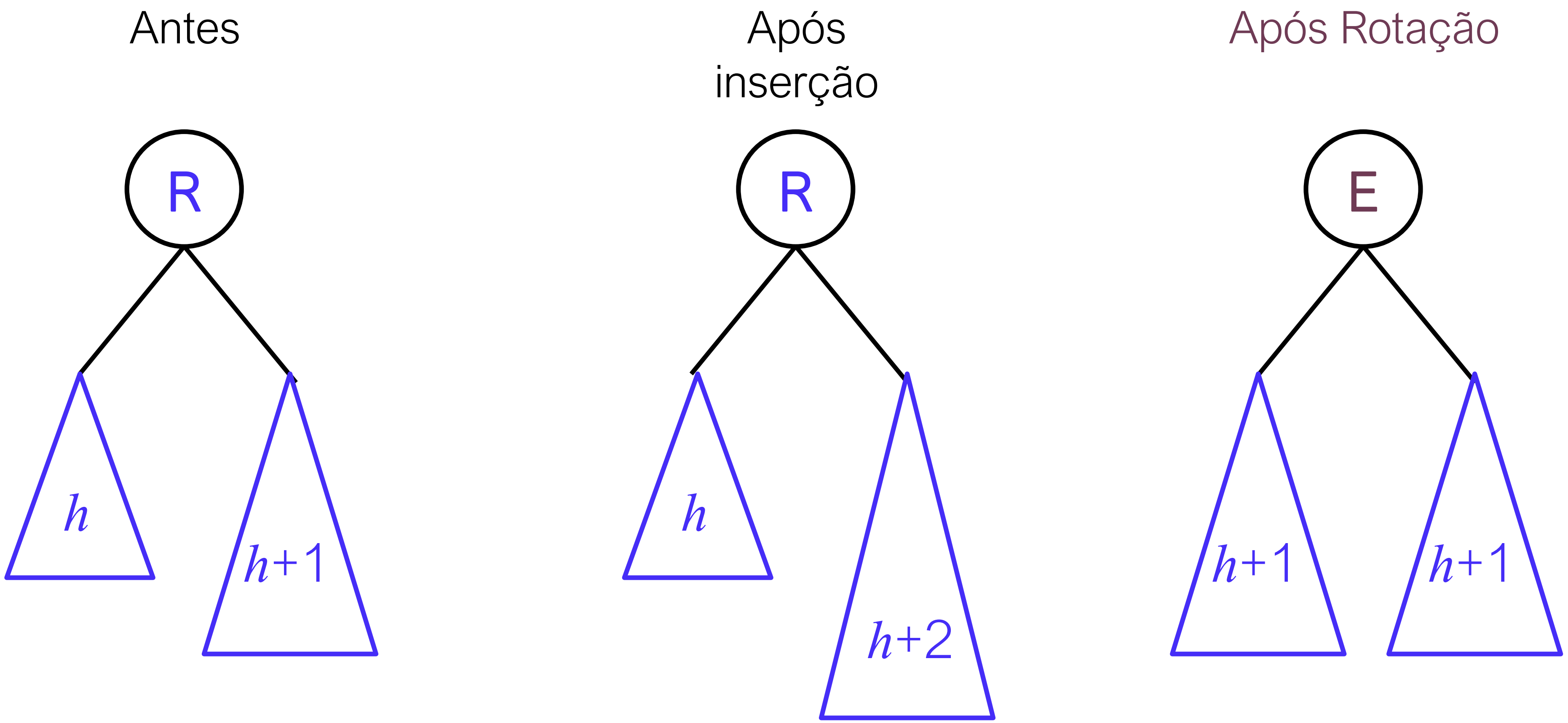
Nó L - Subárvore Direita Cresceu



A árvore não cresceu

Inserção

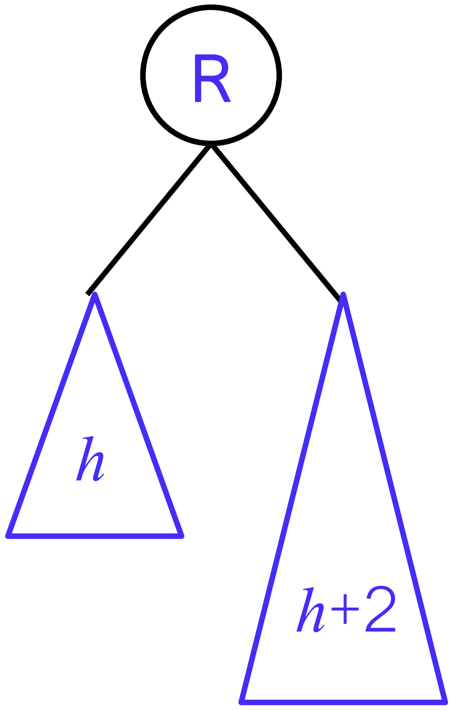
Nó R - Subárvore Direita Cresceu



A árvore não cresceu

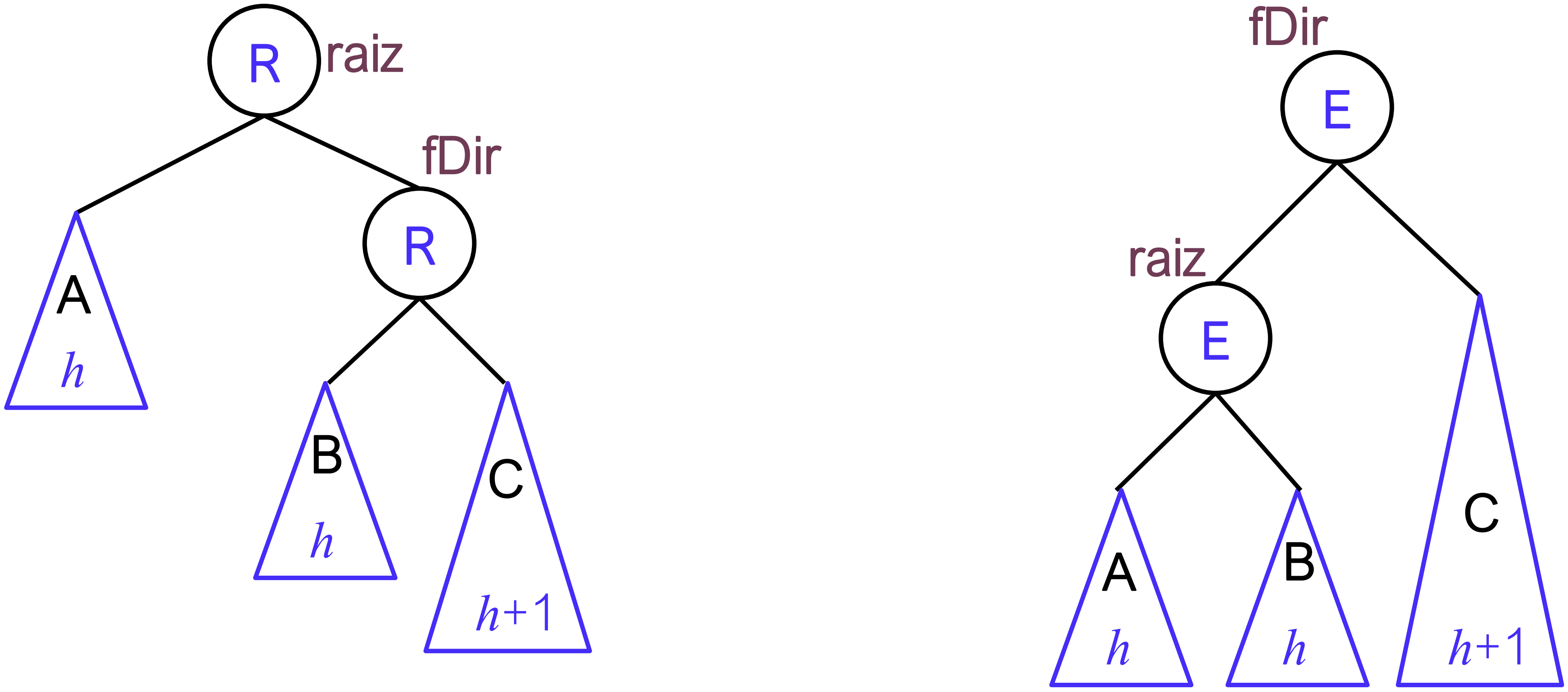
Inserção

Nó R - Subárvore Direita Cresceu



- Esta situação poderá ocorrer de 3 formas diferentes:
  - A raiz da subárvore direita é R
  - A raiz da subárvore direita é L
  - A raiz da subárvore direita é E (impossível)

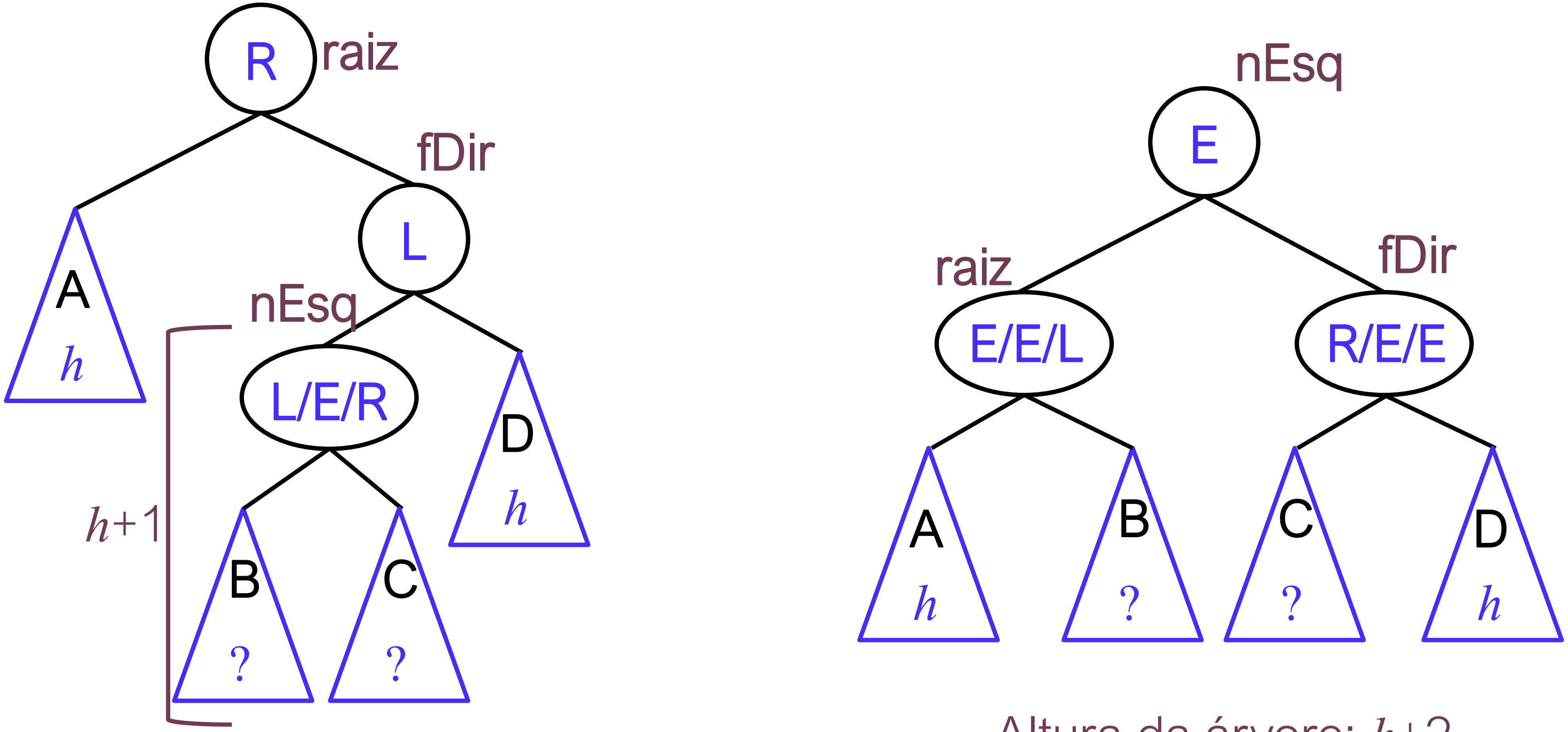
Inserção R-R / Rotação simples à direita



Ordenação: A raiz B fDir C

Altura da árvore: h+2

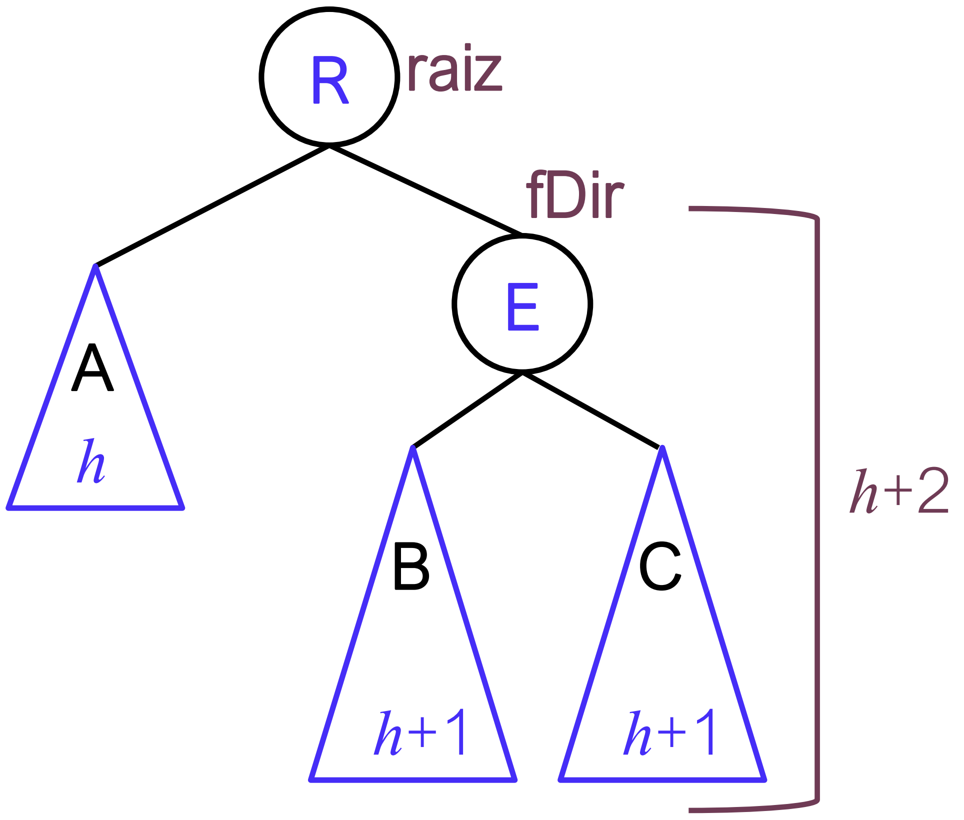
Inserção R-L / Rotação dupla à direita



Ordenação: A raiz B nEsq C fDir D

Altura da árvore: h+2

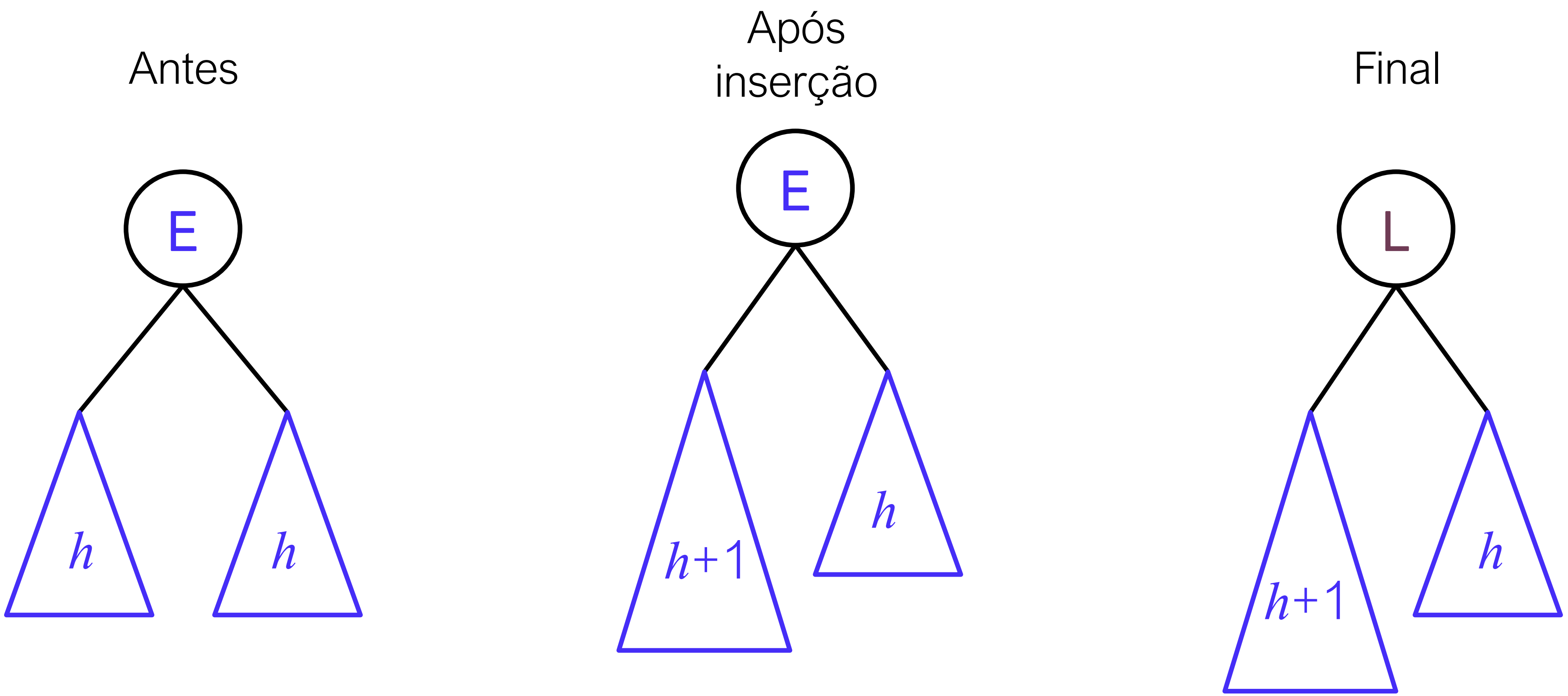
Inserção R-E



Impossível: Inseriu-se um nó

Inserção

Nó E - Subárvore Esquerda Cresceu

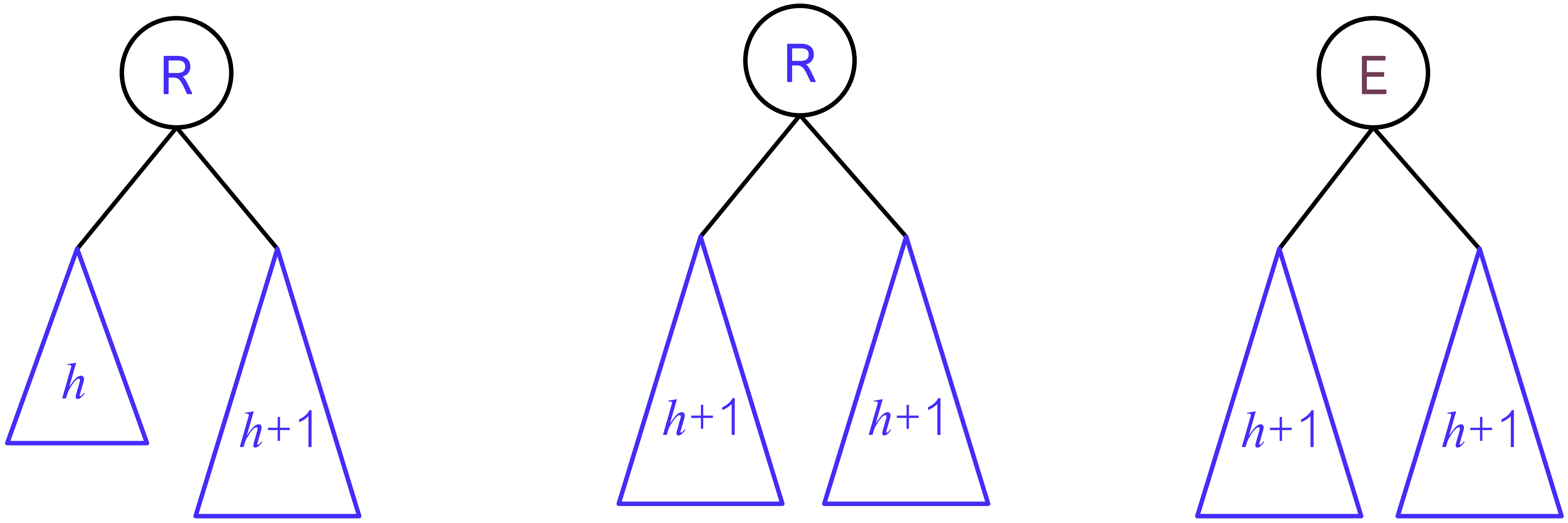


A árvore cresceu

Inserção

Nó R - Subárvore Esquerda Cresceu

Antes | Após inserção | Final

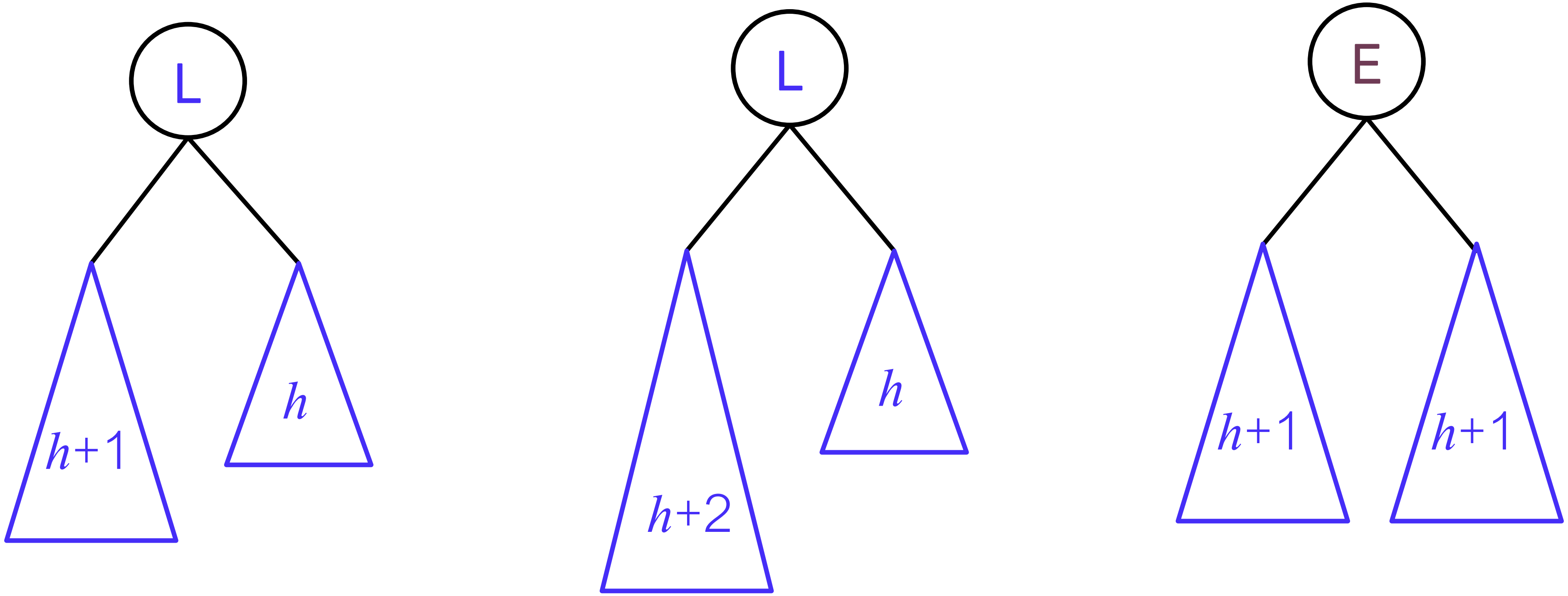


A árvore não cresceu

Inserção

Nó L - Subárvore Esquerda Cresceu

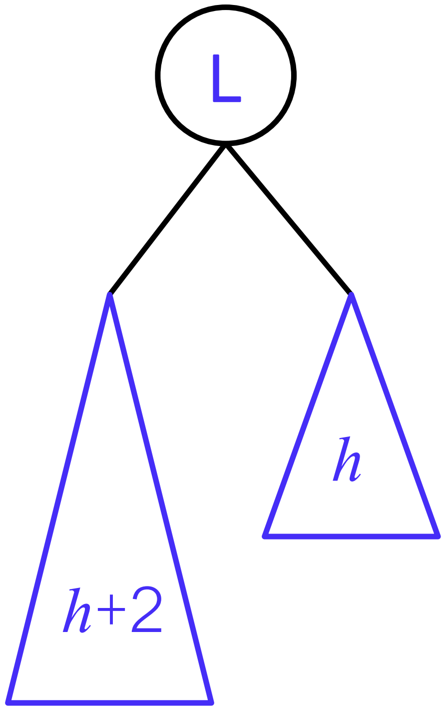
Antes | Após inserção | Após Rotação



A árvore não cresceu

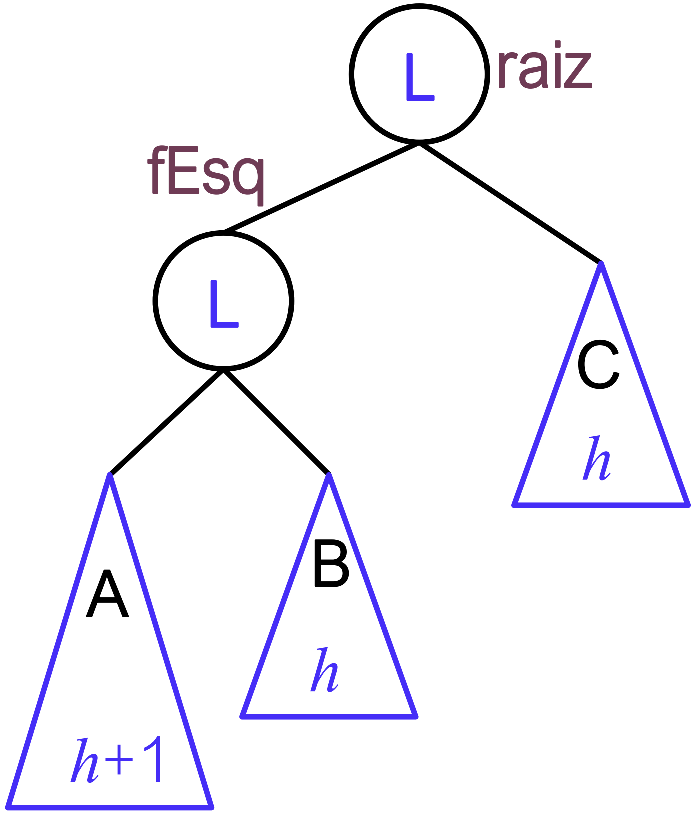
Inserção

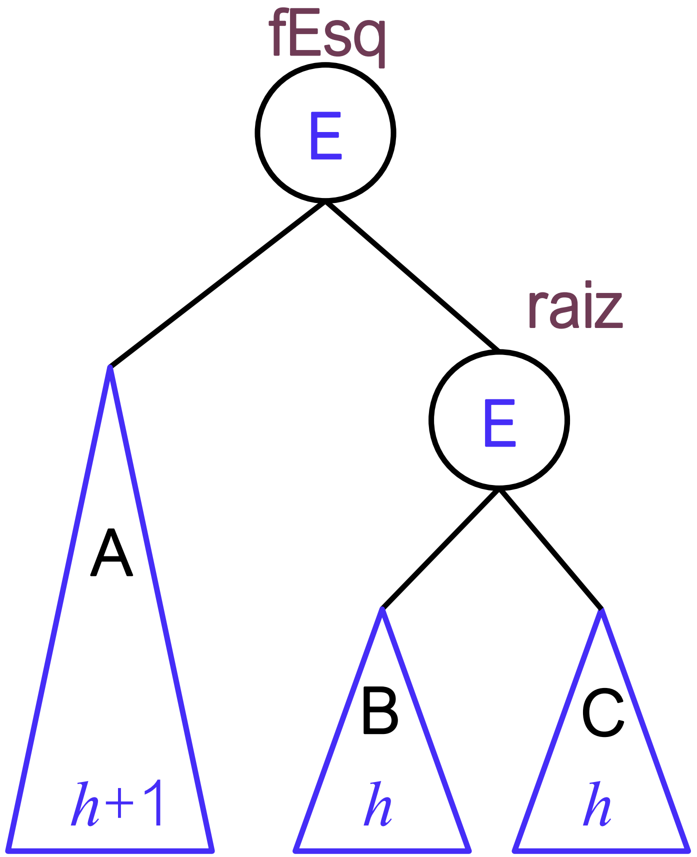
Nó L - Subárvore Esquerda Cresceu



- Esta situação poderá ocorrer de 3 formas diferentes:
  - A raiz da subárvore esquerda é L
  - A raiz da subárvore esquerda é R
  - A raiz da subárvore esquerda é E (impossível)

Inserção L- L / Rotação simples à esquerda

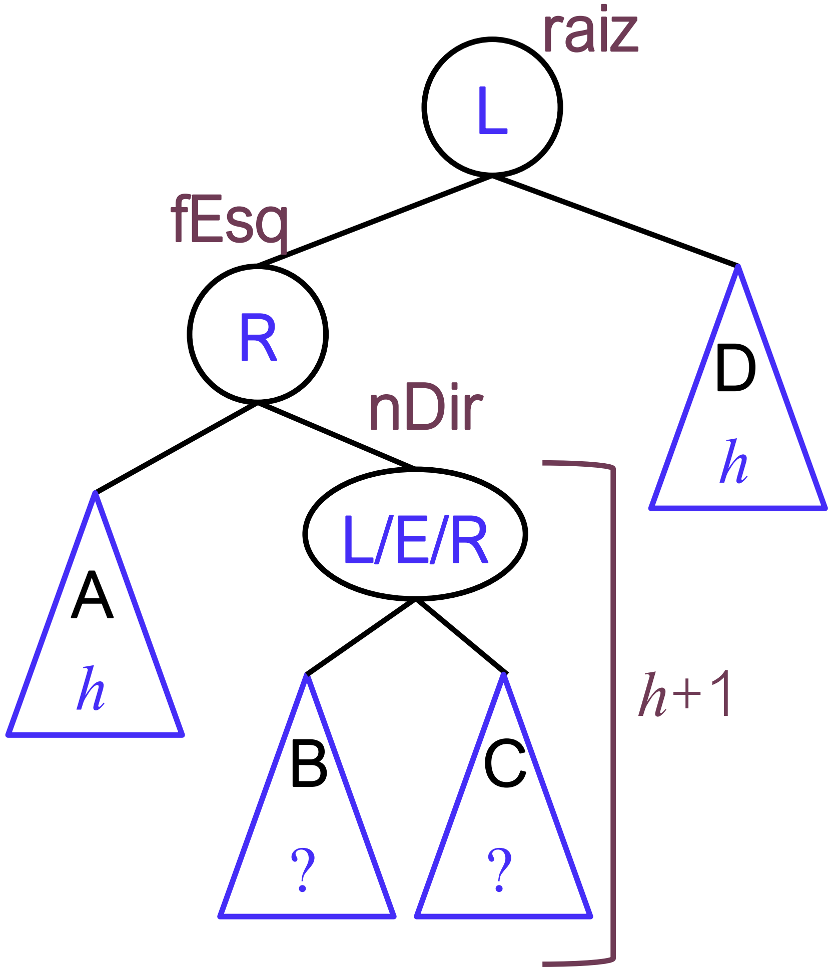


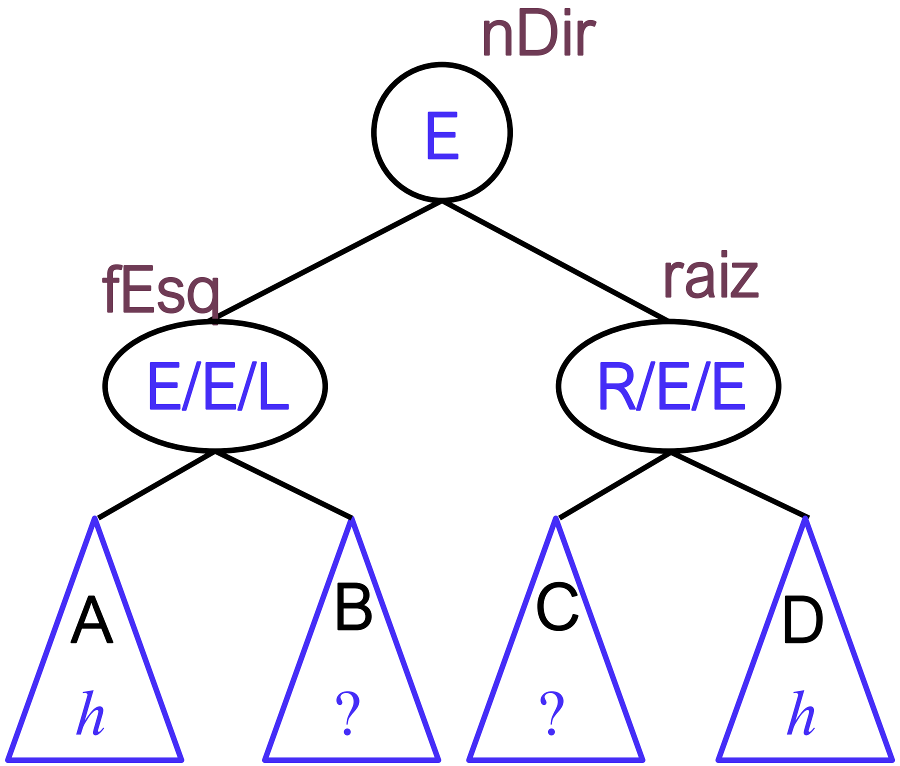


<u>Ordenação:</u> A fEsq B raiz C

Altura da árvore: h+2

Inserção L-R / Rotação dupla à esquerda

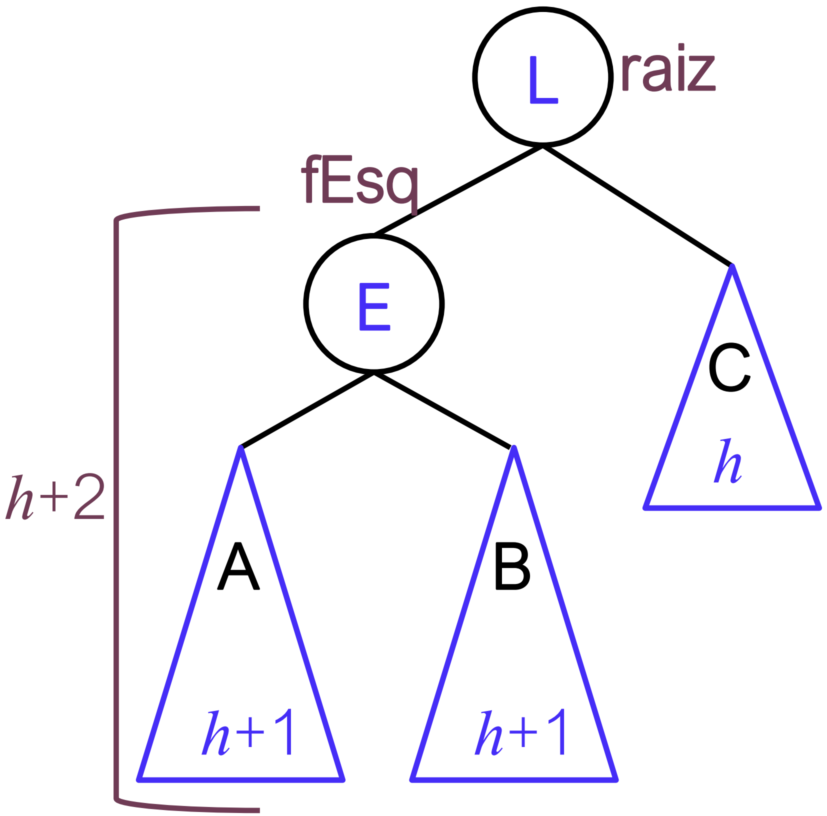




Altura da árvore: h+2

<u>Ordenação:</u> A fEsqB nDirC raizD

Inserção L-E

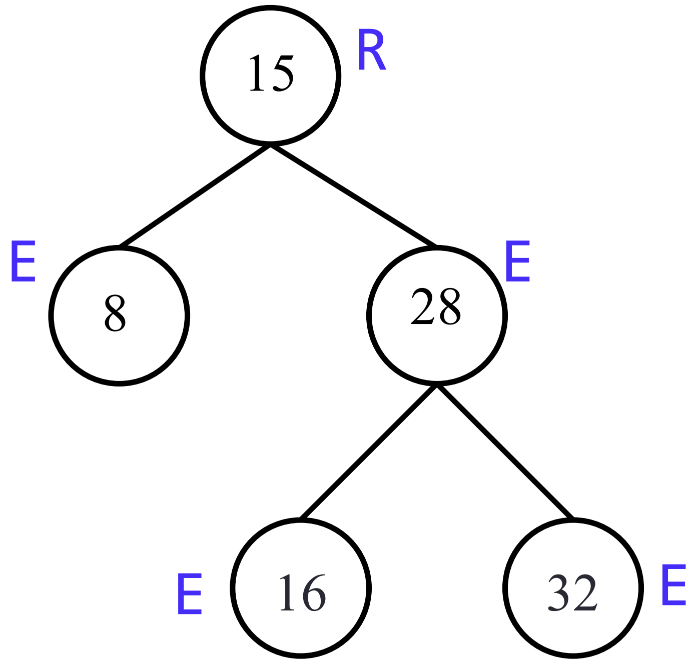


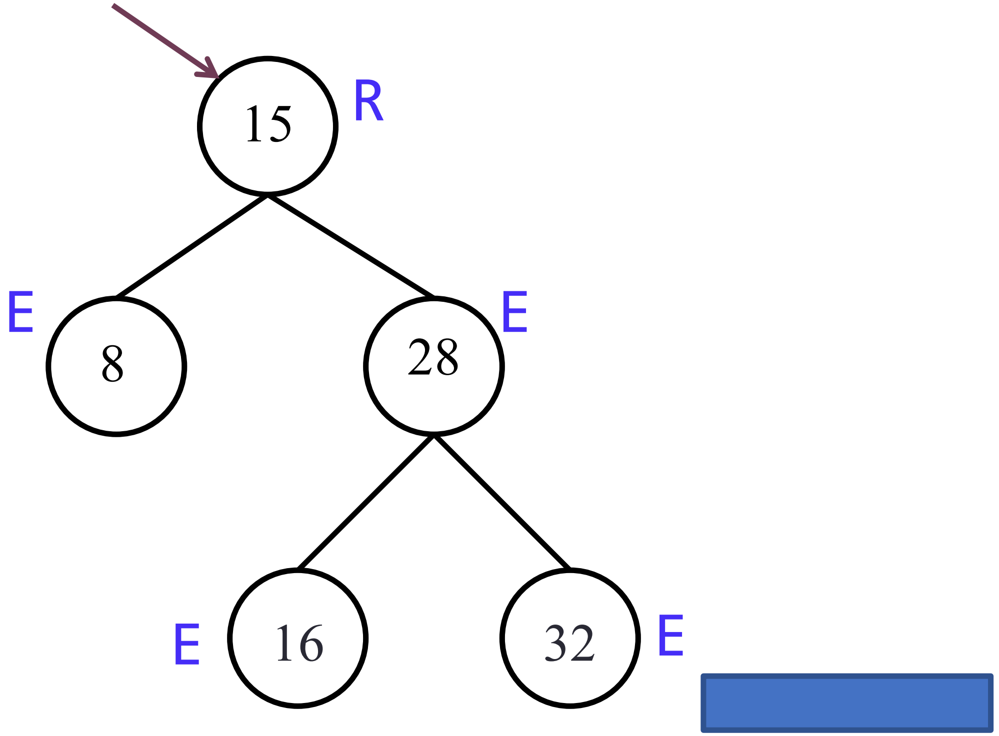
Impossível: Inseriu-se um nó

Inserção - Exemplo aproximado à implementação

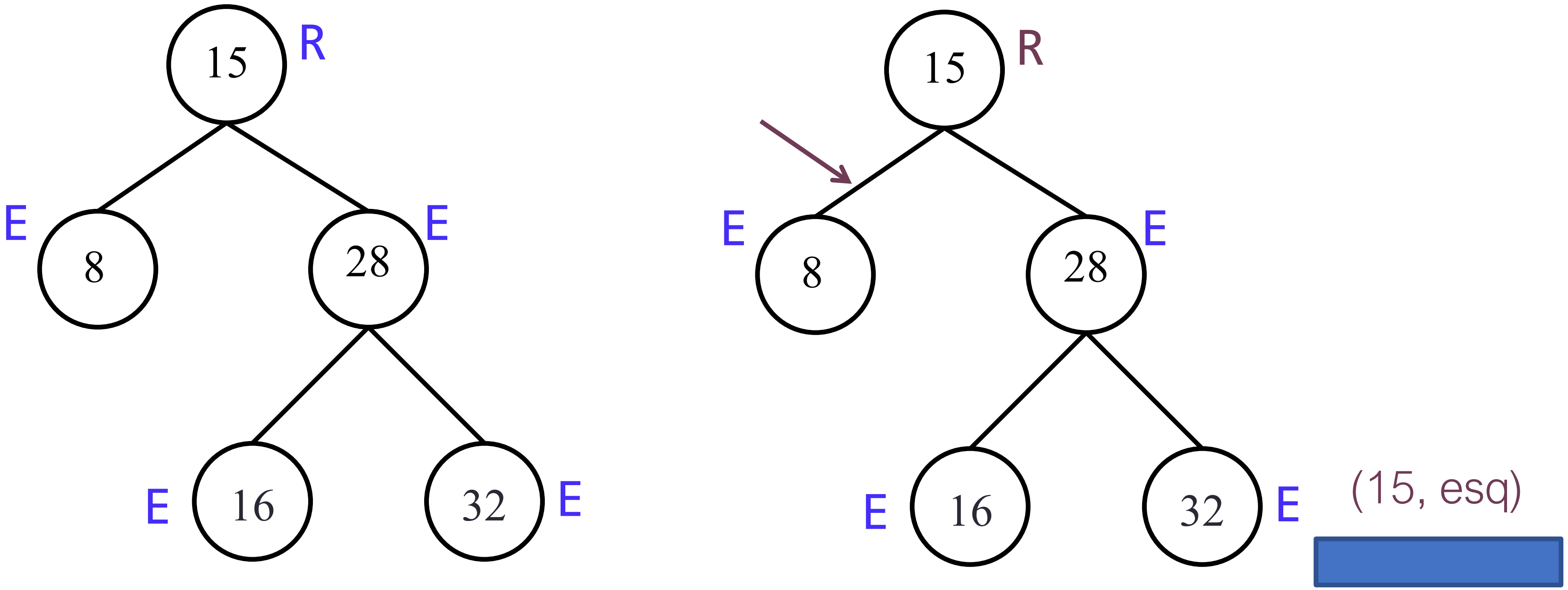
- Em cada nó, vai ser preciso manter informação sobre a diferença entre as alturas das subárvores do nó
- Depois de feita uma inserção, esta informação tem de ser atualizada, no caminho desde o nó inserido, até à raiz
  - Vamos utilizar uma pilha de objetos PathStep<K,V> que irá conter os vários antecessores do nó (o caminho percorrido até à inserção)

Inserir 4

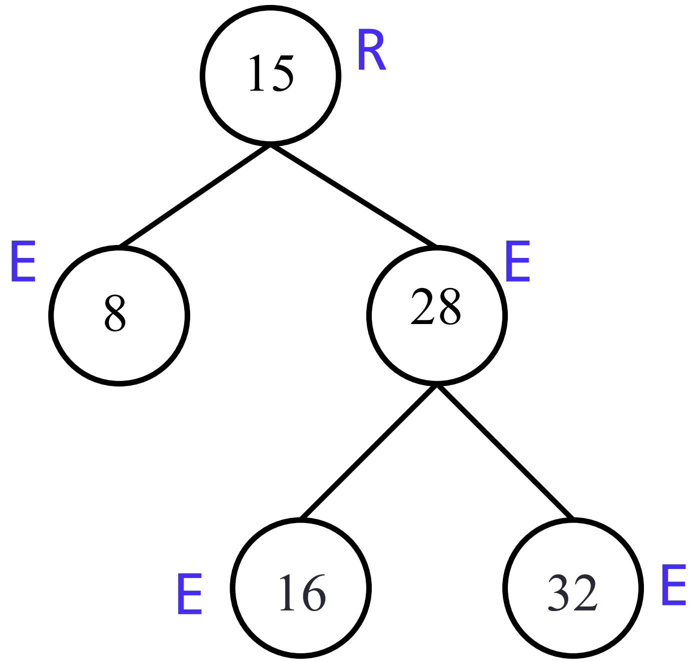


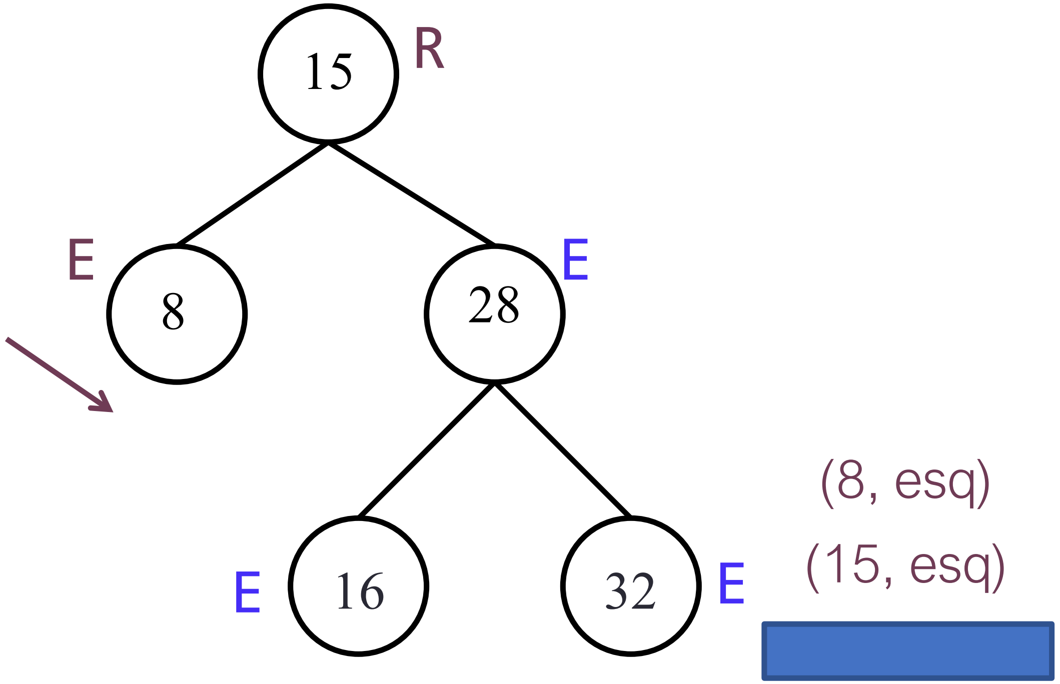


Inserir 4

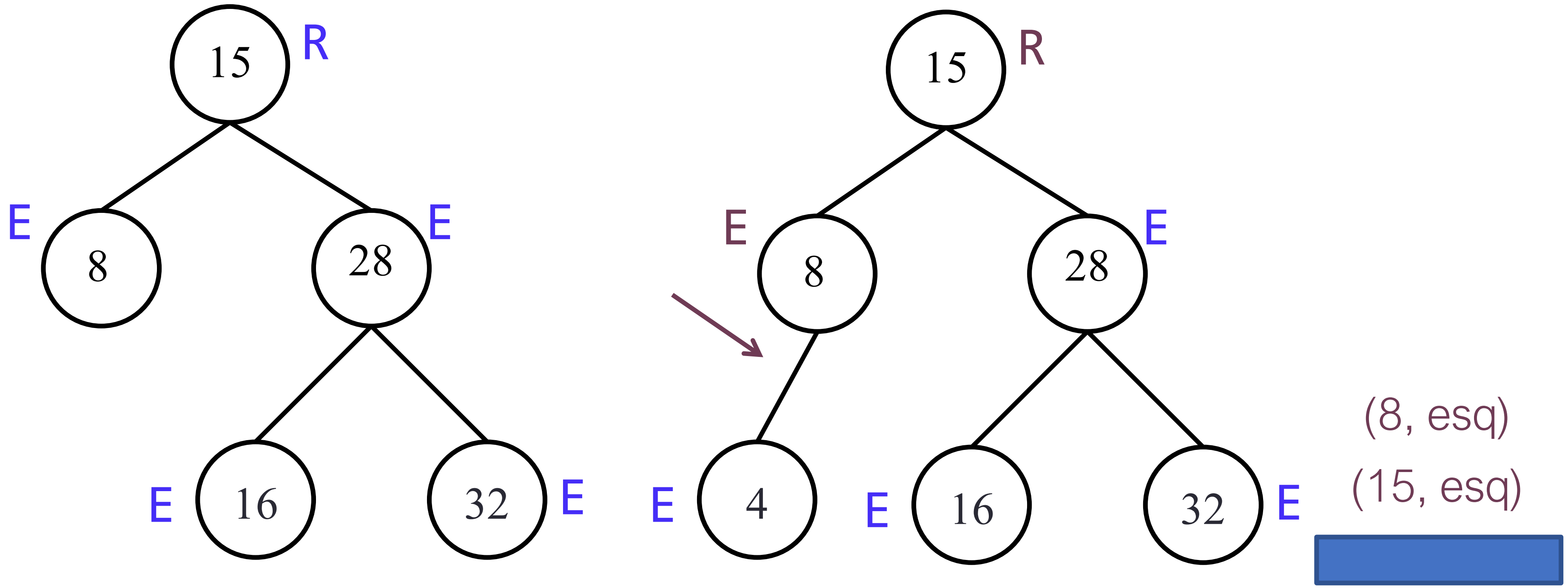


Inserir 4





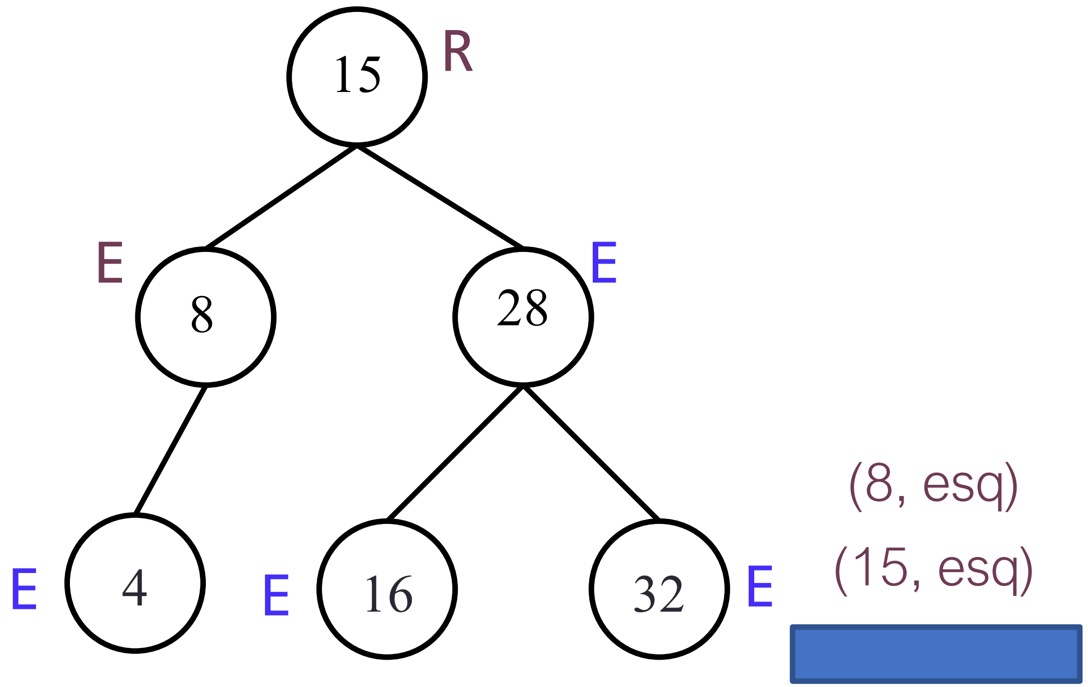
Inserir 4



A árvore cresceu

Inserir 4

Passo:



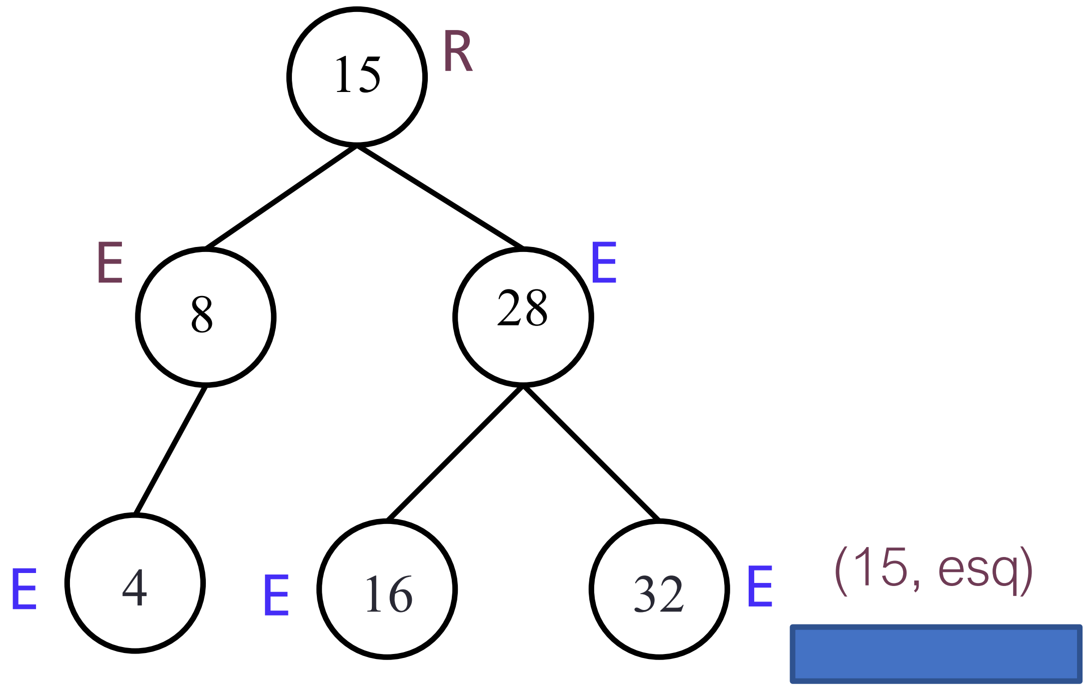
Inserir 4

Passo: (8, esq)

Subárvore esquerda cresceu

E → L

Árvore Cresceu



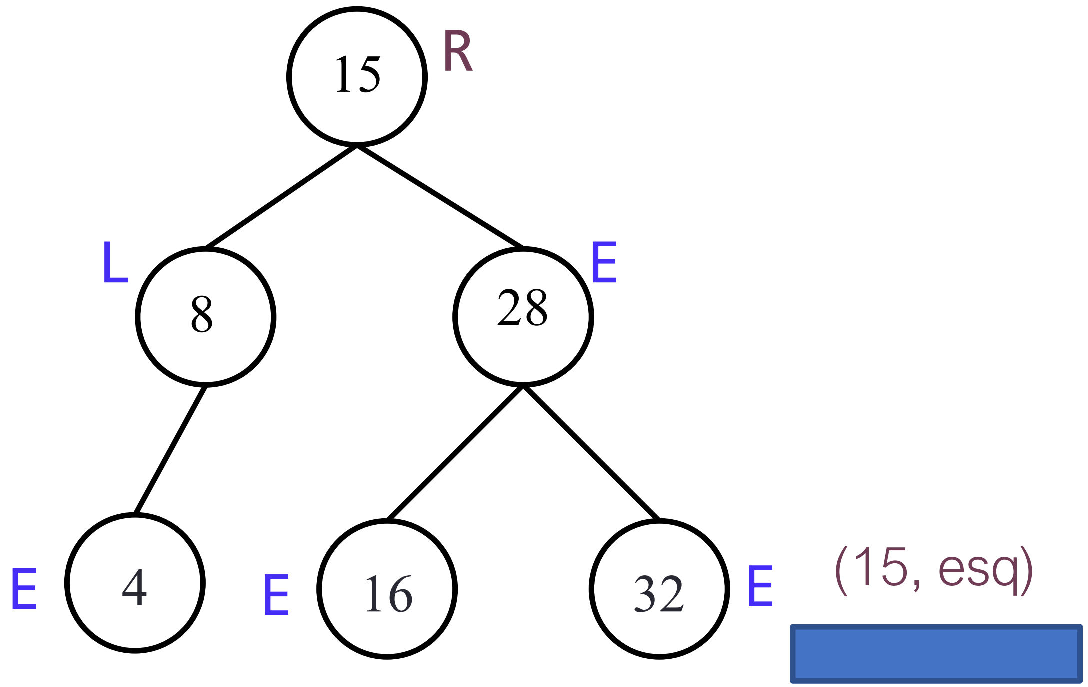
Inserir 4

Passo: (8, esq)

Subárvore esquerda cresceu

E → L

Árvore Cresceu



Inserir 4

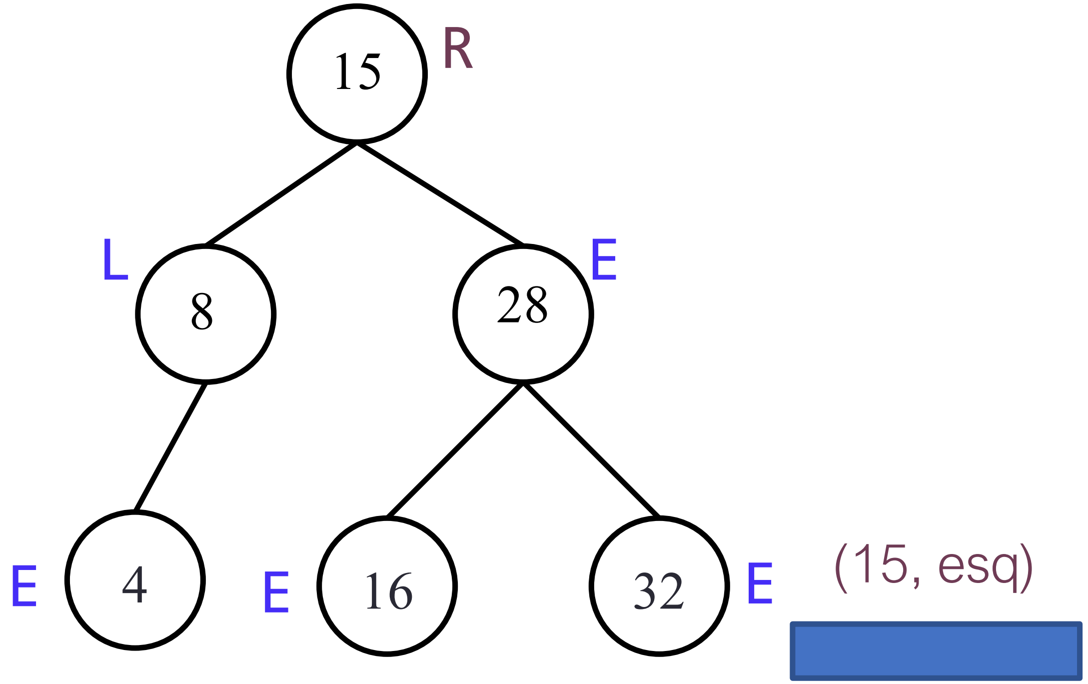
Passo: (8, esq)

Subárvore esquerda cresceu

E → L

Árvore Cresceu

Passo:



Inserir 4

Passo: (8, esq)

Subárvore esquerda cresceu

E → L

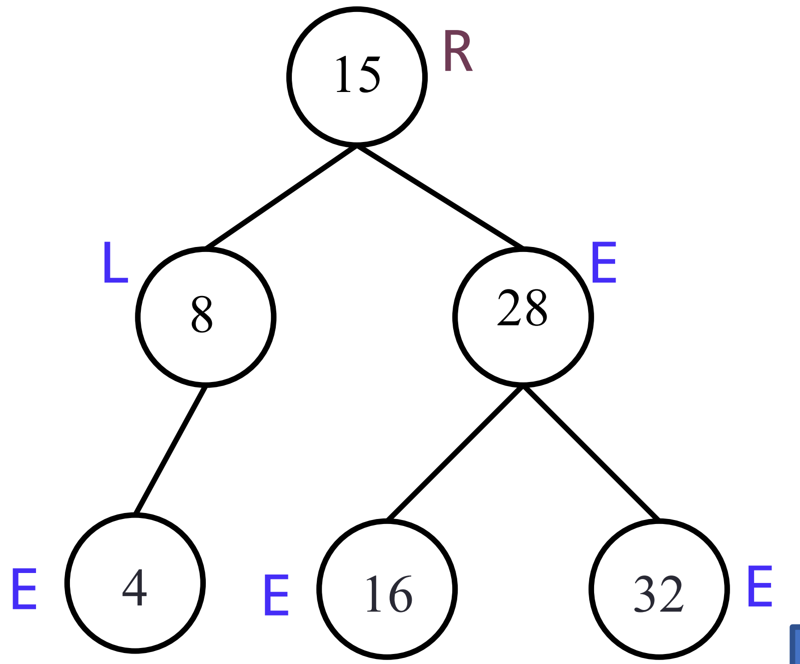
Árvore Cresceu

Passo: (15, esq)

Subárvore esquerda cresceu

R → E

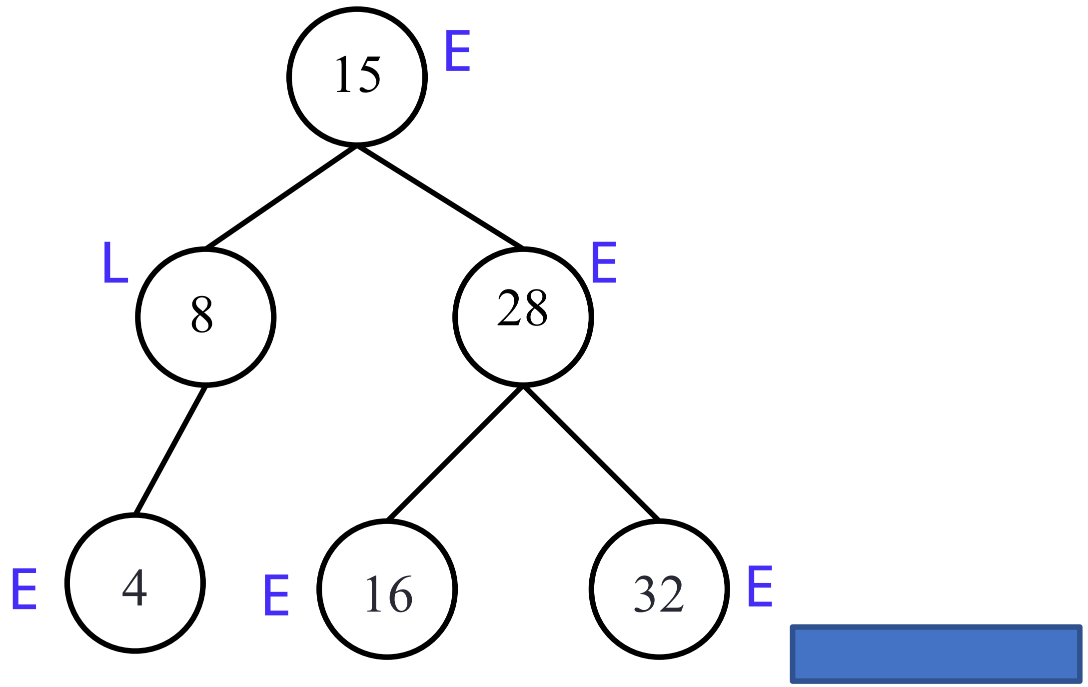
Árvore Não Cresceu



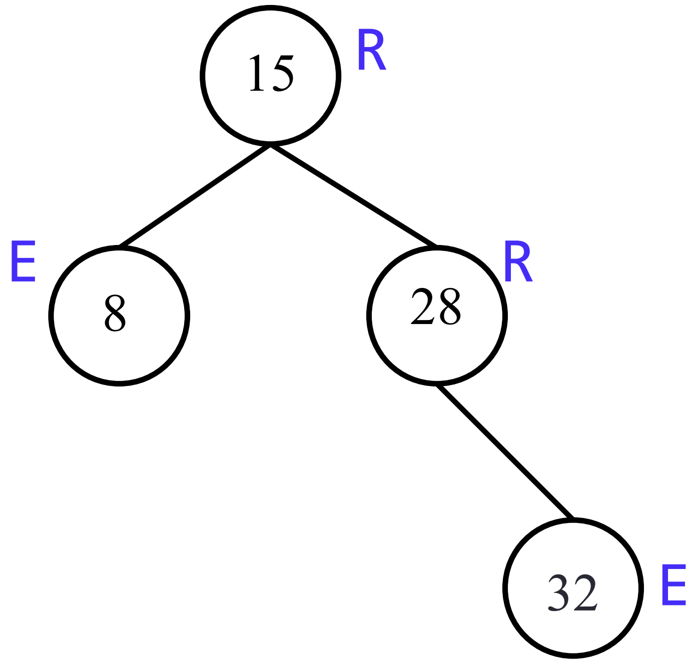
Inserir 4

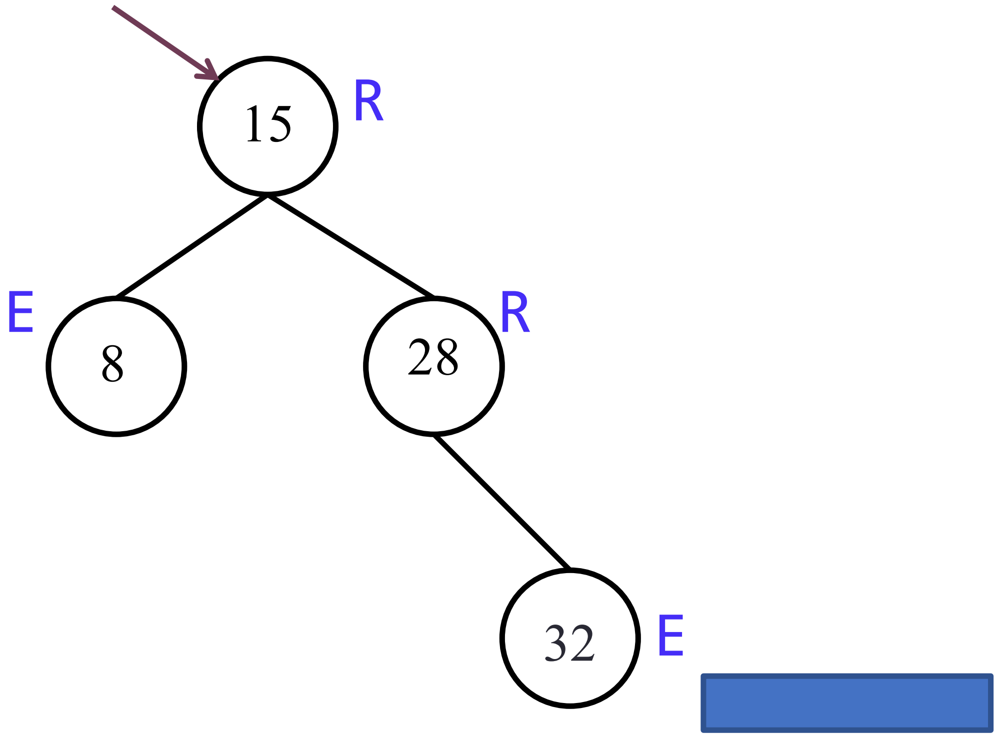
Passo: (8, esq)
  Subárvore esquerda cresceu
  E → L
  Árvore Cresceu

Passo: (15, esq)
  Subárvore esquerda cresceu
  R → E
  Árvore Não Cresceu

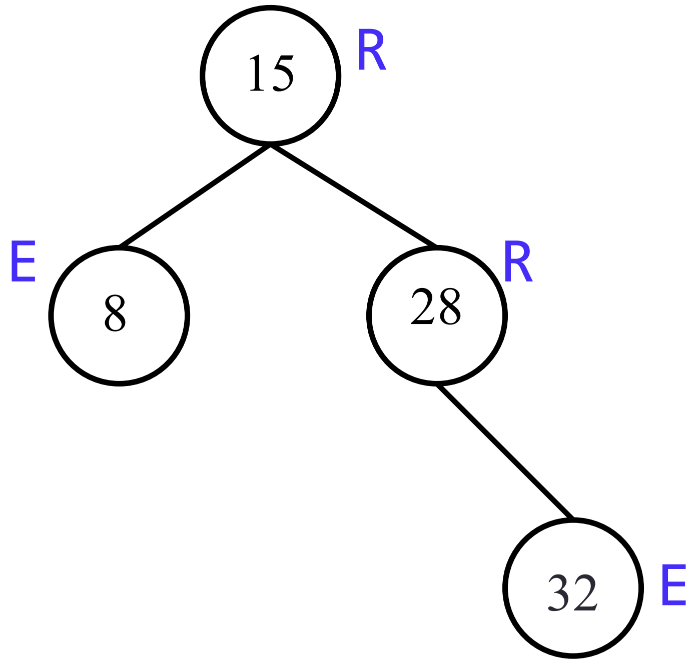


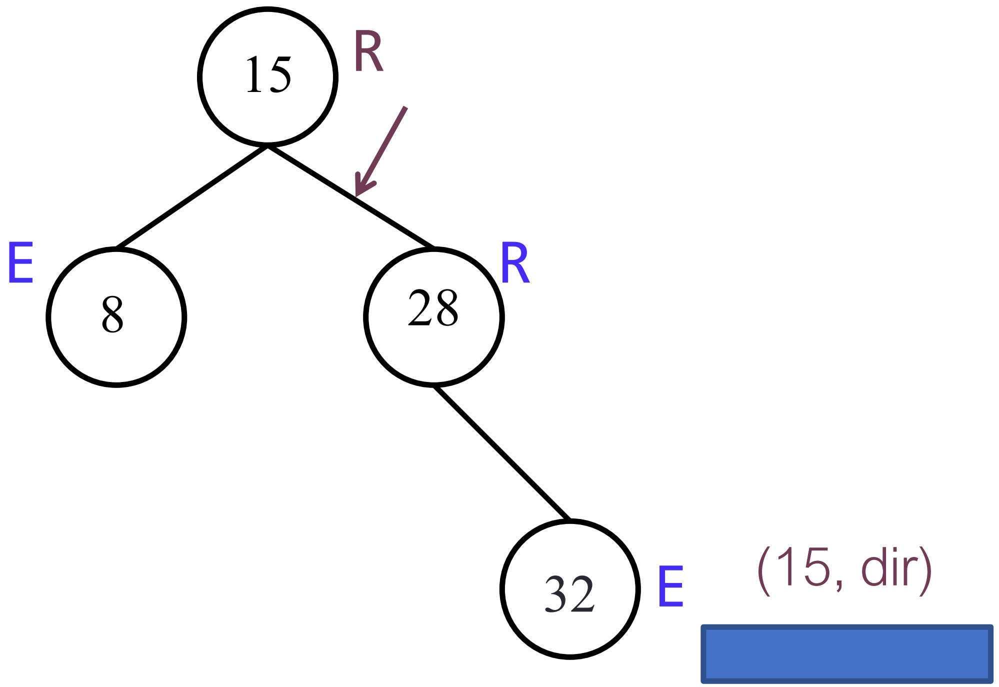
Inserir 45



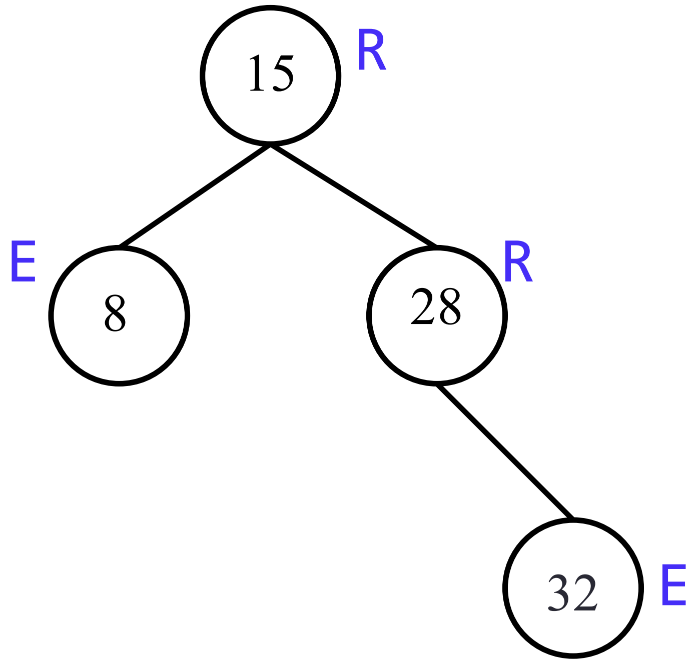


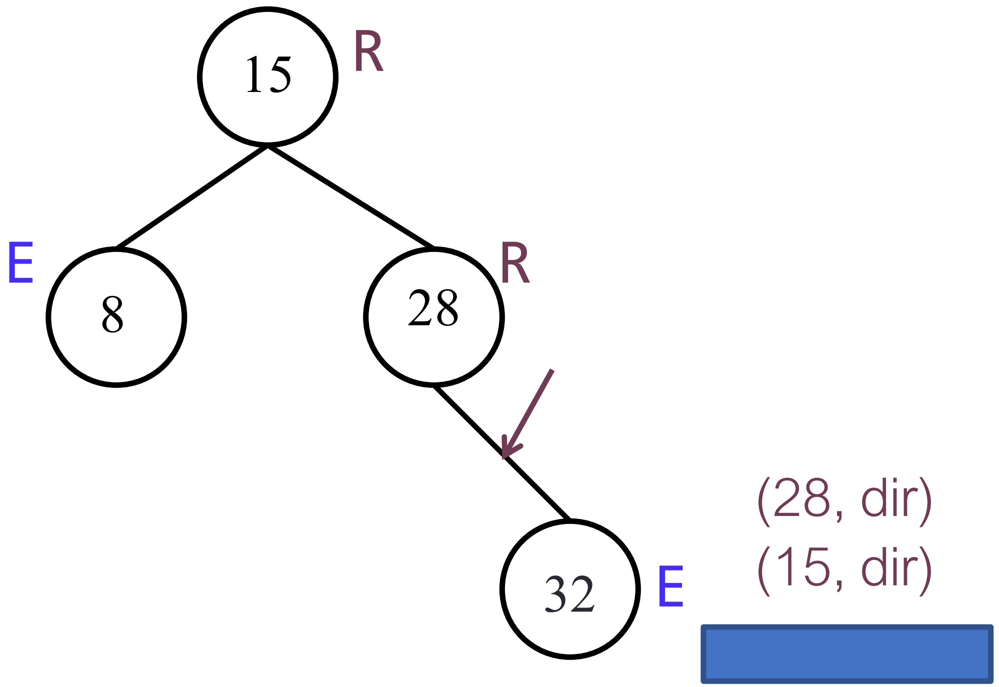
Inserir 45



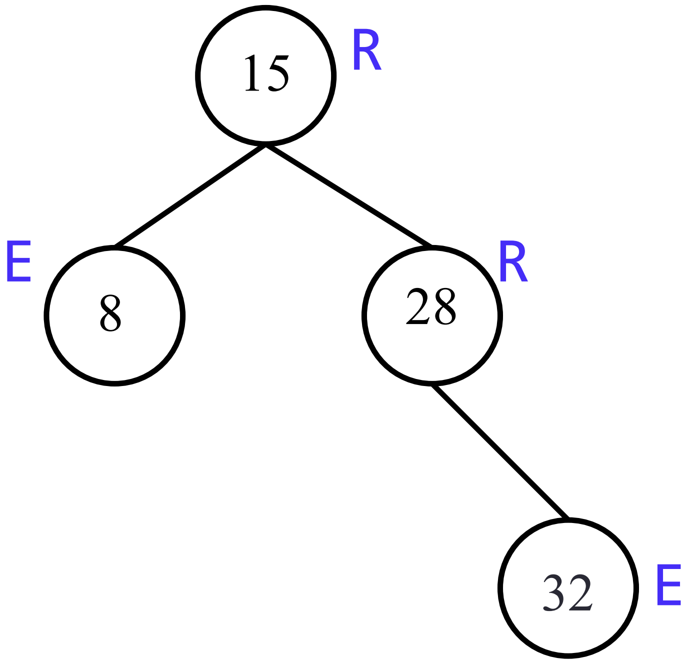


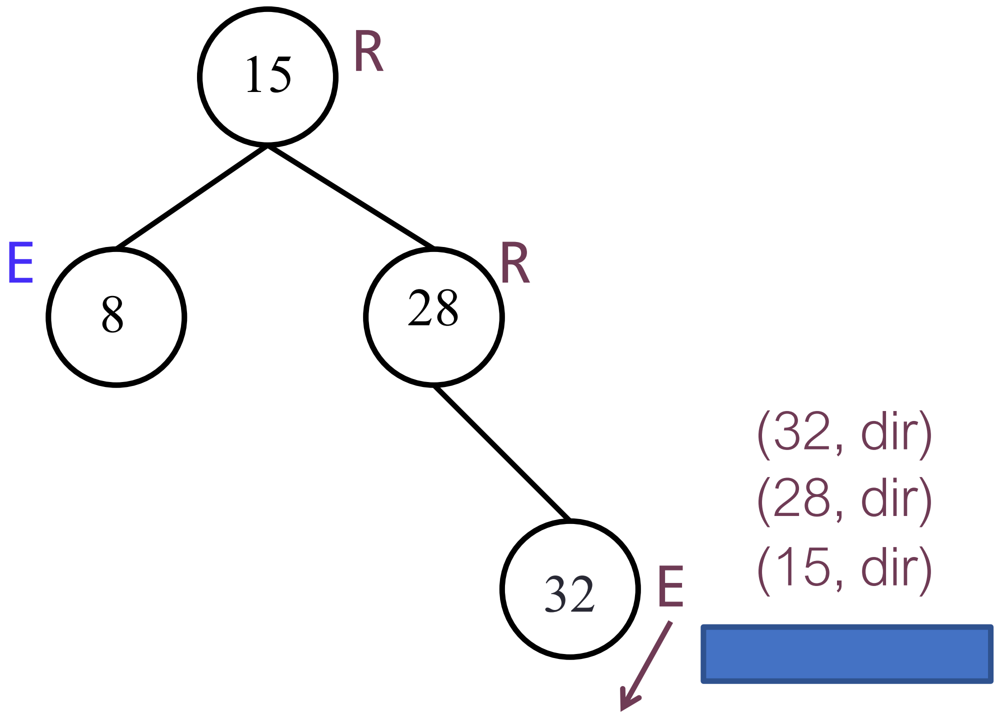
Inserir 45



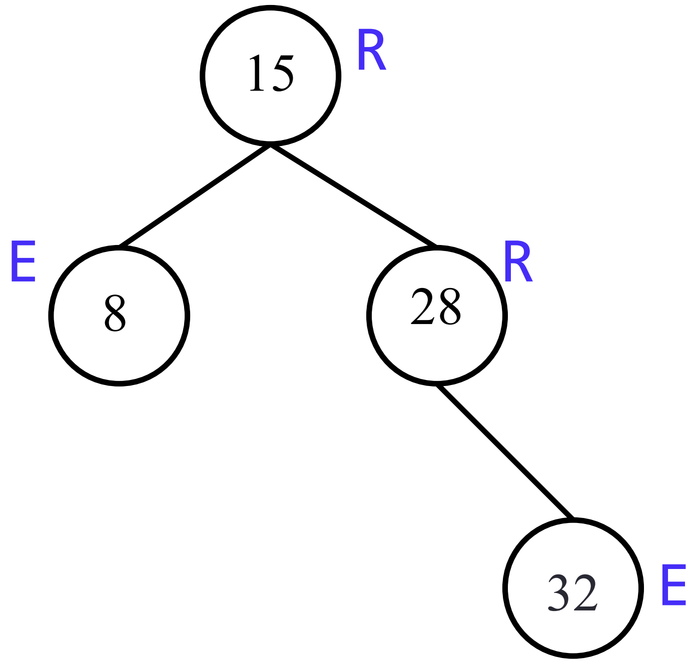


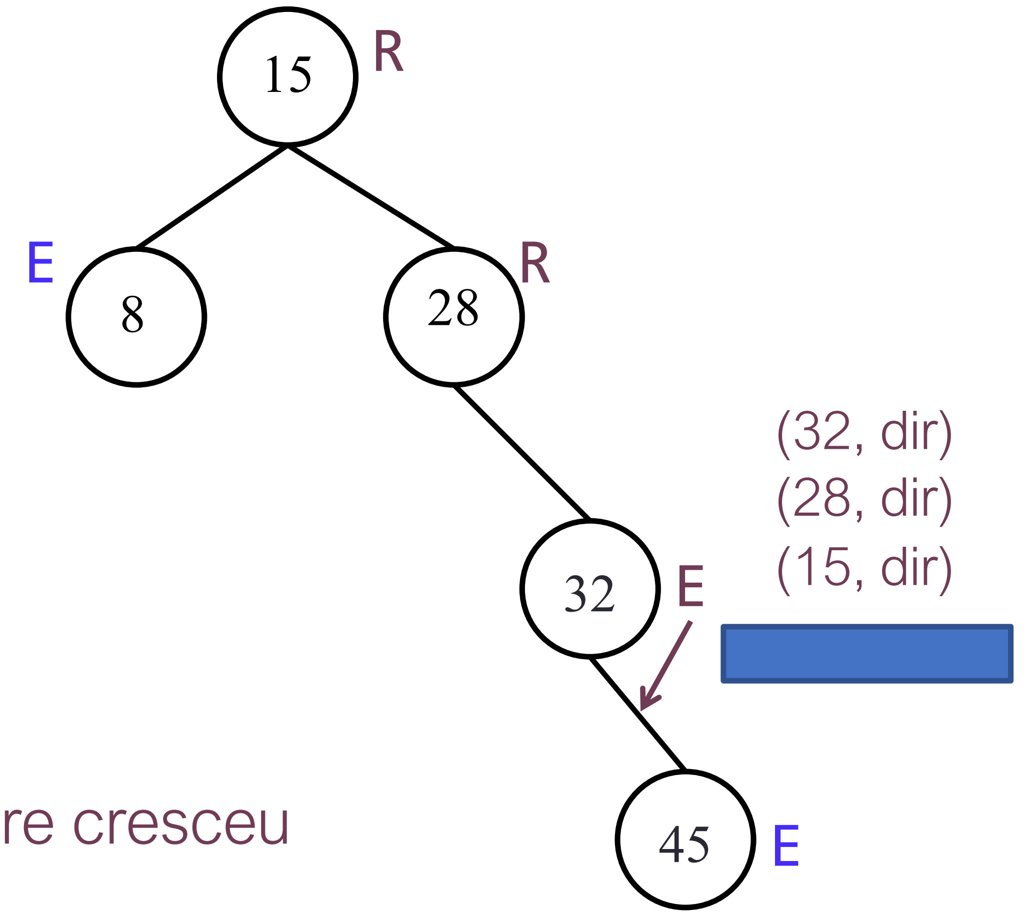
Inserir 45





Inserir 45

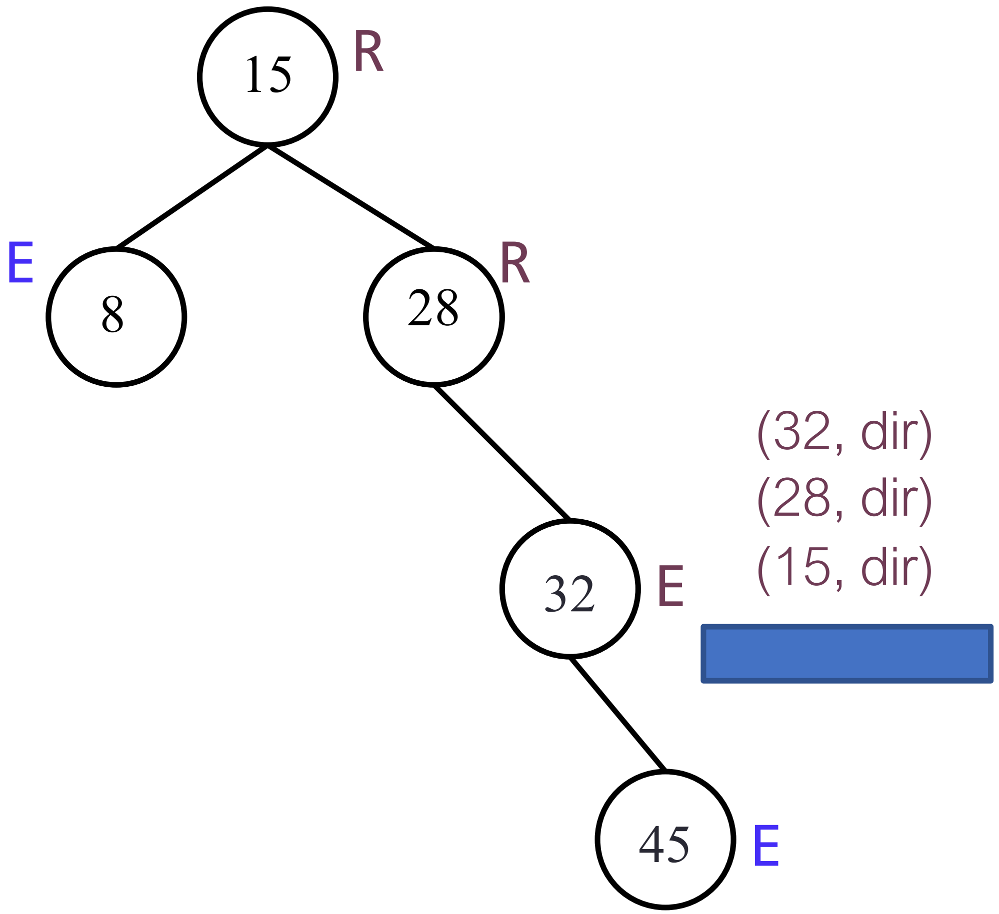




A árvore cresceu

Inserir 45

Passo:



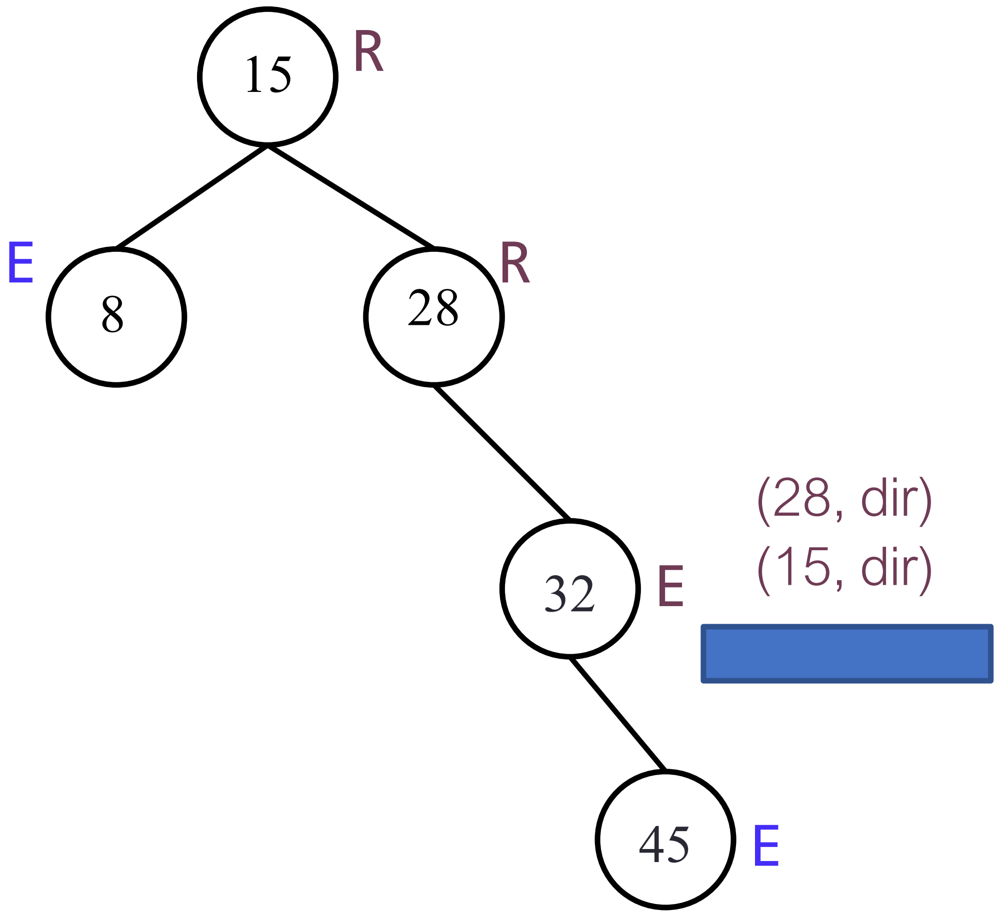
Inserir 45

Passo: (32, dir)

Subárvore direita cresceu

E → R

Árvore Cresceu



Inserir 45

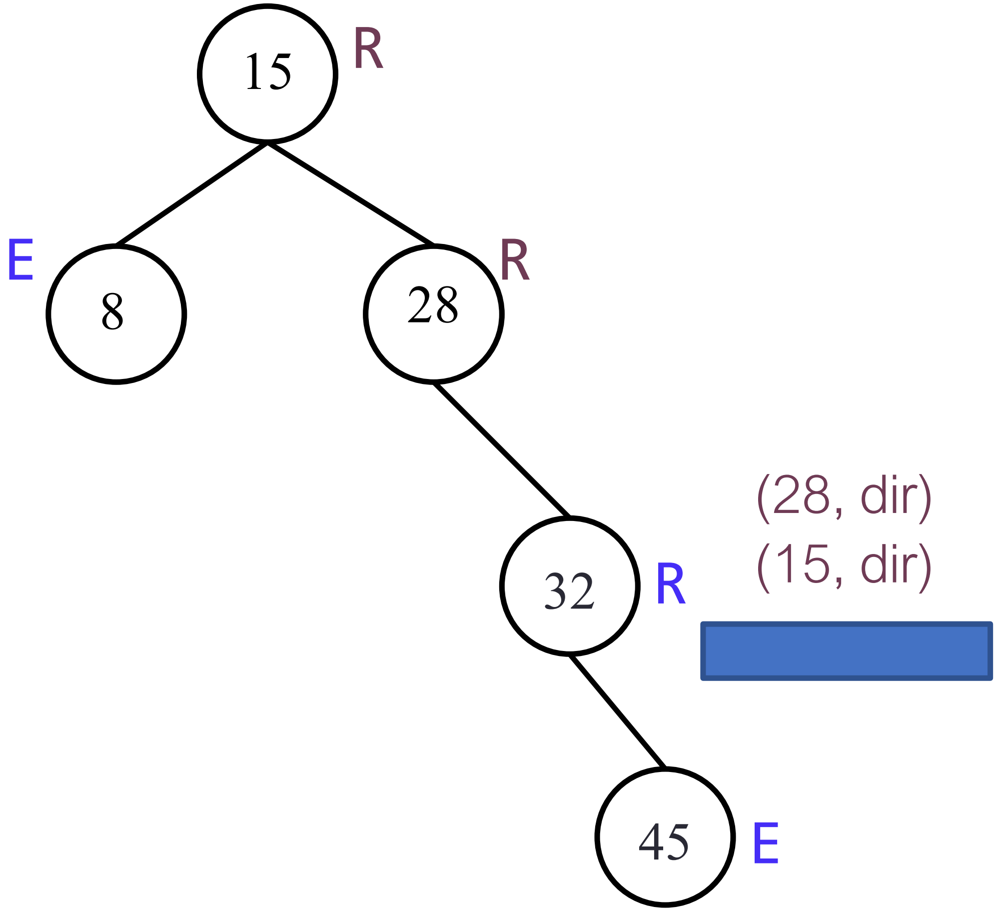
Passo: (32, dir)

Subárvore direita cresceu

E → R

Árvore Cresceu

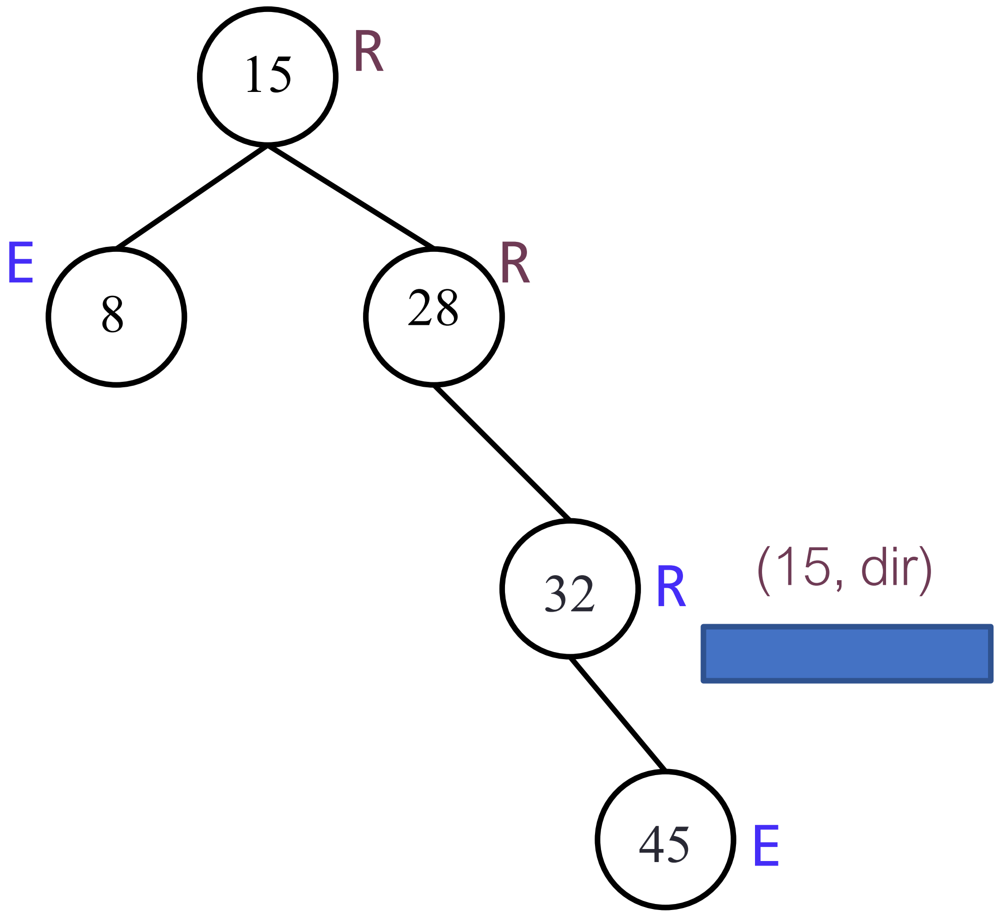
Passo:



Inserir 45

Passo: (32, dir)
Subárvore direita cresceu
E → R
Árvore Cresceu

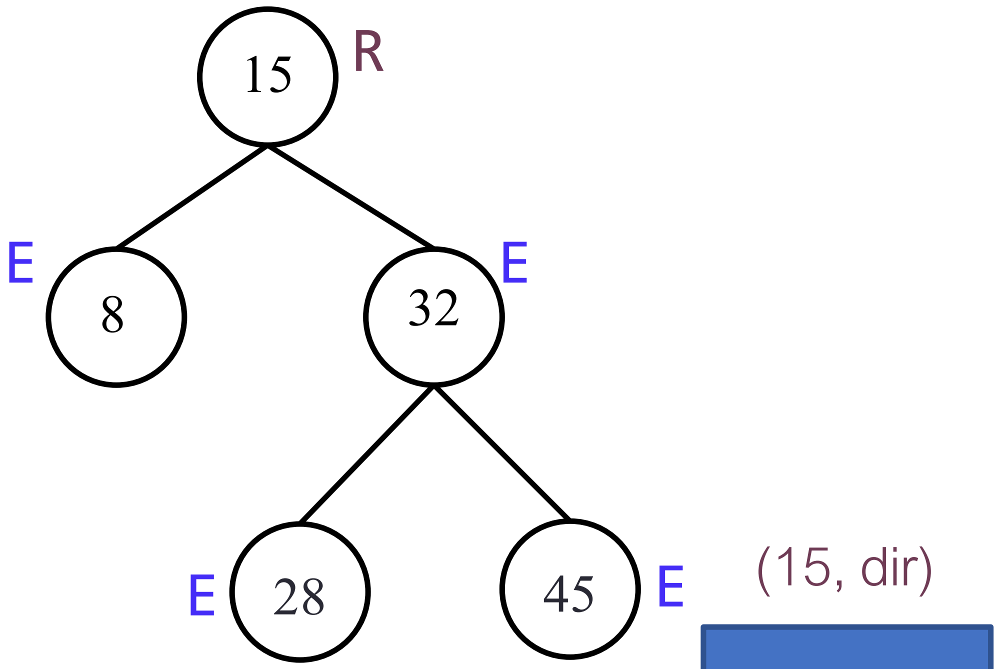
Passo: (28, dir)
Subárvore direita cresceu
R → Rotação à direita
fDir R → Simples



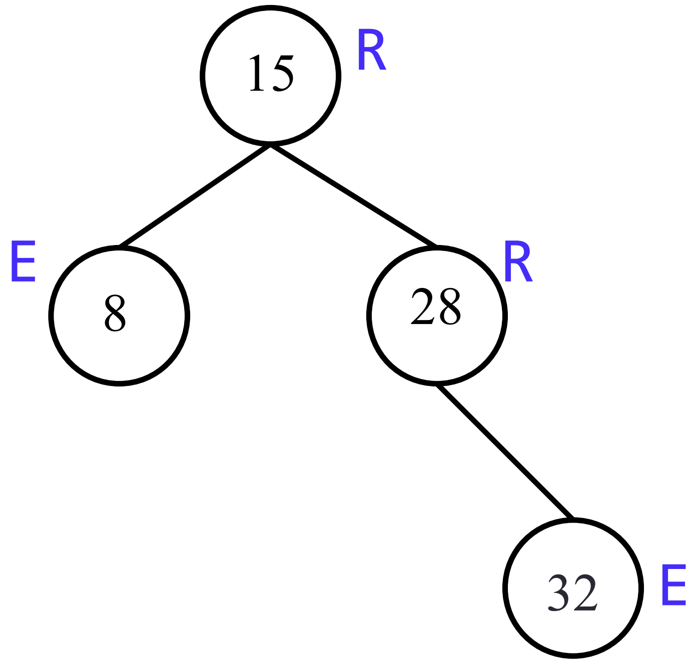
Inserir 45

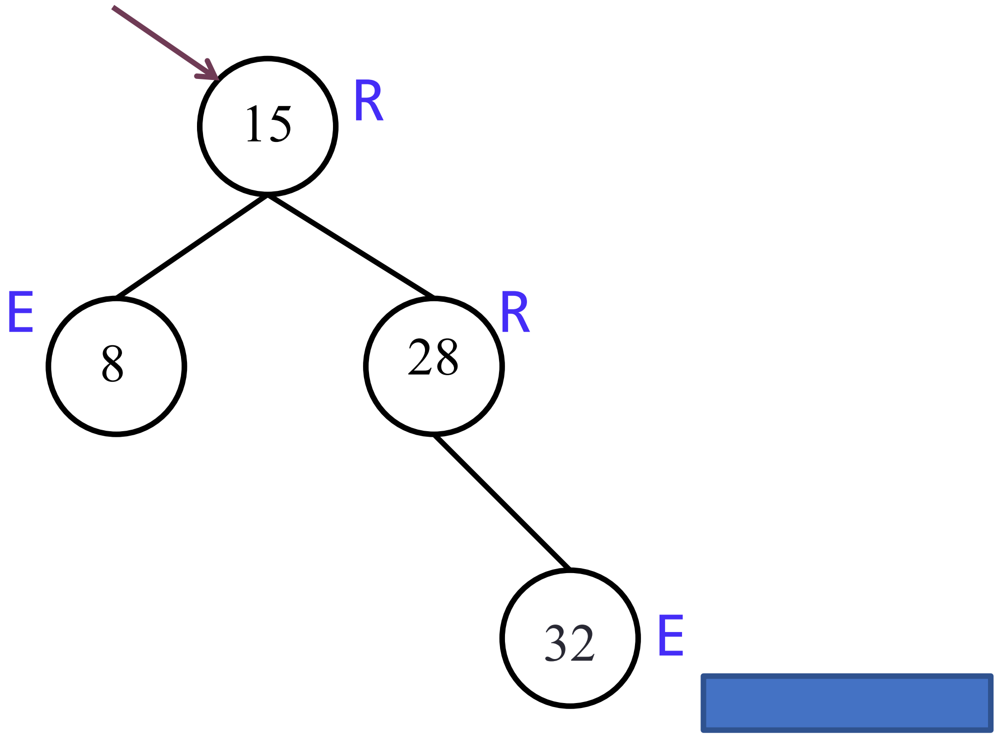
Passo: (32, dir)
  Subárvore direita cresceu
  E → R
  Árvore Cresceu

Passo: (28, dir)
  Subárvore direita cresceu
  R → Rotação à direita
  fDir R → Simples
  Árvore Não Cresceu

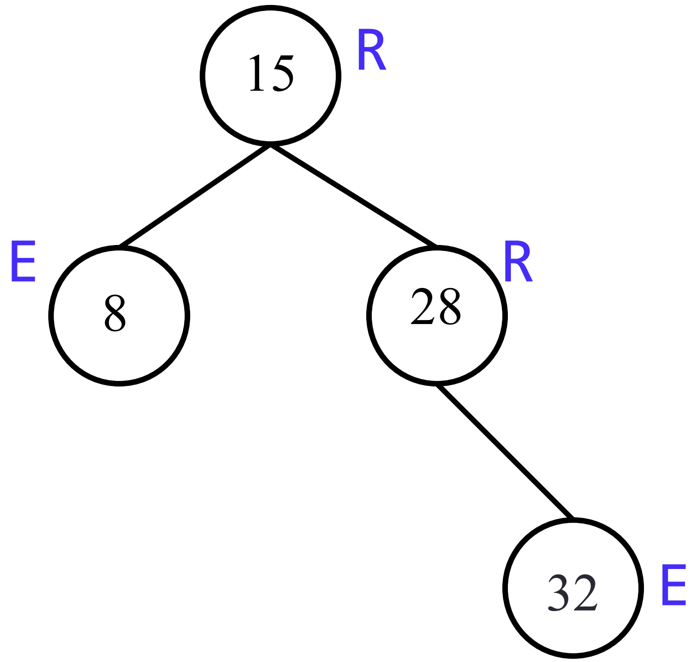


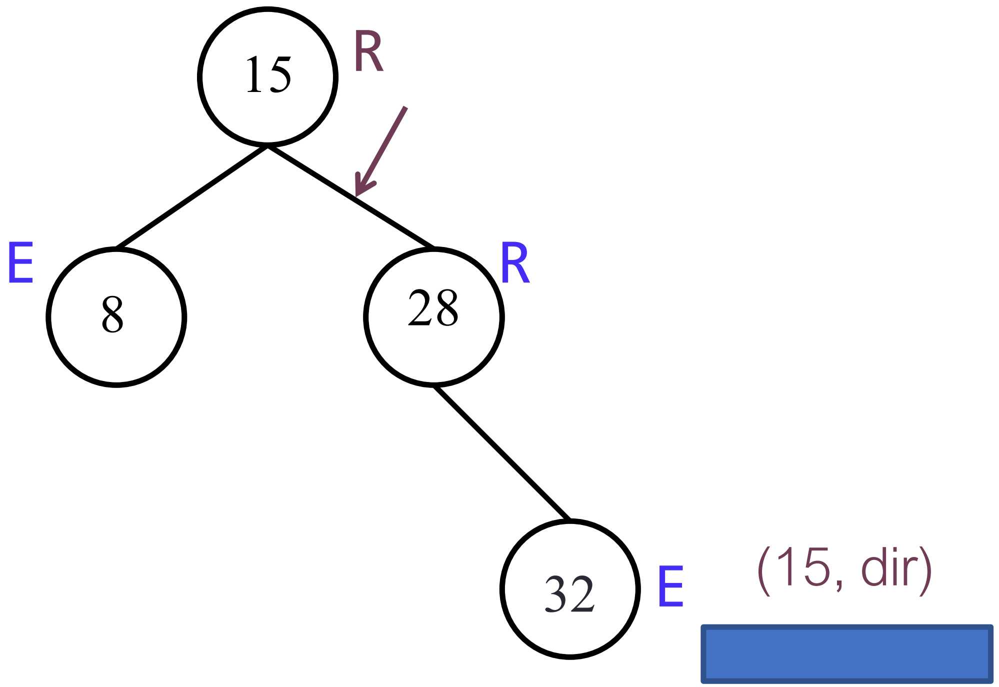
Inserir 30



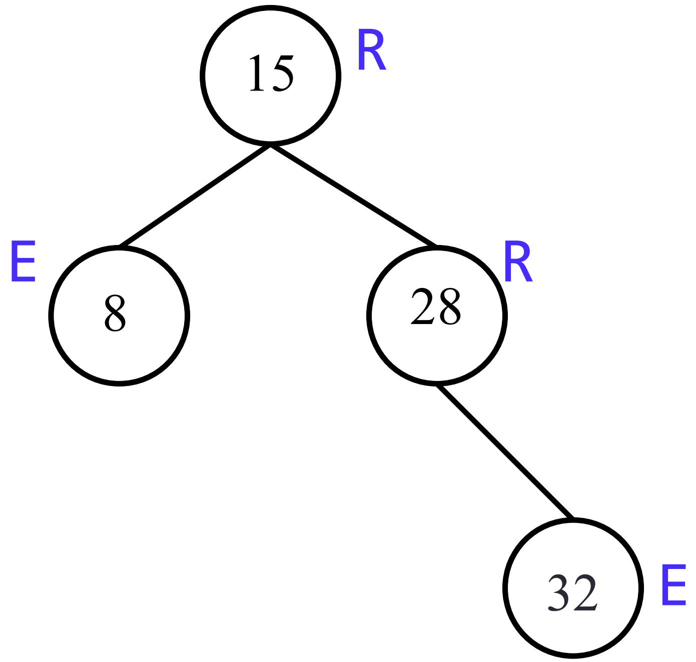


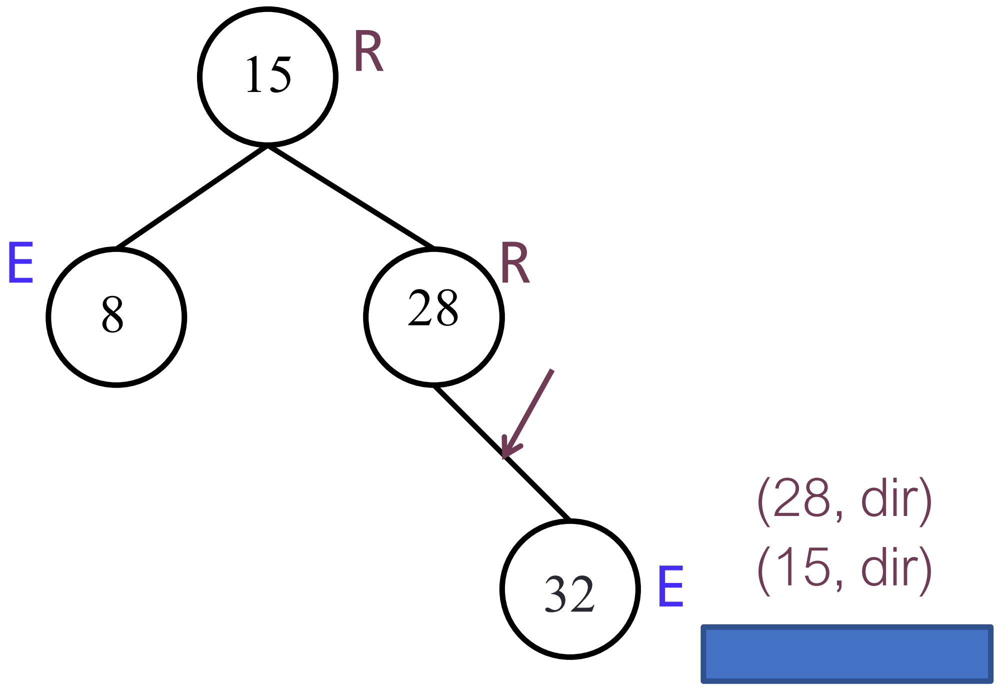
Inserir 30



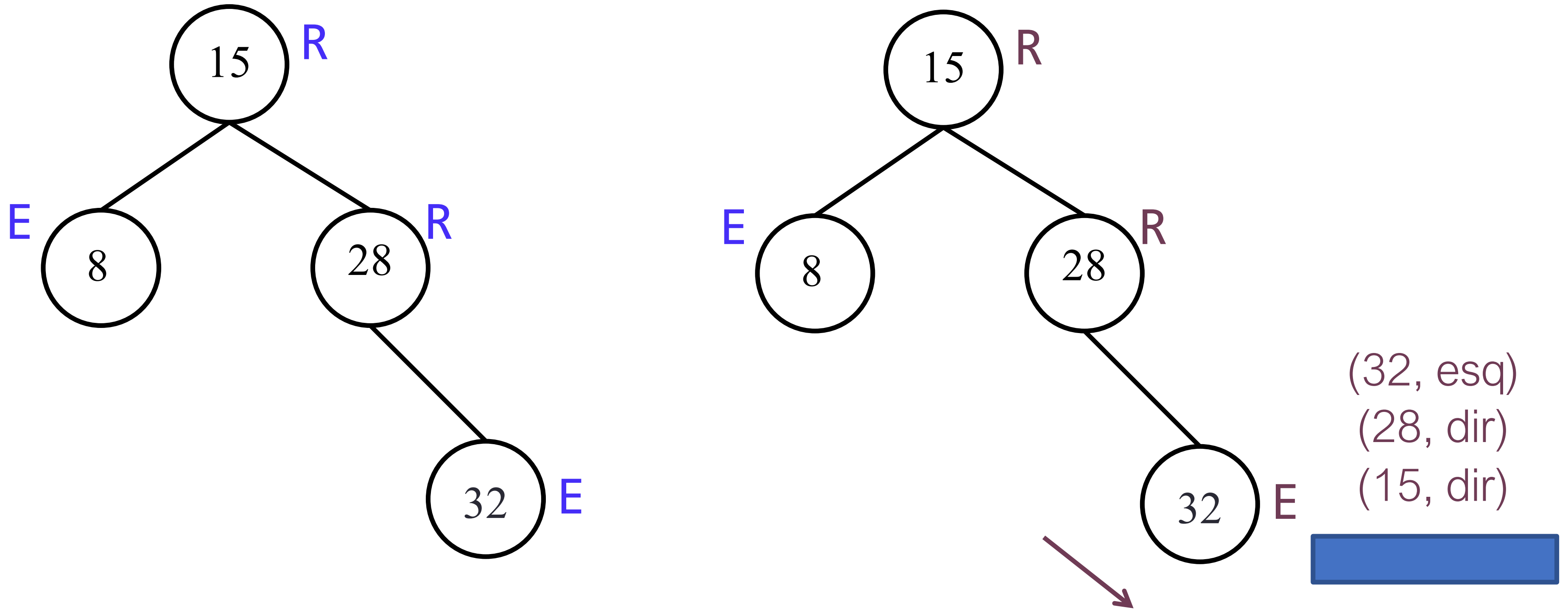


Inserir 30

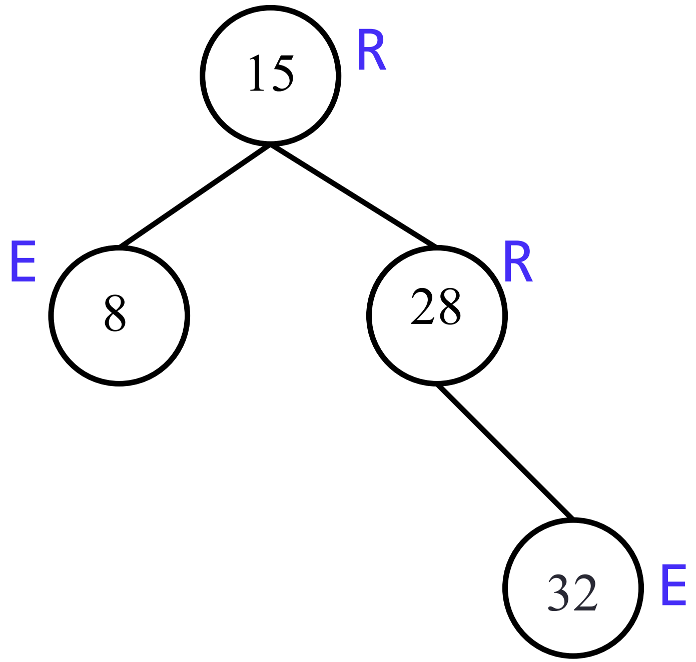


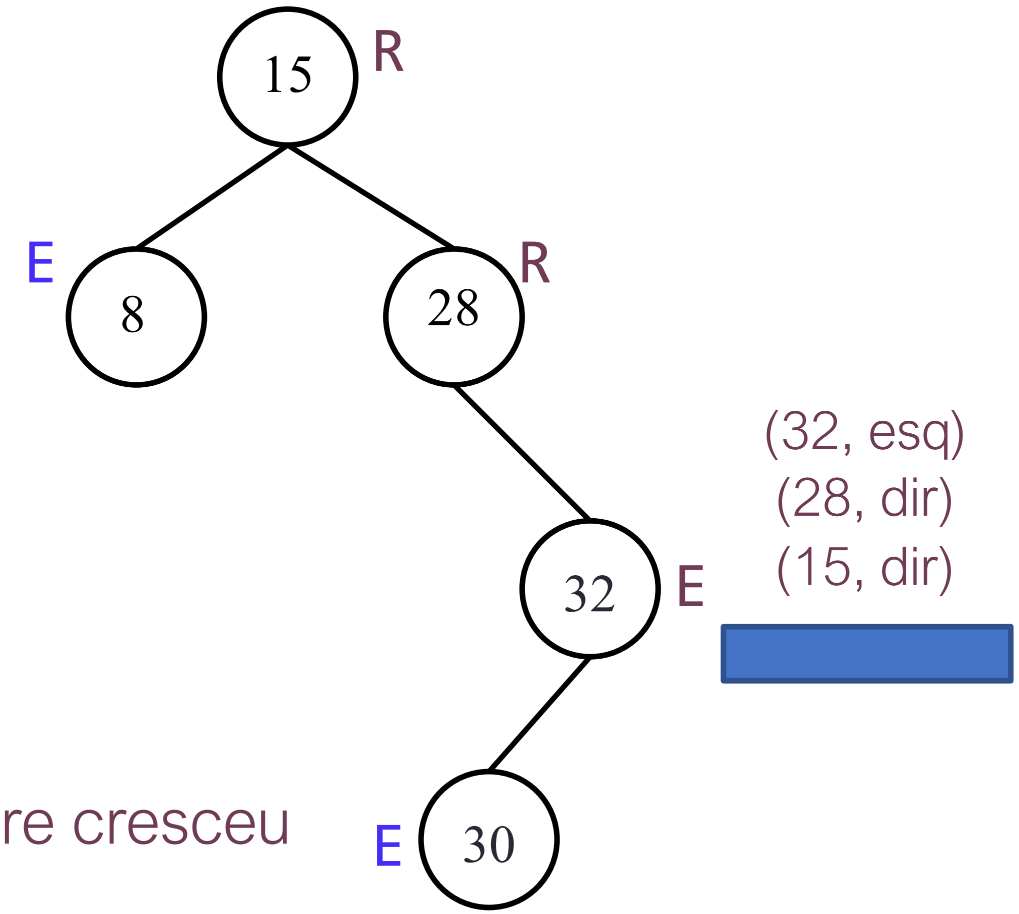


Inserir 30



Inserir 30

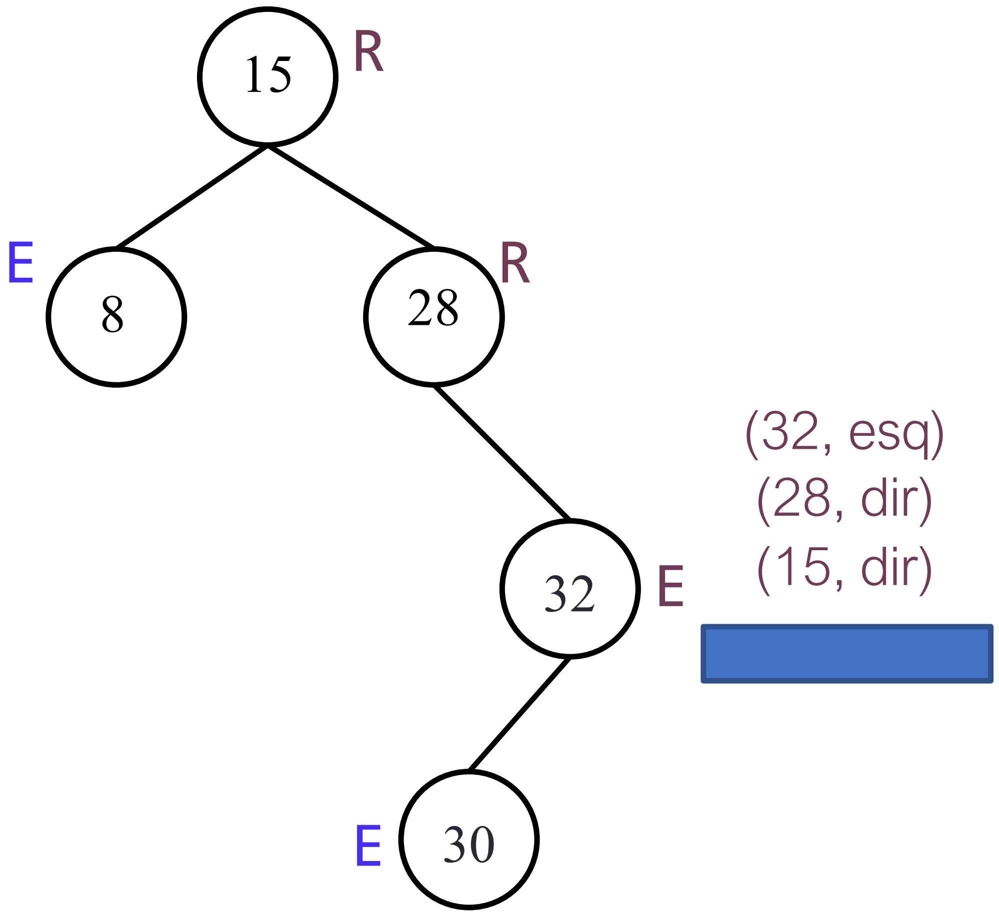




A árvore cresceu

Inserir 30

Passo:



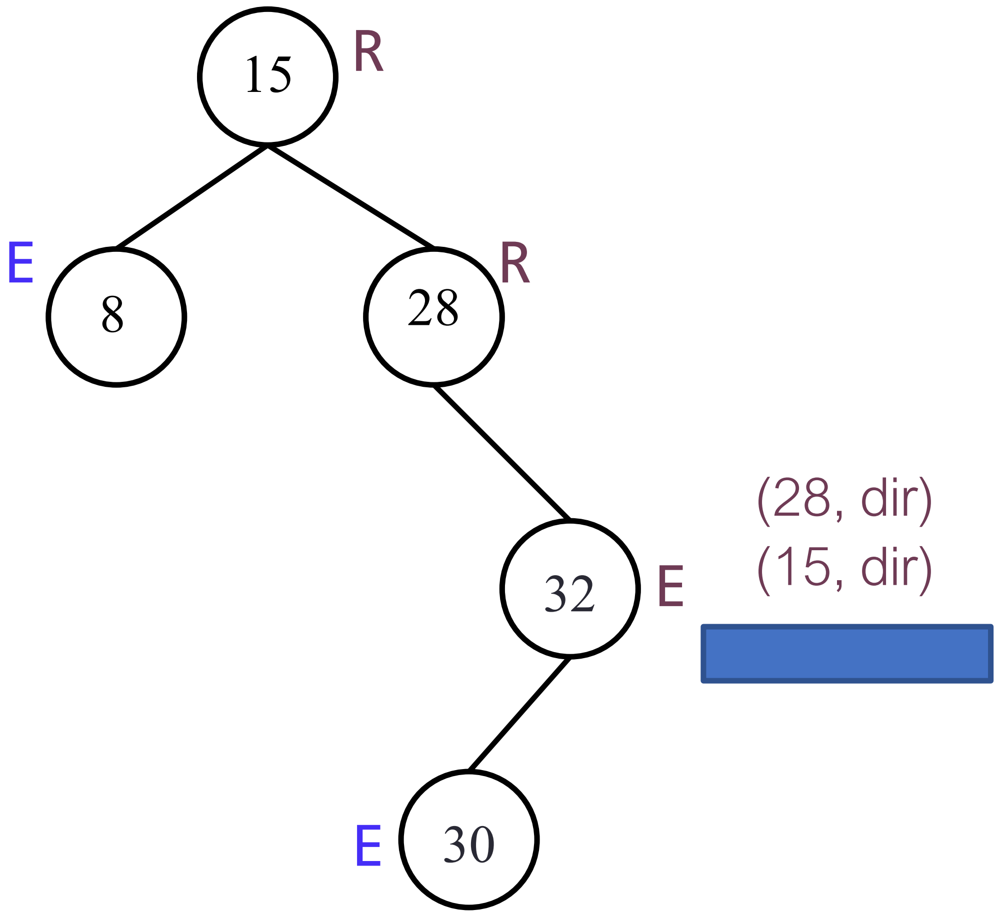
Inserir 30

Passo: (32, esq)

Subárvore esquerda cresceu

E → L

Árvore Cresceu



Inserir 30

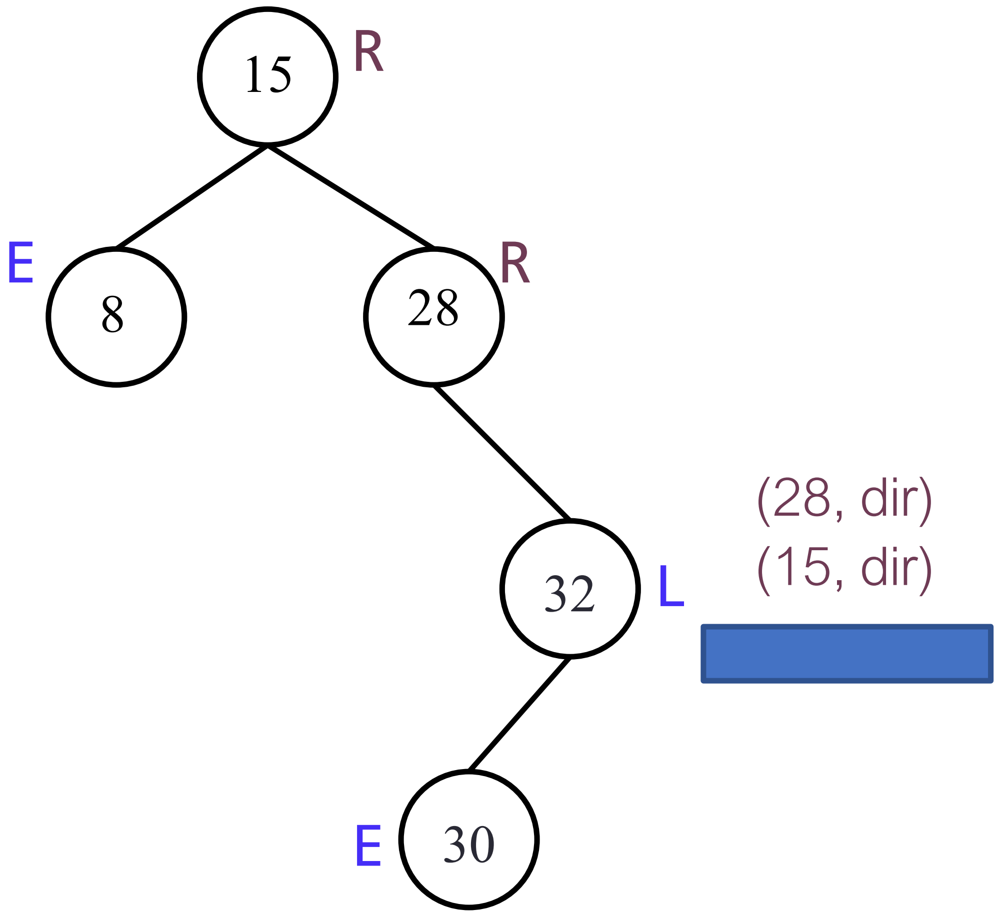
Passo: (32, esq)

Subárvore esquerda cresceu

E → L

Árvore Cresceu

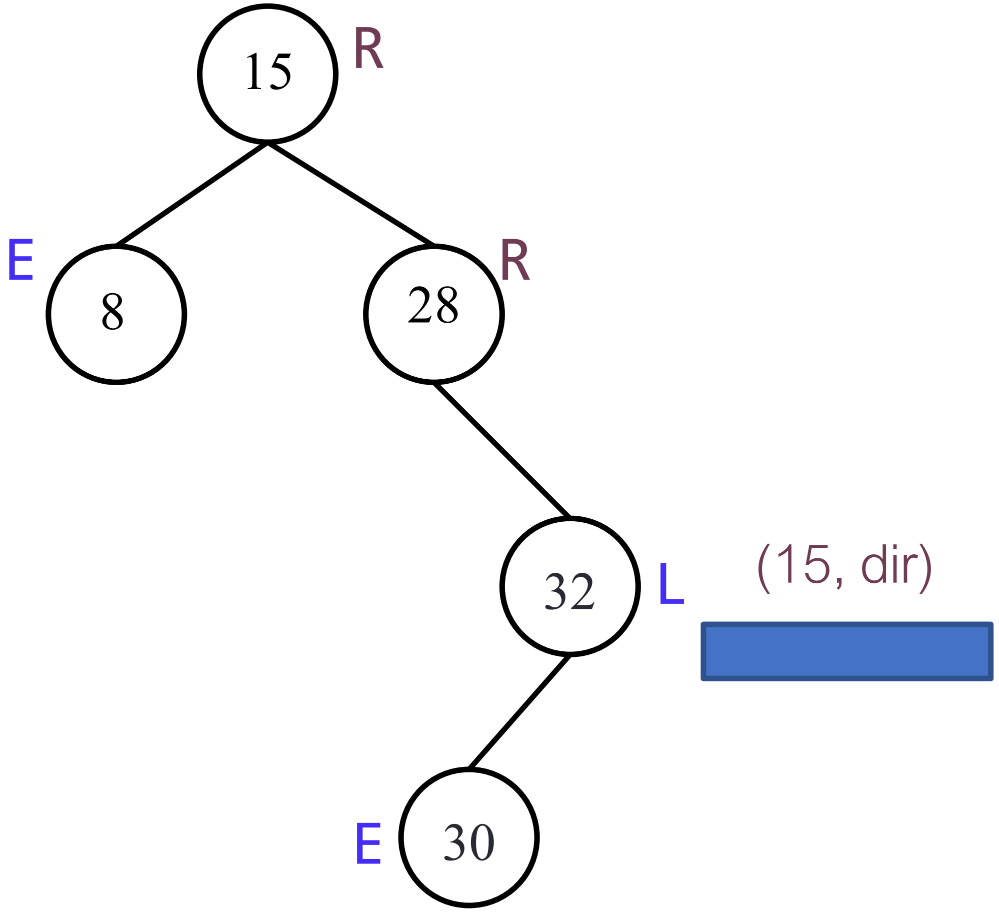
Passo:



Inserir 30

Passo: (32, esq)
Subárvore esquerda cresceu
E → L
Árvore Cresceu

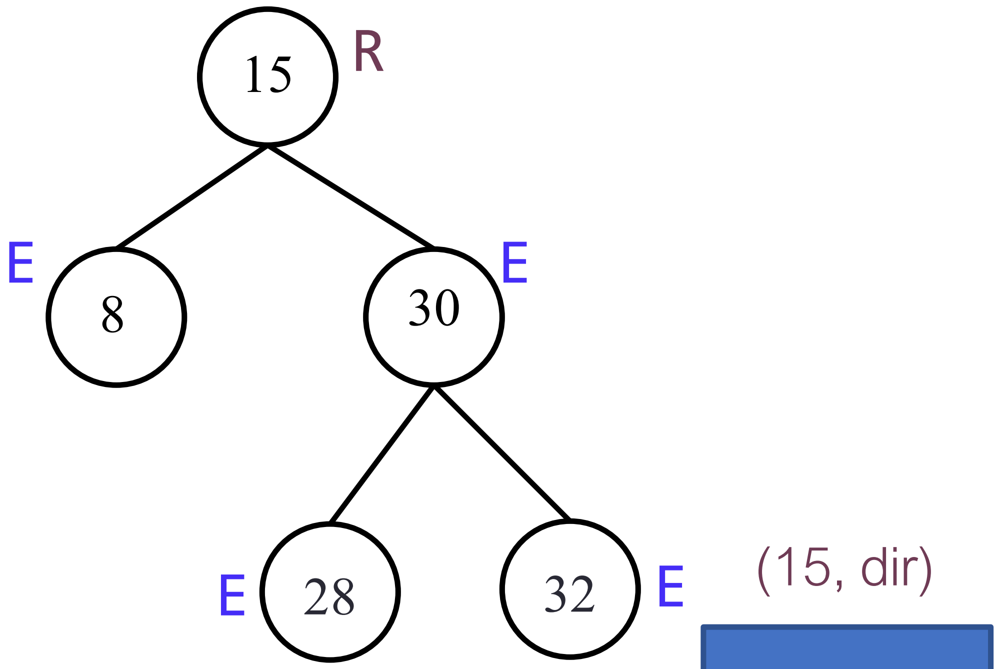
Passo: (28, dir)
Subárvore direita cresceu
R → Rotação à direita
fDir L → Dupla



Inserir 30

Passo: (32, esq)
Subárvore esquerda cresceu
E → L
Árvore Cresceu

Passo: (28, dir)
Subárvore direita cresceu
R → Rotação à direita
fDir L → Dupla
Árvore Não Cresceu



Classe Nó de Árvore AVL (1)

```
package dataStructures;

class AVLNode<K,V> extends BSTNode<K,V>{

// The balance factor of the tree rooted at the node,
// which is:
// 'E' iff height( node.getLeft() ) = height( node.getRight() );
// 'L' iff height( node.getLeft() ) = height( node.getRight() ) + 1;
// 'R' iff height( node.getLeft() ) = height( node.getRight() ) - 1.
private char balanceFactor;
```

Classe Nó de Árvore AVL (2)

```
public AVLNode( K key, V value, char balance,
                AVLNode<K,V> left, AVLNode<K,V> right ){

   super(key, value, left, right);
   balanceFactor = balance;
}

public AVLNode( K key, V value ){
   this(key, value, 'E', null, null);
}
```

Classe Nó de Árvore AVL (3)

```
    public char getBalance( ){
        return balanceFactor;
    }

    public void setBalance( char newBalance ){
        balanceFactor = newBalance;
    }
} // End of AVLNode.
```

Árvore Binária de Pesquisa Avançada - Abstrata

```
package dataStructures;

public abstract class AdvancedBSTree<K extends Comparable<K>, V>
   extends BinarySearchTree<K,V>{

// Inclui seis métodos protegidos: findNode/2, minNode/2,
// rotateLeft/3, rotateLeft/4, rotateRight/3 e rotateRight/4.


................................

}
```

Nó mínimo que guarda o caminho

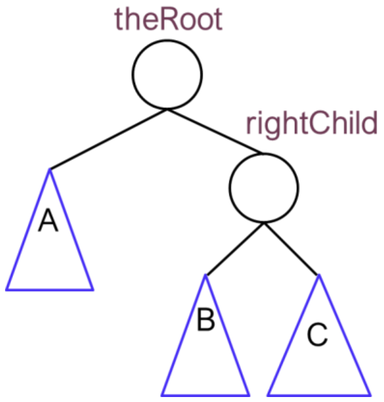
```
// Returns the node with the smallest key
// in the tree rooted at the specified node.
// Moreover, stores the path into the stack.
// Requires: theRoot != null.
protected BSTNode<K,V> minNode( BSTNode<K,V> theRoot,
                                Stack<PathStep<K,V>> path ){

   BSTNode<K,V> node = theRoot;
   while ( node.getLeft() != null ){
      path.push( new PathStep<K,V>(node, true) );
      node = node.getLeft();
   }
   return node;
}
```

```java
// Returns the node whose key is the specified key;
// or null if no such node exists.
// Moreover, stores the path into the stack.
protected BSTNode<K,V> findNode( K key, Stack<PathStep<K,V>> path )
{
    path.push( new PathStep<K,V>(null, false) );
    BSTNode<K,V> node = root;
    while ( node != null )
    {
        int compResult = key.compareTo( node.getKey() );
        if ( compResult == 0 )
            return node;
        else if ( compResult < 0 )
        {
            path.push( new PathStep<K,V>(node, true) );
            node = node.getLeft();
        }
        else
        {
            path.push( new PathStep<K,V>(node, false) );
            node = node.getRight();
        }
    }
    return null;
}
```

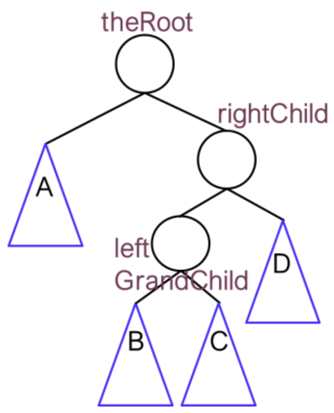
findNode que guarda o caminho

Rotação Simples à direita

```
// Performs a single right rotation rooted at theRoot.
// Every ancestor of theRoot is stored in the stack,
// which is not empty.
protected void rotateRight( BSTNode<K,V> theRoot,
      BSTNode<K,V> rightChild, Stack<PathStep<K,V>> path ){

   theRoot.setRight( rightChild.getLeft() );
   rightChild.setLeft(theRoot);
   this.linkSubtree(rightChild, path.top());
}
```



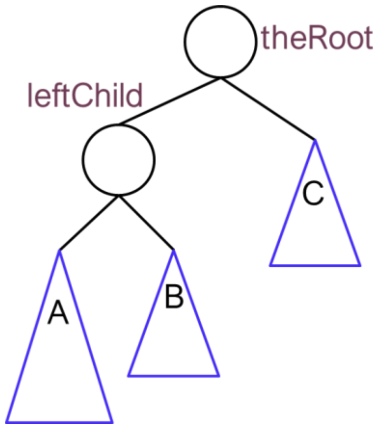
Rotação Dupla à direita




```
// Performs a double right rotation rooted at theRoot.
// Every ancestor of theRoot is stored in the stack,
// which is not empty.
protected void rotateRight( BSTNode<K,V> theRoot,
   BSTNode<K,V> rightChild, BSTNode<K,V> leftGrandchild,
   Stack<PathStep<K,V>> path ){

   theRoot.setRight( leftGrandchild.getLeft() );
   rightChild.setLeft( leftGrandchild.getRight() );
   leftGrandchild.setLeft(theRoot);
   leftGrandchild.setRight(rightChild);
   this.linkSubtree(leftGrandchild, path.top());
}
```

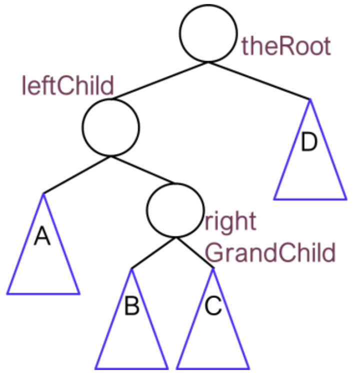
Rotação Simples à esquerda

```
// Performs a single left rotation rooted at theRoot.
// Every ancestor of theRoot is stored in the stack,
// which is not empty.
protected void rotateLeft( BSTNode<K,V> theRoot,
   BSTNode<K,V> leftChild, Stack<PathStep<K,V>> path ){

   theRoot.setLeft( leftChild.getRight() );
   leftChild.setRight(theRoot);
   this.linkSubtree(leftChild, path.top());
}
```



Rotação Dupla à esquerda

```
// Performs a double left rotation rooted at theRoot.
// Every ancestor of theRoot is stored in the stack,
// which is not empty.
protected void rotateLeft( BSTNode<K,V> theRoot,
   BSTNode<K,V> leftChild, BSTNode<K,V> rightGrandchild,
   Stack<PathStep<K,V>> path ) {

   leftChild.setRight( rightGrandchild.getLeft() );
   theRoot.setLeft( rightGrandchild.getRight() );
   rightGrandchild.setLeft(leftChild);
   rightGrandchild.setRight(theRoot);
   this.linkSubtree(rightGrandchild, path.top());
}
```



Classe Árvore AVL

```
package dataStructures;

public class AVLTree<K extends Comparable<K>, V>
             extends AdvancedBSTree<K,V>{

// Apenas dois métodos públicos: insert e remove.

................................

}
```

Classe Árvore AVL (1)

```java
// If there is an entry in the dictionary whose key is the specified key,
// replaces its value by the specified value and returns the old value;
// otherwise, inserts the entry (key, value) and returns null.
public V insert( K key, V value ){

   Stack<PathStep<K,V>> path = new StackInList<PathStep<K,V>>();
   BSTNode<K,V> node = this.findNode(key, path);
   if ( node == null ){
      AVLNode<K,V> newLeaf = new AVLNode<K,V>(key, value);
      this.linkSubtree(newLeaf, path.top());
      currentSize++;
      this.reorganizeIns(path);
      return null;
   }
   else {
      V oldValue = node.getValue();
      node.setValue(value);
      return oldValue;
   }
}
```

Classe Árvore AVL (2)

```
// Every ancestor of the new leaf is stored in the stack,
// which is not empty.
protected void reorganizeIns( Stack<PathStep<K,V>> path ){

    boolean grew = true;
    PathStep<K,V> lastStep = path.pop();
    AVLNode<K,V> parent = (AVLNode<K,V>) lastStep.parent;
    while ( grew && parent != null ){
        if ( lastStep.isLeftChild ) // parent's left subtree has grown.
            switch ( parent.getBalance() )
            { case 'L': ... case 'E':... case 'R': ... }
        else // parent's right subtree has grown.
            switch ( parent.getBalance() )
            { case 'L': ... case 'E': ... case 'R': ... }
        lastStep = path.pop();
        parent = (AVLNode<K,V>) lastStep.parent;
    }
}
```

Classe Árvore AVL (3)

```
if ( lastStep.isLeftChild ) // parent's left subtree has grown.
   switch ( parent.getBalance() ) {
      case 'L': this.rebalanceInsLeft(parent, path);  Equilibrar a árvore à esquerda
                grew = false; break;
      case 'E': parent.setBalance('L'); break;
      case 'R': parent.setBalance('E');
                grew = false; break;
   }
else // parent's right subtree has grown.
   switch ( parent.getBalance() ){
      case 'L': parent.setBalance('E');
                grew = false; break;
      case 'E': parent.setBalance('R'); break;
      case 'R': this.rebalanceInsRight(parent, path);  Equilibrar a árvore à direita
                grew = false; break;
   }
```

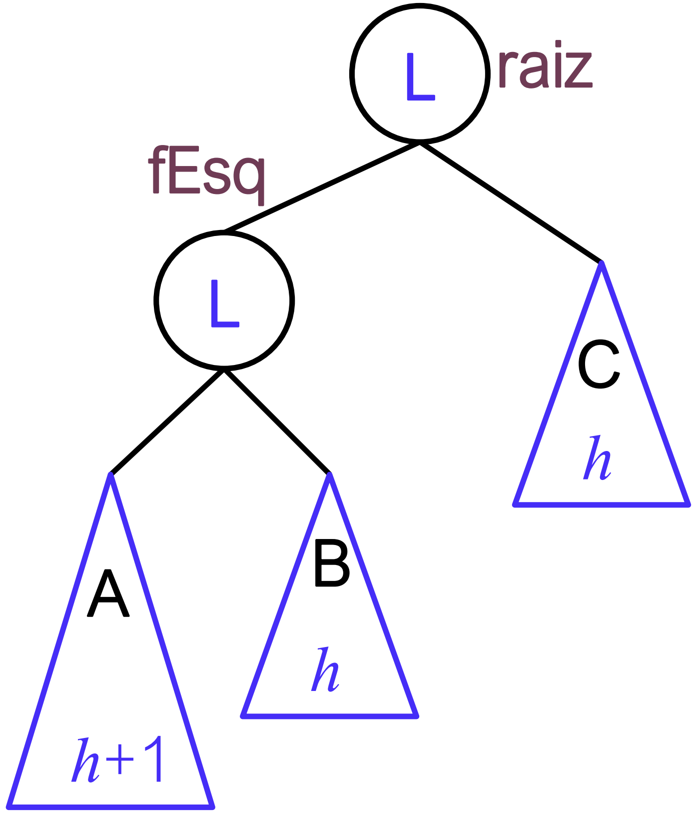
Classe Árvore AVL (4)

```java
// Every ancestor of node is stored in the stack, which is not empty.
// height( node.getLeft() ) - height( node.getRight() ) = 2.
protected void rebalanceInsLeft( AVLNode<K,V> node,
                                 Stack<PathStep<K,V>> path ) {

   AVLNode<K,V> leftChild = (AVLNode<K,V>) node.getLeft();
   switch ( leftChild.getBalance() ) {
      case 'L':
            this.rotateLeft1L(node, leftChild, path);
            break;
      // case 'E':
      //    Impossible.
      case 'R':
            this.rotateLeft2(node, leftChild, path);
            break;
   }
}
```

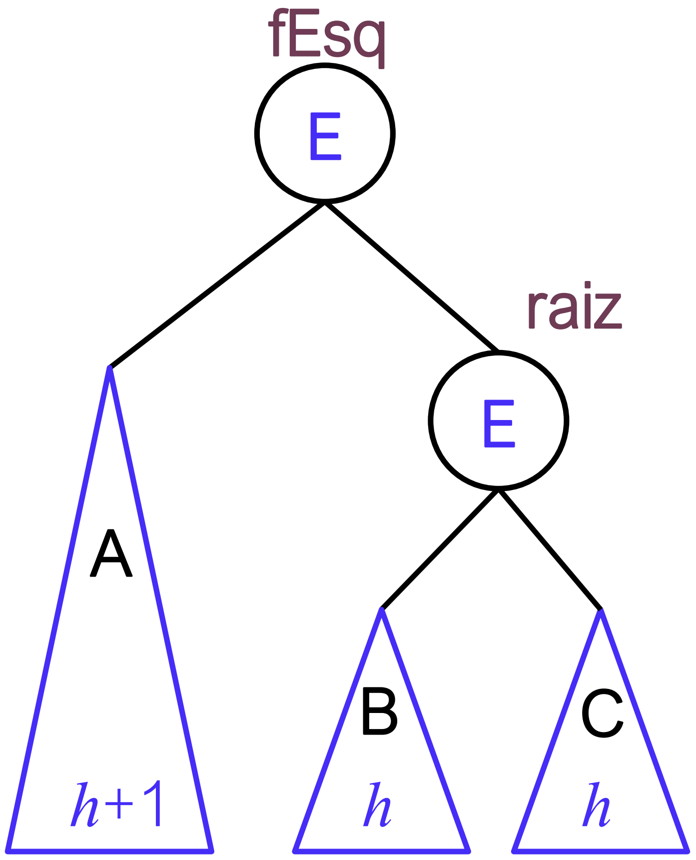
Rotação simples à esquerda

Rotação Dupla à esquerda

Inserção L- L / Rotação simples à esquerda



Ordenação: A fEsq B raiz C



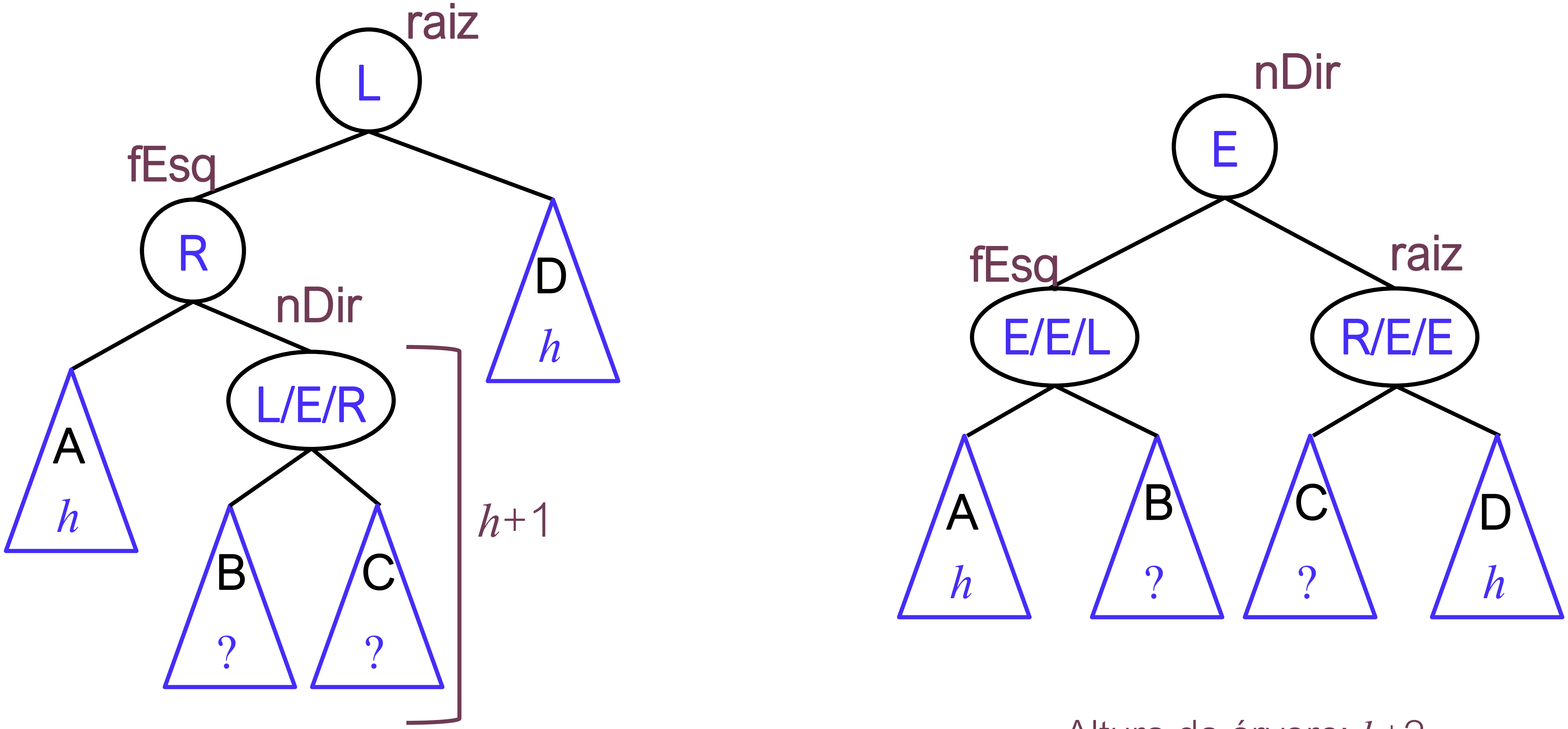
Altura da árvore: h+2

Classe Árvore AVL (5)

```
// Performs a single left rotation rooted at theRoot,
// when the balance factor of its leftChild is 'L'.
// Every ancestor of theRoot is stored in the stack, which is not empty.
// height( node.getLeft() ) - height( node.getRight() ) = 2.
protected void rotateLeft1L( AVLNode<K,V> theRoot, AVLNode<K,V> leftChild,
                             Stack<PathStep<K,V>> path ) {

   theRoot.setBalance('E');
   leftChild.setBalance('E');
   this.rotateLeft(theRoot, leftChild, path);
}
```

Rotação simples à esquerda - criação das ligações

Inserção L-R / Rotação dupla à esquerda



Ordenação: A fEsqB nDirC raizD

Altura da árvore: $h+2$

Classe Árvore AVL (6)

```
// Performs a double left rotation rooted at theRoot.
// Every ancestor of theRoot is stored in the stack, which is not empty.
/ height( node.getLeft() ) - height( node.getRight() ) = 2.
protected void rotateLeft2( AVLNode<K,V> theRoot, AVLNode<K,V> leftChild,
                            Stack<PathStep<K,V>> path ) {

   AVLNode<K,V> rightGrandchild = (AVLNode<K,V>) leftChild.getRight();
   switch ( rightGrandchild.getBalance() ) {
      case 'L': leftChild.setBalance('E');
                theRoot.setBalance('R'); break;
      case 'E': leftChild.setBalance('E');
                theRoot.setBalance('E'); break;
      case 'R': leftChild.setBalance('L');
                theRoot.setBalance('E'); break;
   }
   rightGrandchild.setBalance('E');
   this.rotateLeft(theRoot, leftChild, rightGrandchild, path);
}
```

Rotação dupla à esquerda - criação das ligações

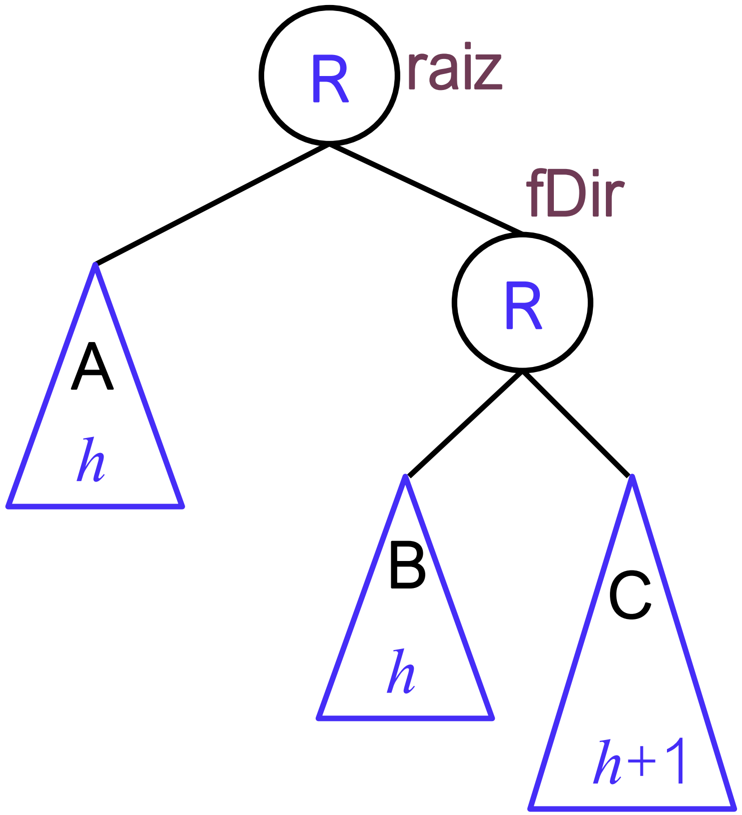
Classe Árvore AVL (7)

```
// Every ancestor of node is stored in the stack, which is not empty.
// height( node.getRight() ) - height( node.getLeft() ) = 2.
protected void rebalanceInsRight( AVLNode<K,V> node,
                                  Stack<PathStep<K,V>> path ) {
   AVLNode<K,V> rightChild = (AVLNode<K,V>) node.getRight();
   switch ( rightChild.getBalance() ) {
      case 'L':
            this.rotateRight2(node, rightChild, path);
            break;
      // case 'E':
      //     Impossible.
      case 'R':
            this.rotateRight1R(node, rightChild, path);
            break;
   }
}
```

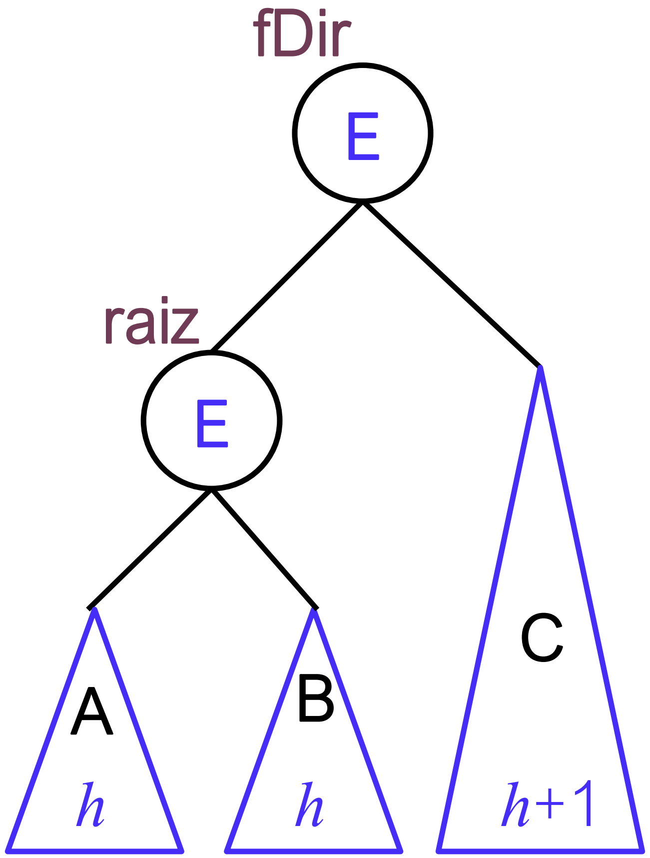
Rotação dupla à direita

Rotação simples à Direita

Inserção R-R / Rotação simples à direita



Ordenação: A raiz B fDir C



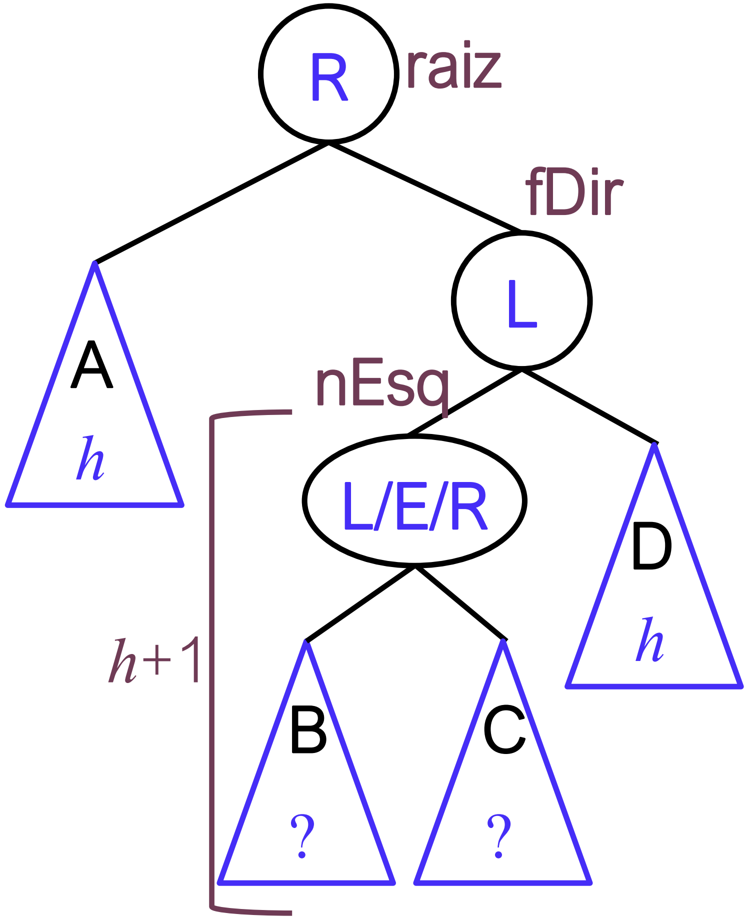
Altura da árvore: h+2

Classe Árvore AVL (8)

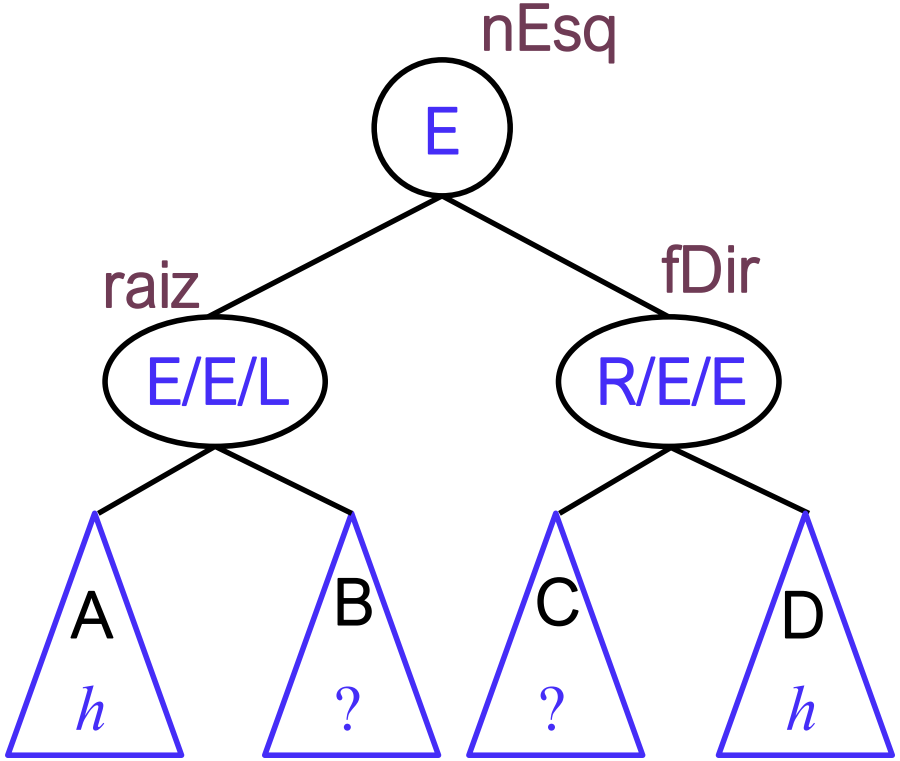
```
// Performs a single right rotation rooted at theRoot,
// when the balance factor of its rightChild is 'R'.
// Every ancestor of theRoot is stored in the stack, which is not empty.
// height( node.getRight() ) - height( node.getLeft() ) = 2.
protected void rotateRight1R( AVLNode<K,V> theRoot,
   AVLNode<K,V> rightChild, Stack<PathStep<K,V>> path ) {

   theRoot.setBalance('E');
   rightChild.setBalance('E');
   this.rotateRight(theRoot, rightChild, path);
}
```

Rotação simples à direita- criação das ligações

Inserção R-L / Rotação dupla à direita



<u>Ordenação:</u> A raiz B nEsq C fDir D



Altura da árvore: h+2

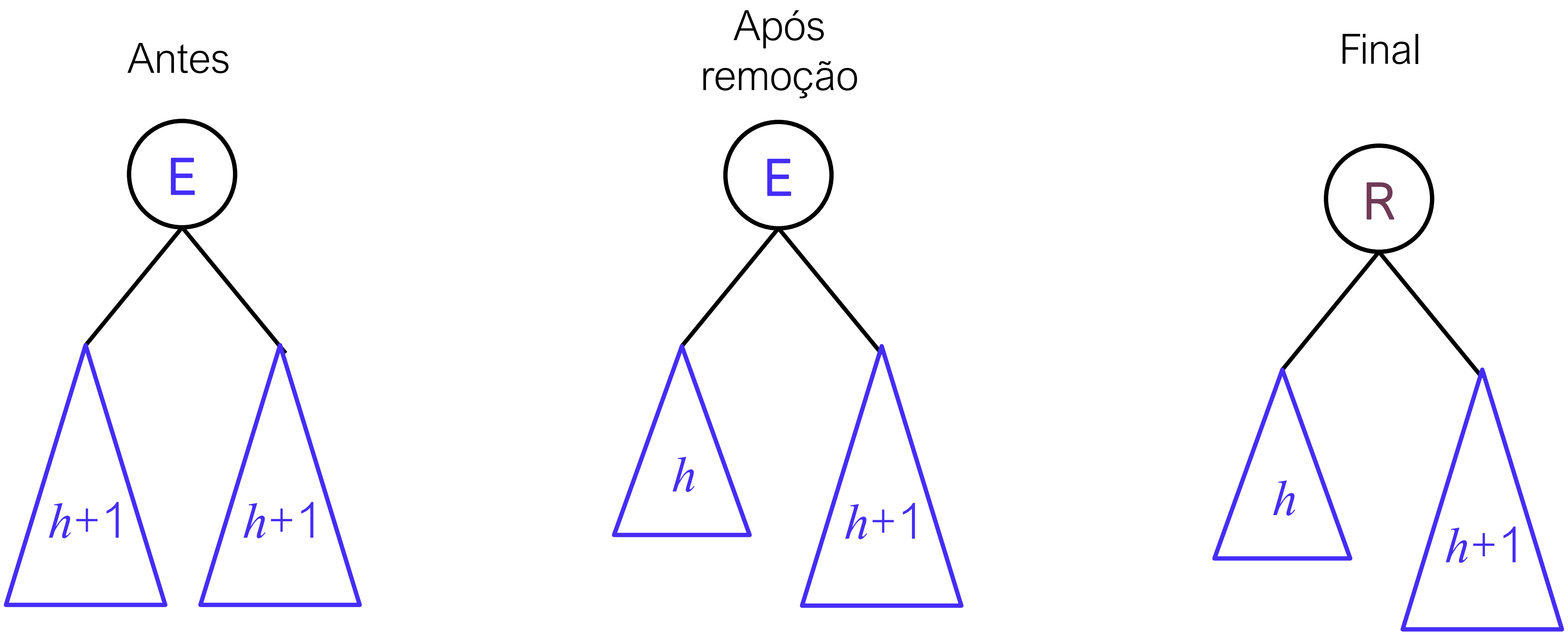
Classe Árvore AVL (9)

```java
// Performs a double right rotation rooted at theRoot.
// Every ancestor of theRoot is stored in the stack, which is not empty.
// height( node.getRight() ) - height( node.getLeft() ) = 2.
protected void rotateRight2( AVLNode<K,V> theRoot,
   AVLNode<K,V> rightChild, Stack<PathStep<K,V>> path ) {

   AVLNode<K,V> leftGrandchild = (AVLNode<K,V>) rightChild.getLeft();
   switch ( leftGrandchild.getBalance() ) {
      case 'L': theRoot.setBalance('E');
                rightChild.setBalance('R'); break;
      case 'E': theRoot.setBalance('E');
                rightChild.setBalance('E'); break;
      case 'R': theRoot.setBalance('L');
                rightChild.setBalance('E'); break;
   }
   leftGrandchild.setBalance('E');
   this.rotateRight(theRoot, rightChild, leftGrandchild, path);
}
```

Rotação dupla à direita - criação das ligações

Remoção

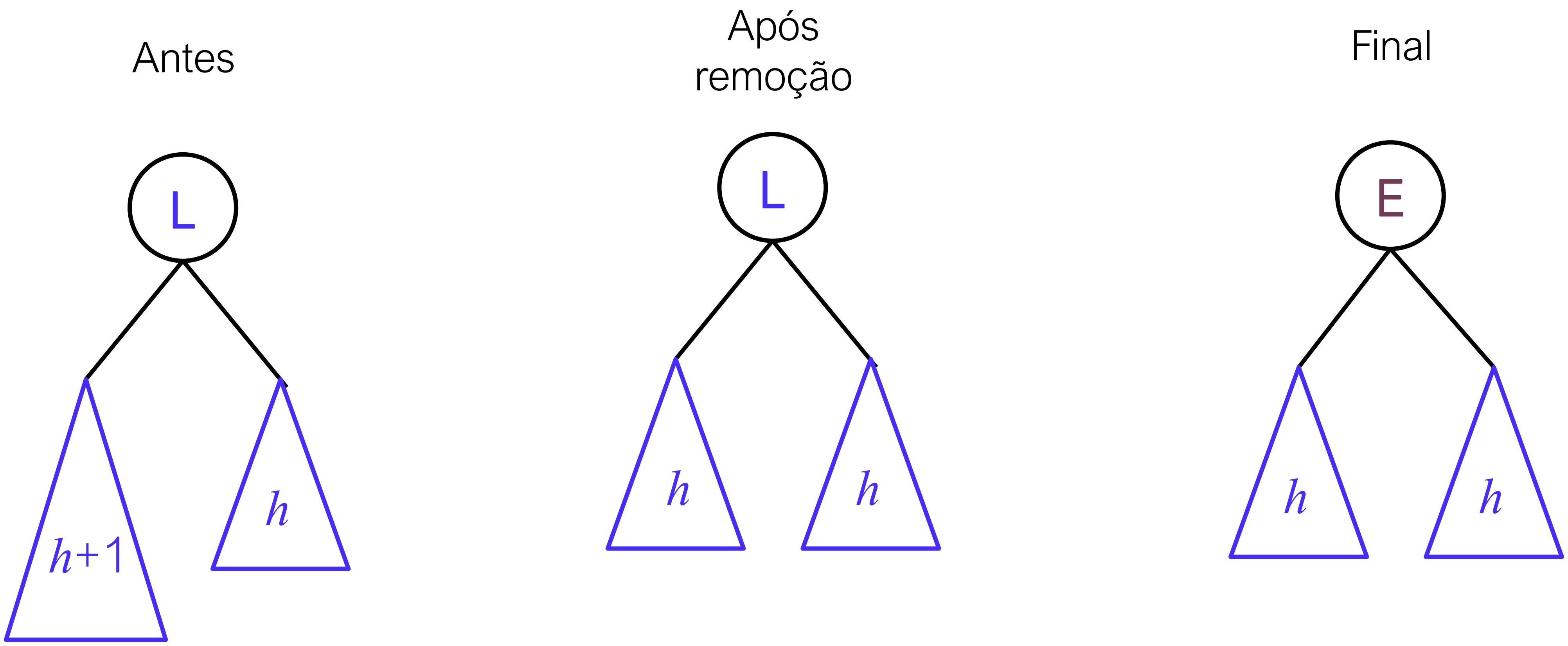
Nó E - Subárvore Esquerda diminuiu



A árvore não diminuiu

Remoção

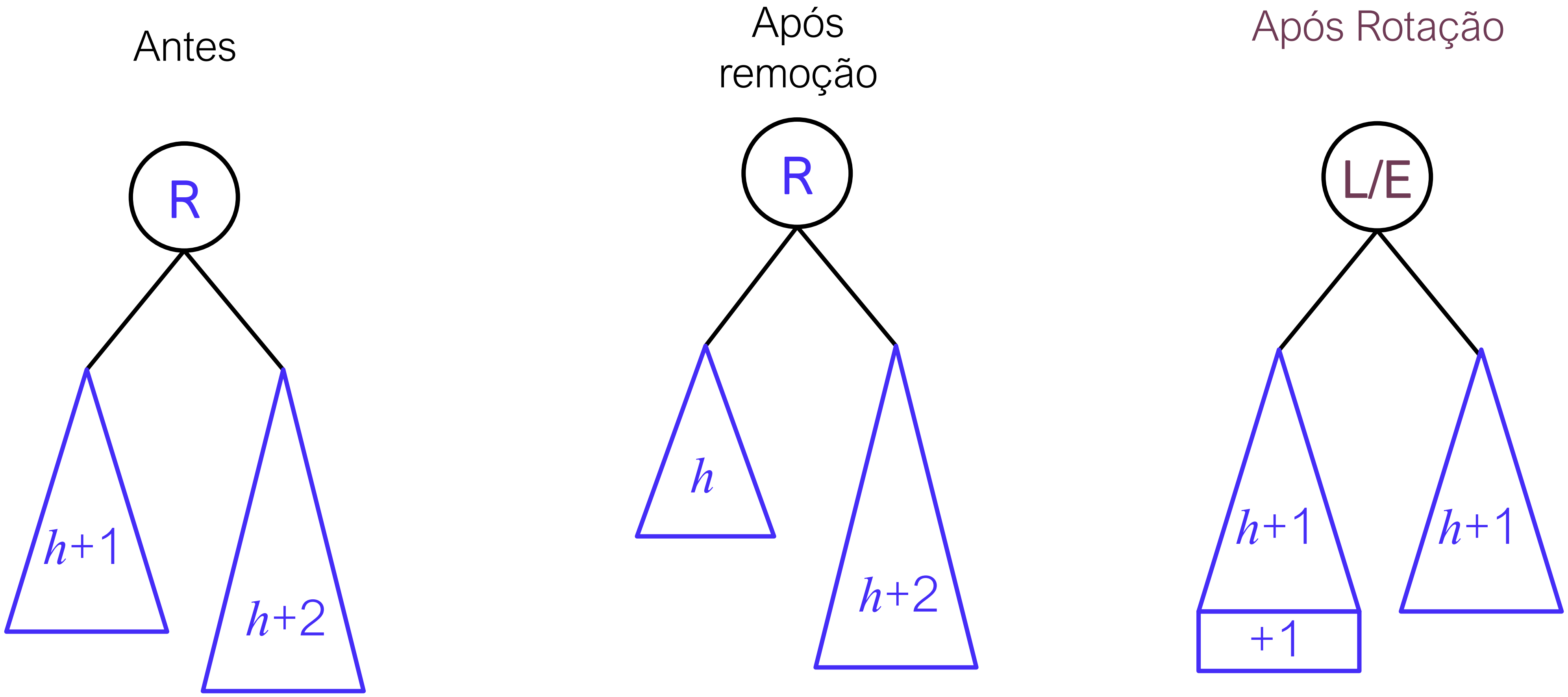
Nó L - Subárvore Esquerda Diminuiu



A árvore Diminuiu

Remoção

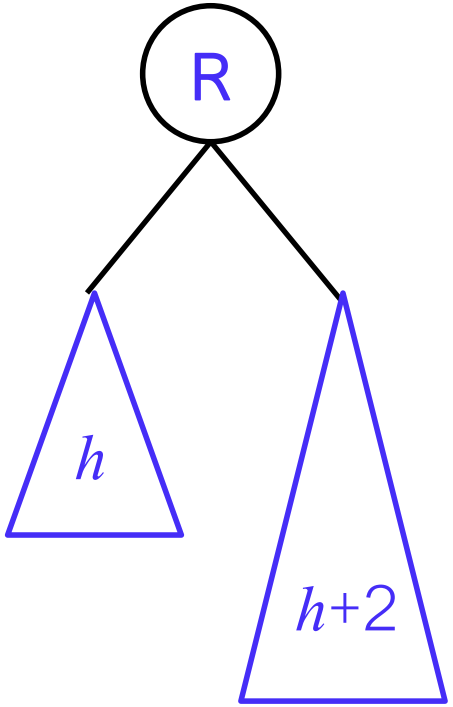
Nó R - Subárvore Esquerda Diminuiu



A árvore pode Diminuir

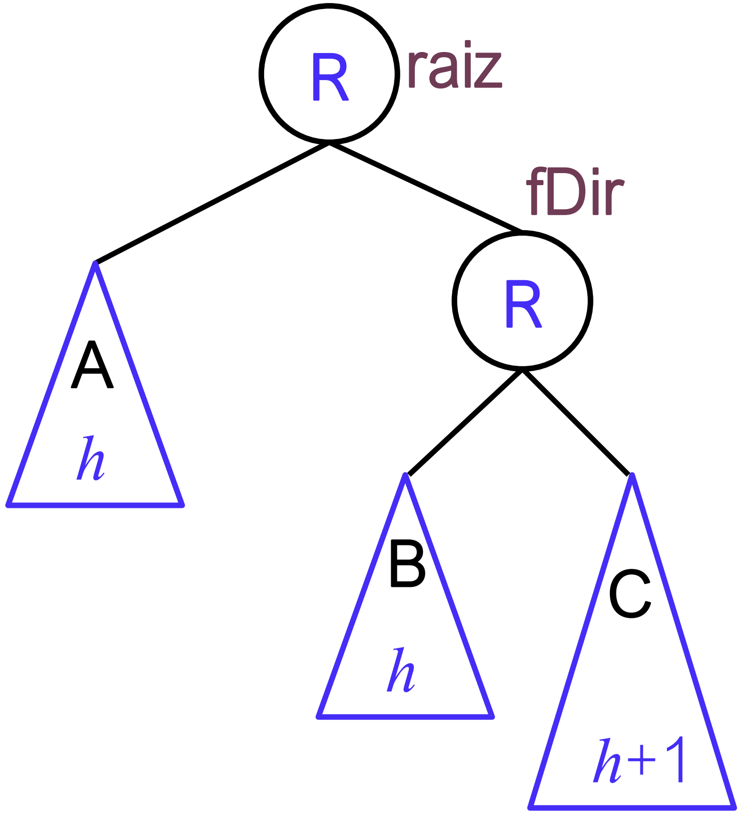
Remoção

Nó R - Subárvore Esquerda Diminuiu

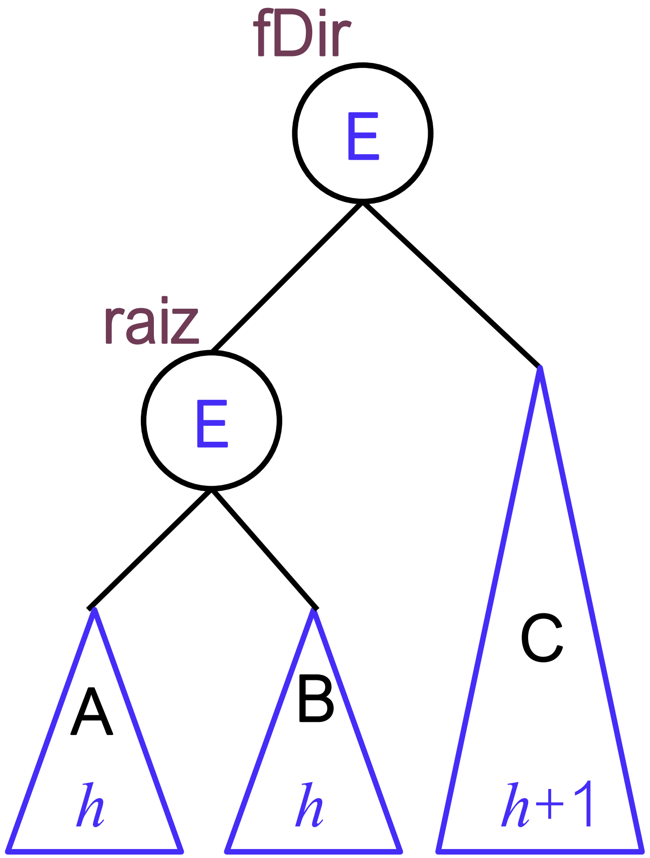


- Esta situação poderá ocorrer de 3 formas diferentes:
  - A raiz da subárvore direita é R
  - A raiz da subárvore direita é E
  - A raiz da subárvore direita é L

Remoção R-R / Rotação simples à direita

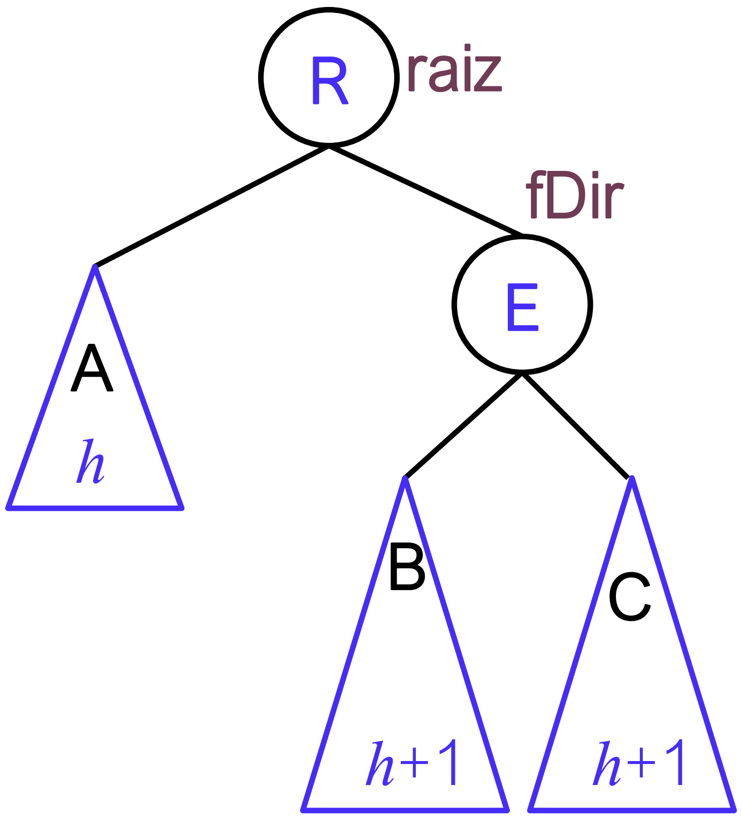


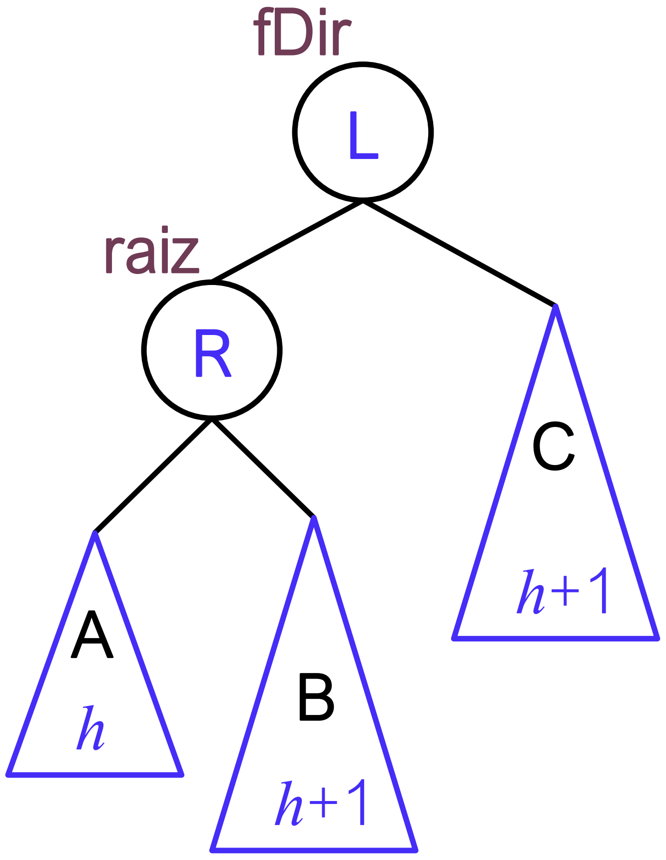
<u>Ordenação:</u> A raiz B fDir C



Altura da árvore: $h+2$
A árvore Diminuiu

Remoção R-E / Rotação simples à direita

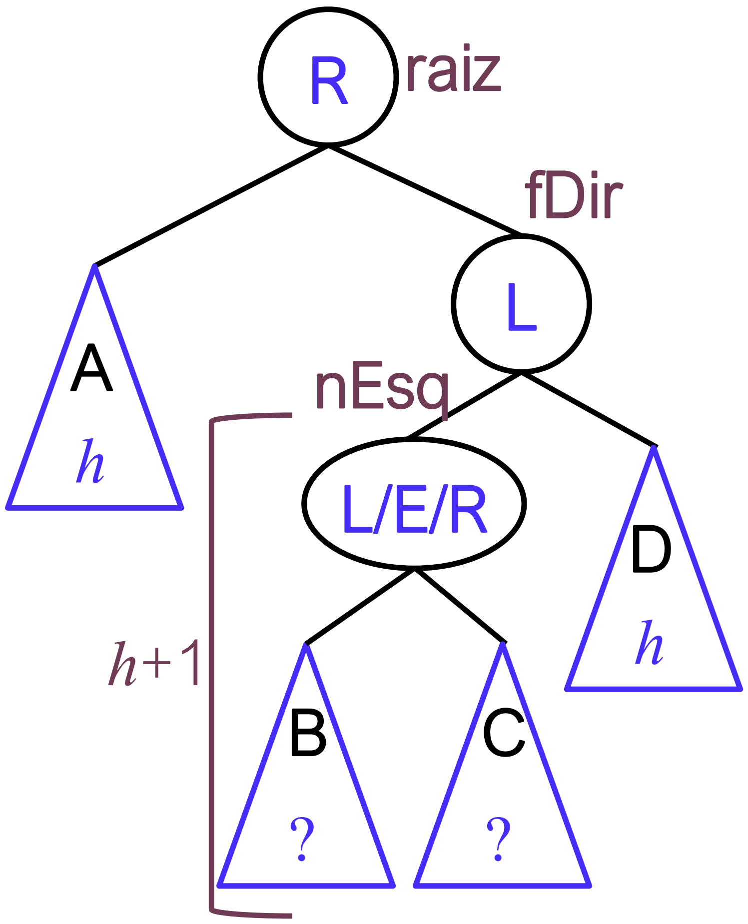


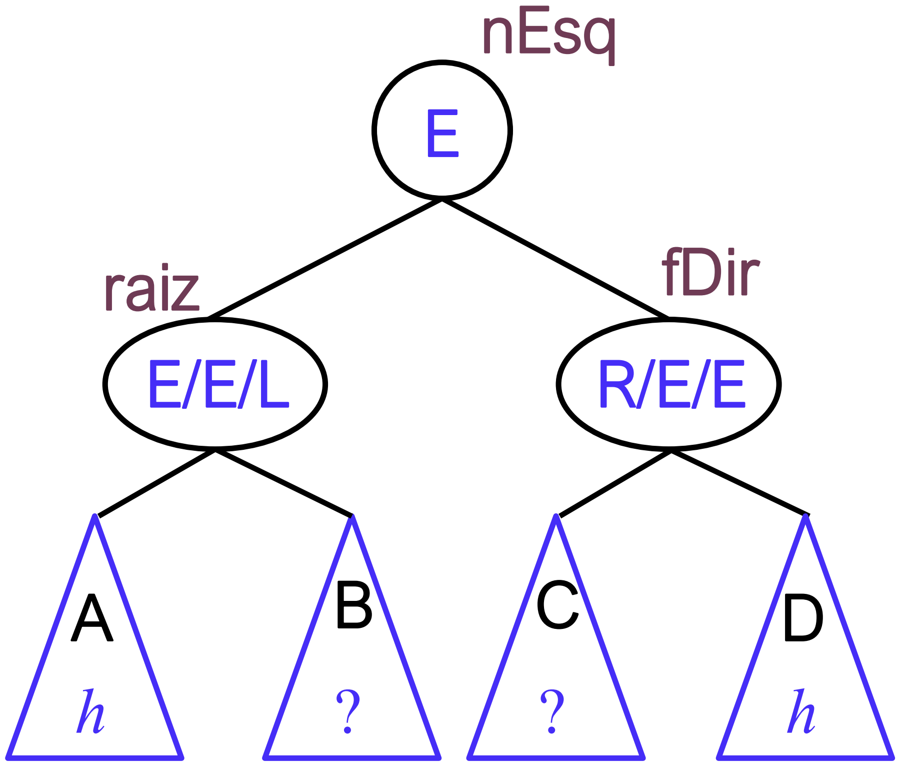


<u>Ordenação:</u> A raiz B fDir C

Altura da árvore: h+3
A árvore não Diminuiu

Remoção R-L / Rotação dupla à direita



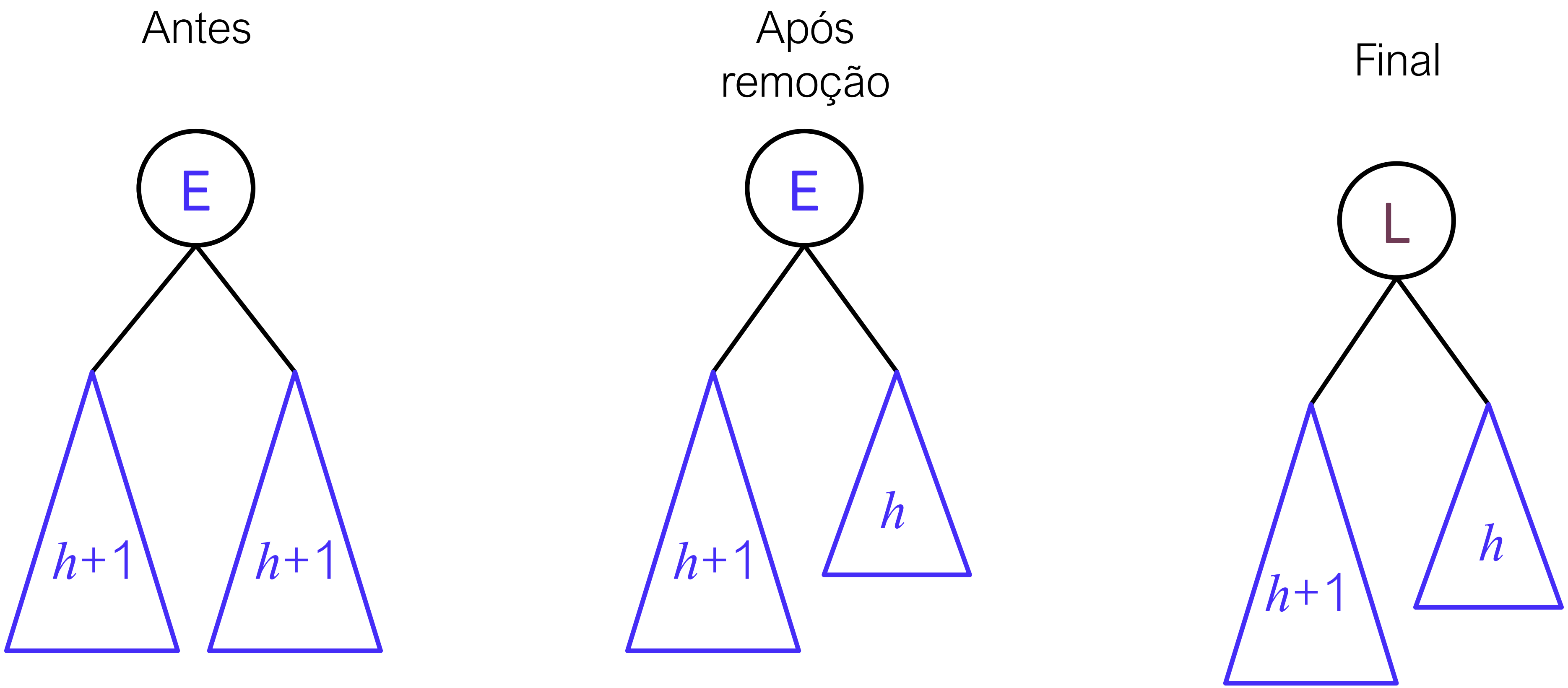


Ordenação: A raiz B nEsq C fDir D

Altura da árvore: h+2
A árvore Diminuiu

Remoção

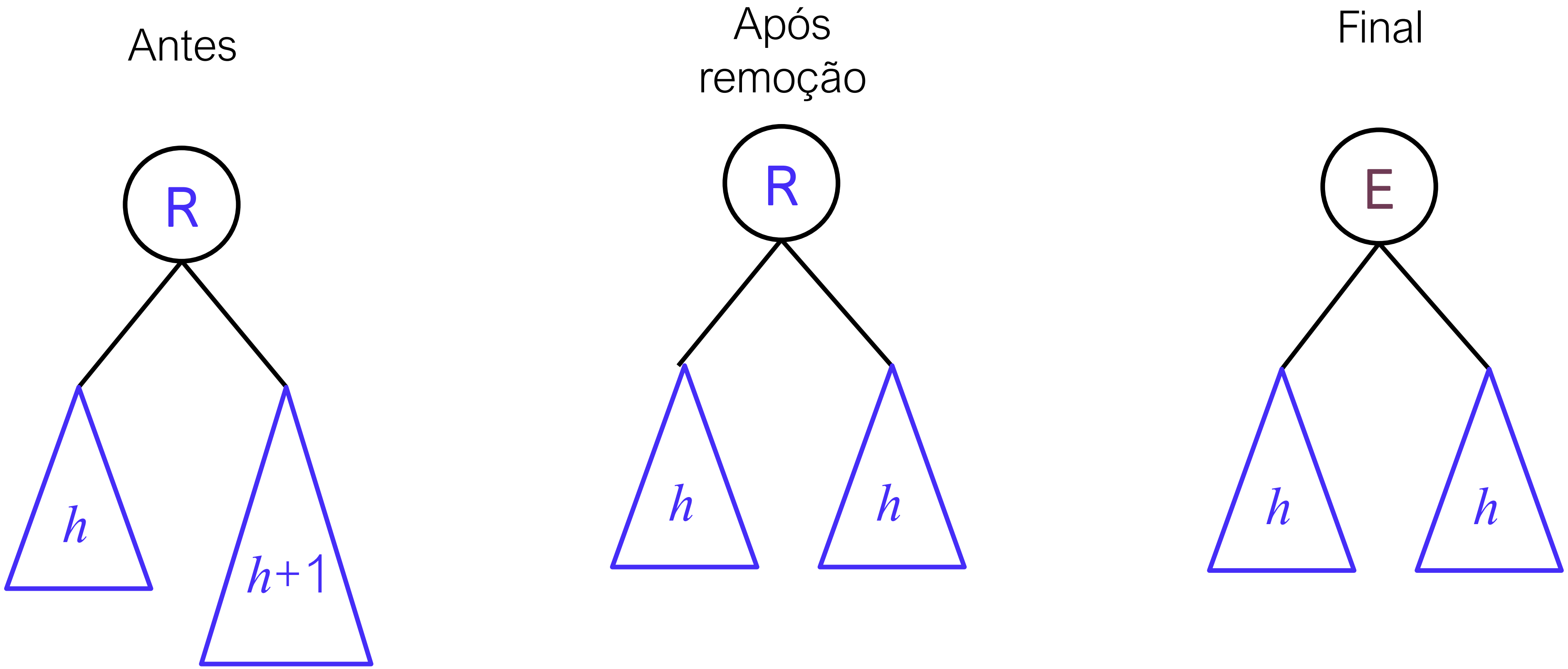
Nó E - Subárvore Direita Diminuiu



A árvore não Diminuiu

Remoção

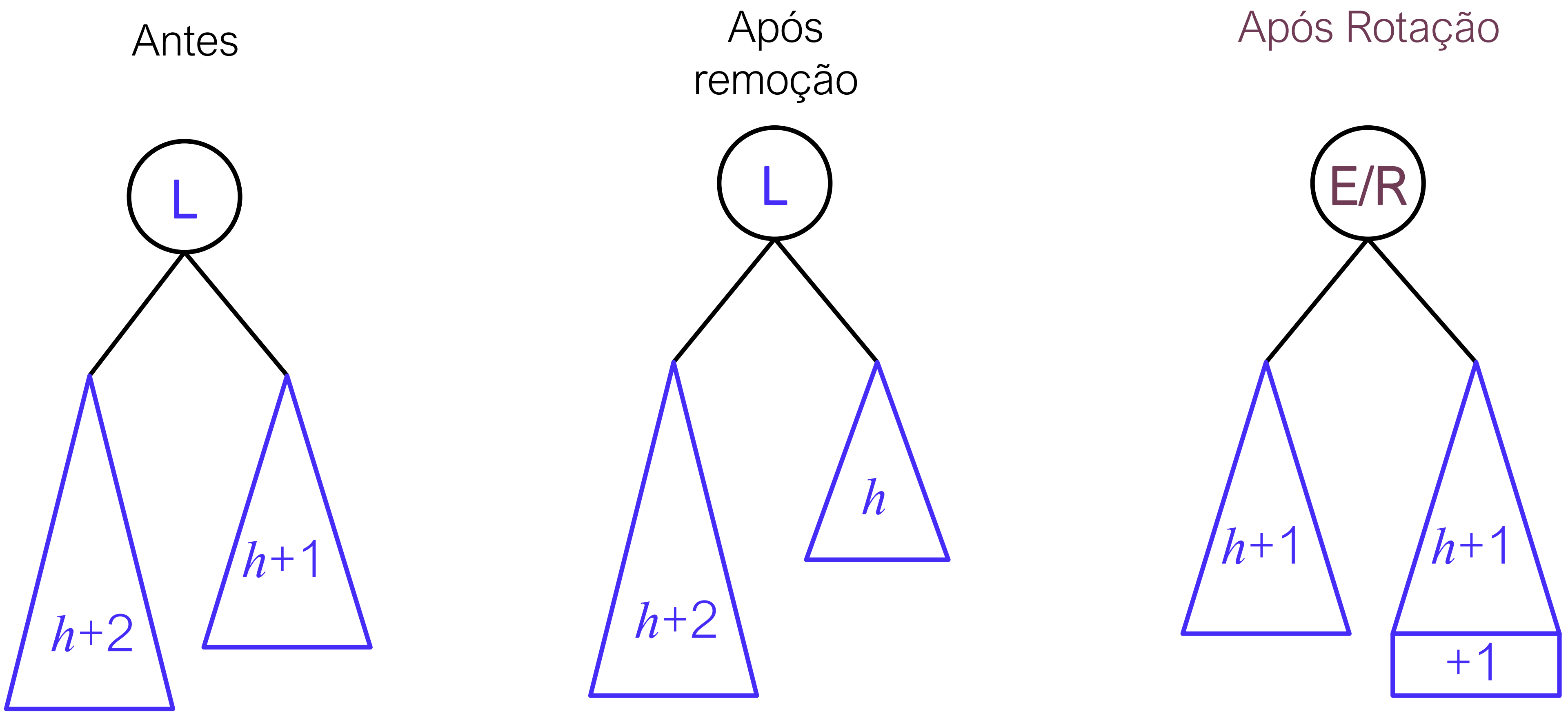
Nó R - Subárvore Direita Diminuiu



A árvore Diminuiu

Remoção

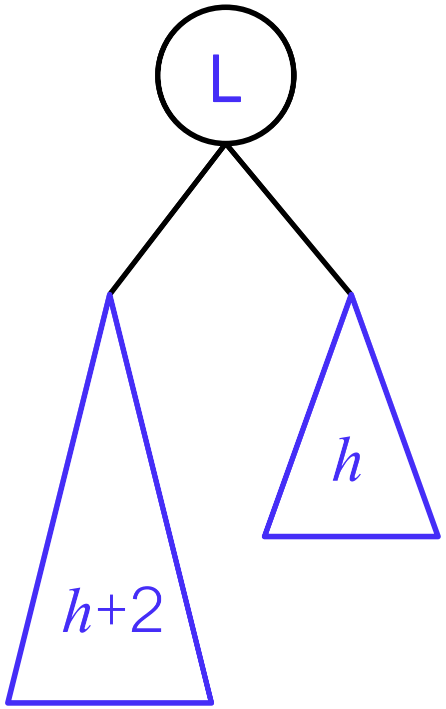
Nó L - Subárvore Direita Diminuiu



A árvore pode Diminuir

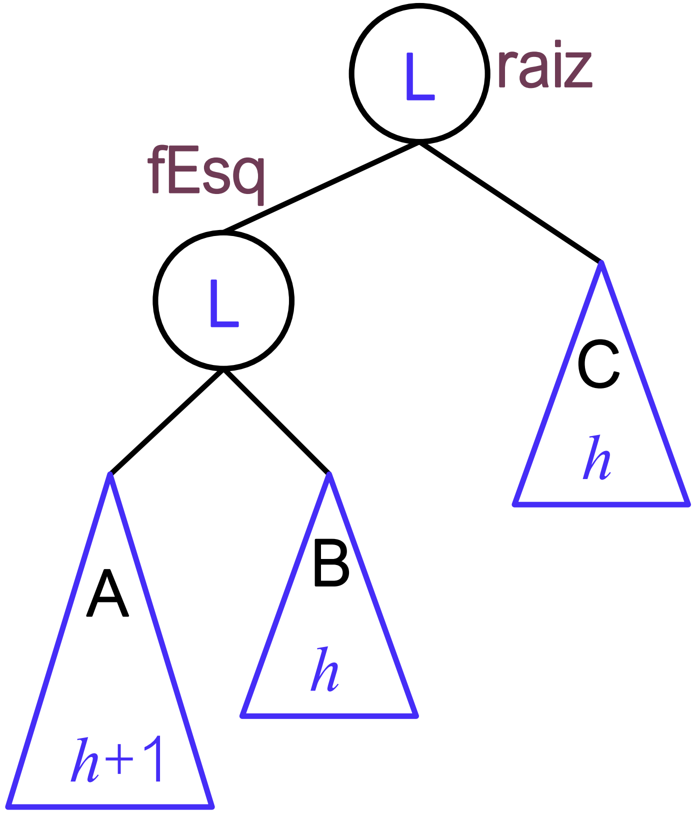
Remoção

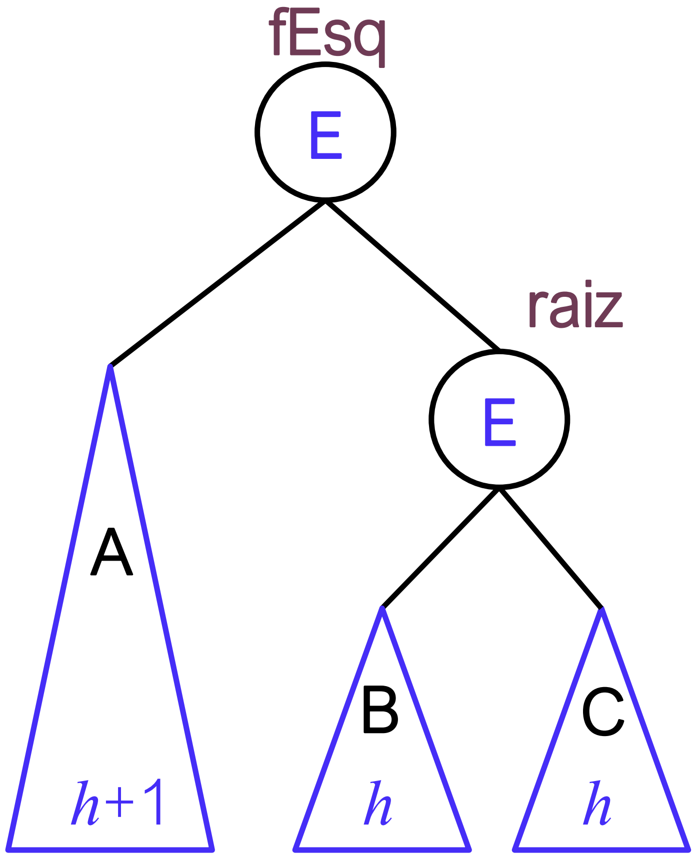
Nó L - Subárvore Direita Diminuiu



- Esta situação poderá ocorrer de 3 formas diferentes:
  - A raiz da subárvore esquerda é L
  - A raiz da subárvore esquerda é E
  - A raiz da subárvore esquerda é R

Remoção L- L / Rotação simples à esquerda

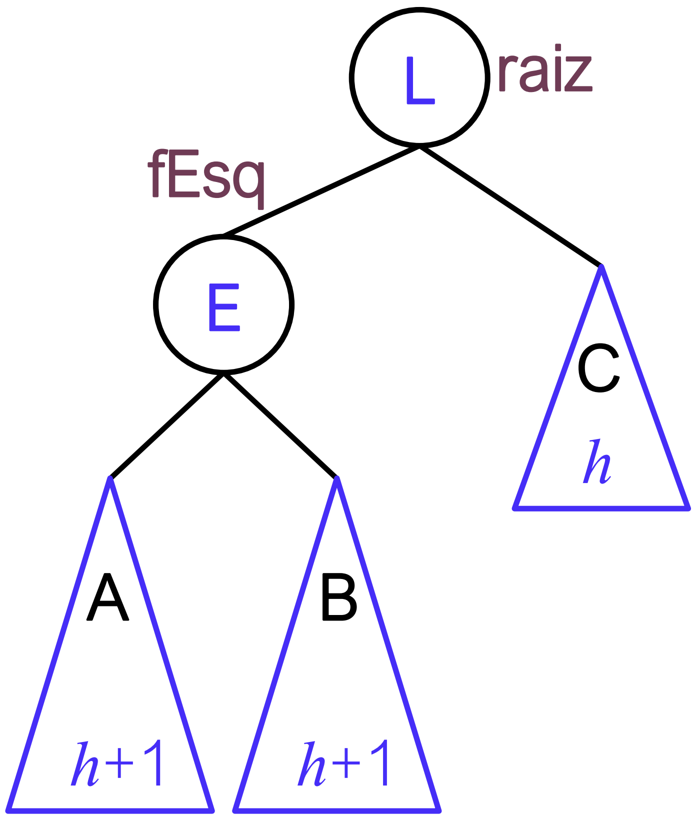


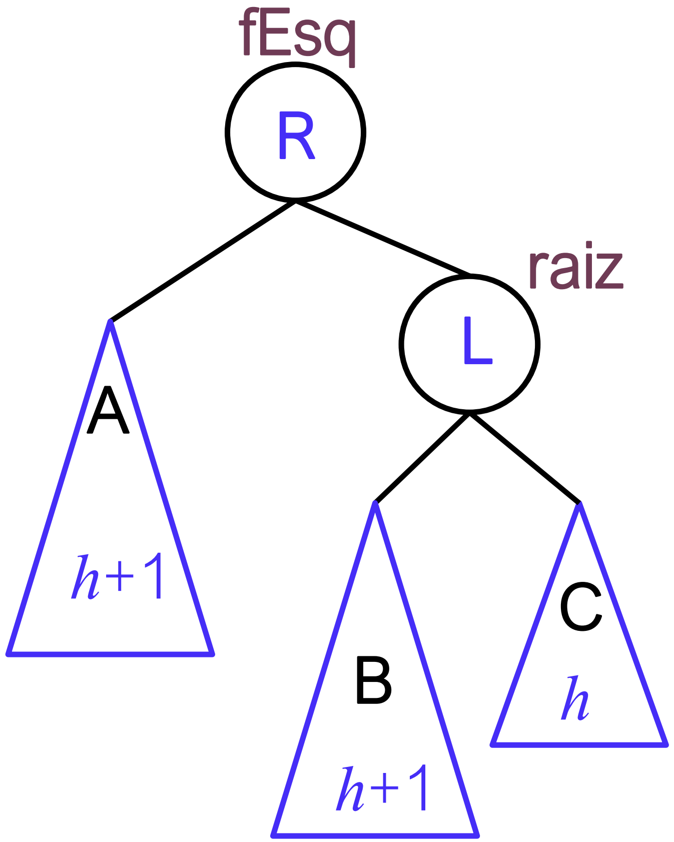


Ordenação: A fEsq B raiz C

Altura da árvore: h+2
A árvore Diminuiu

Remoção L- E / Rotação simples à esquerda

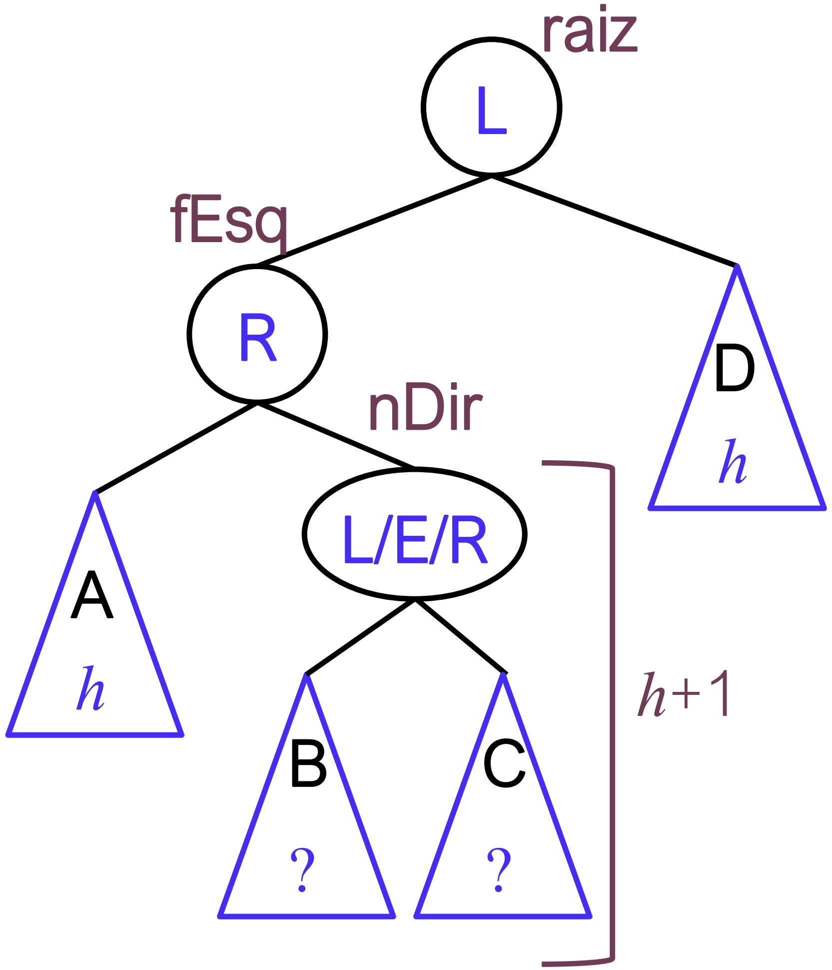




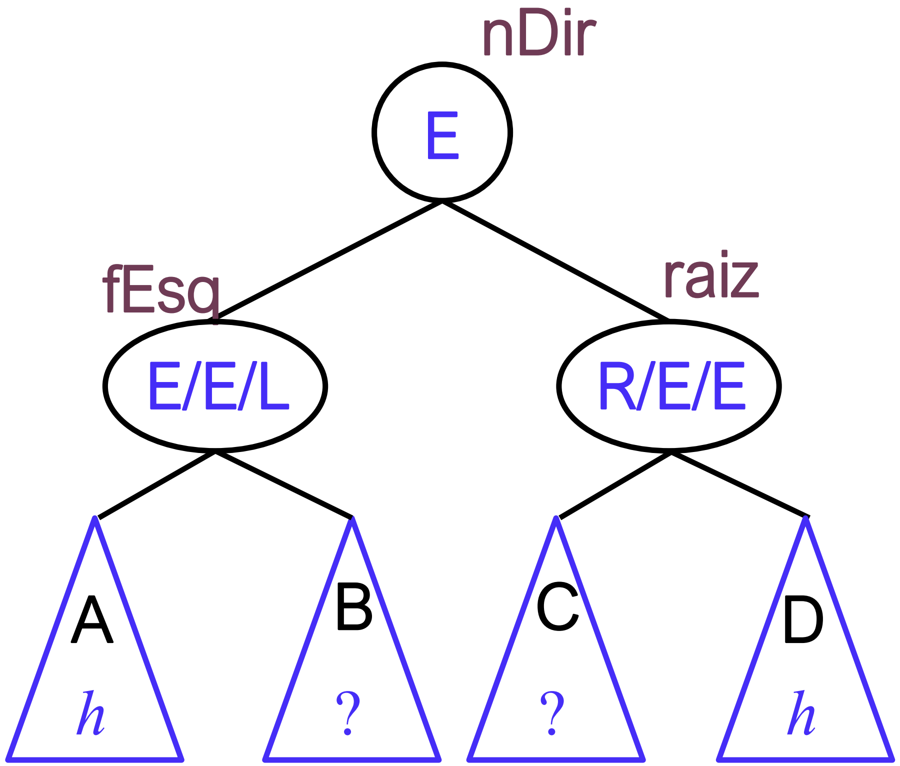
Ordenação: A fEsq B raiz C

Altura da árvore: h+3
A árvore não Diminuiu

Remoção L-R / Rotação dupla à esquerda



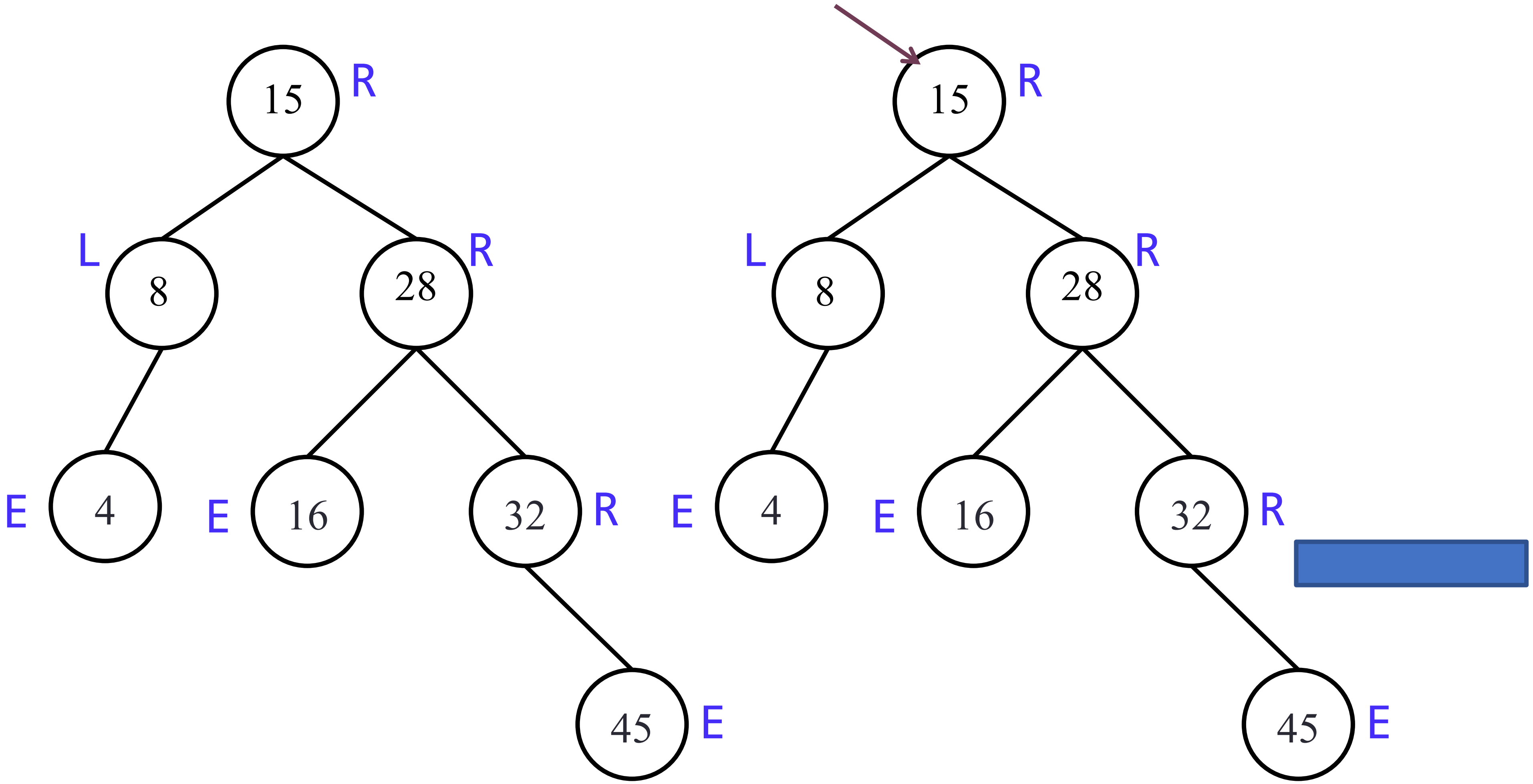
<u>Ordenação:</u> A fEsqB nDirC raizD



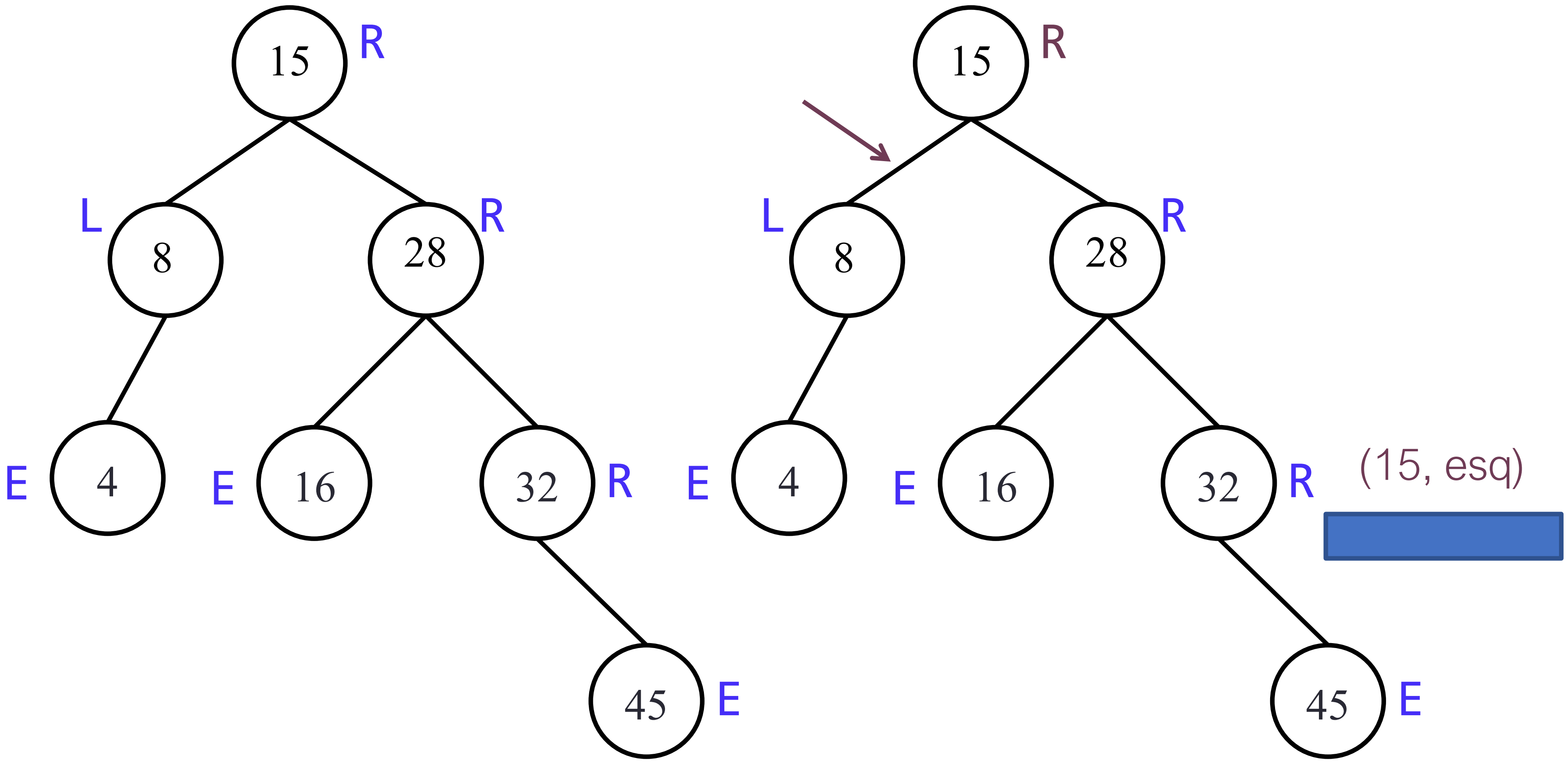
Altura da árvore: h+2

A árvore Diminuiu

Remover nó com subárvore vazia (8)

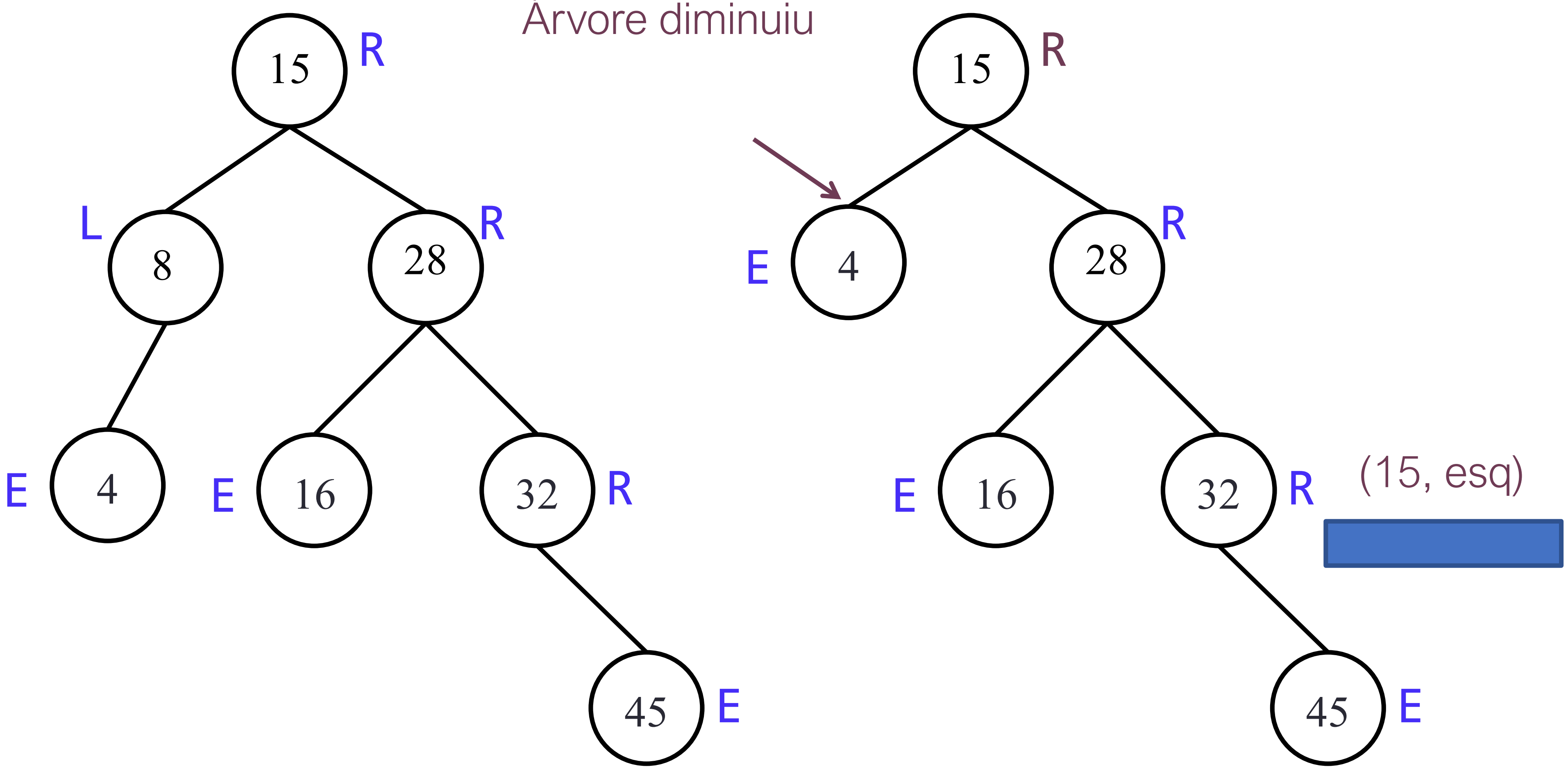


Remover nó com subárvore vazia (8)



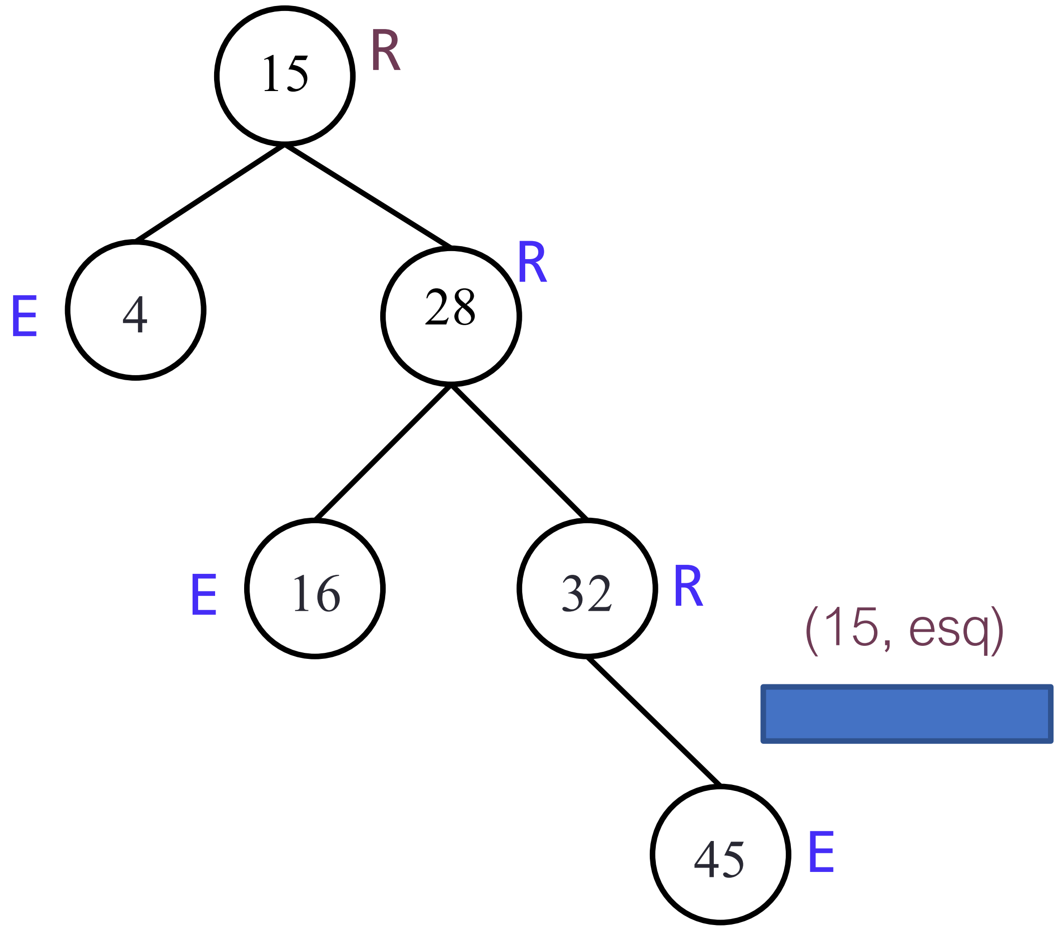
Remover nó com subárvore vazia (8)

Pai herda subárvore
Árvore diminuiu



Remover nó com subárvore vazia (8)

Passo:



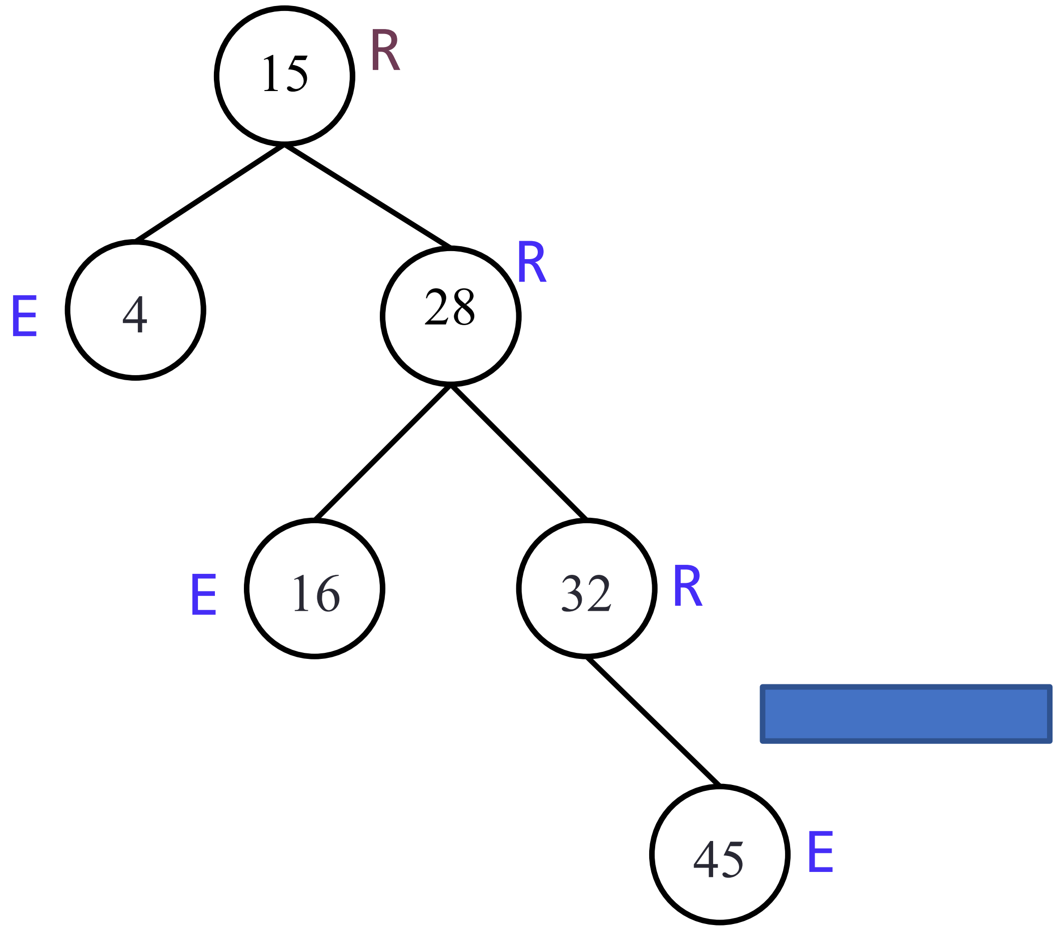
Remover nó com subárvore vazia (8)

Passo: (15, esq)

Subárvore esquerda diminuiu

R → Rotação à Direita

fDir R → Simples



Remover nó com subárvore vazia (8)

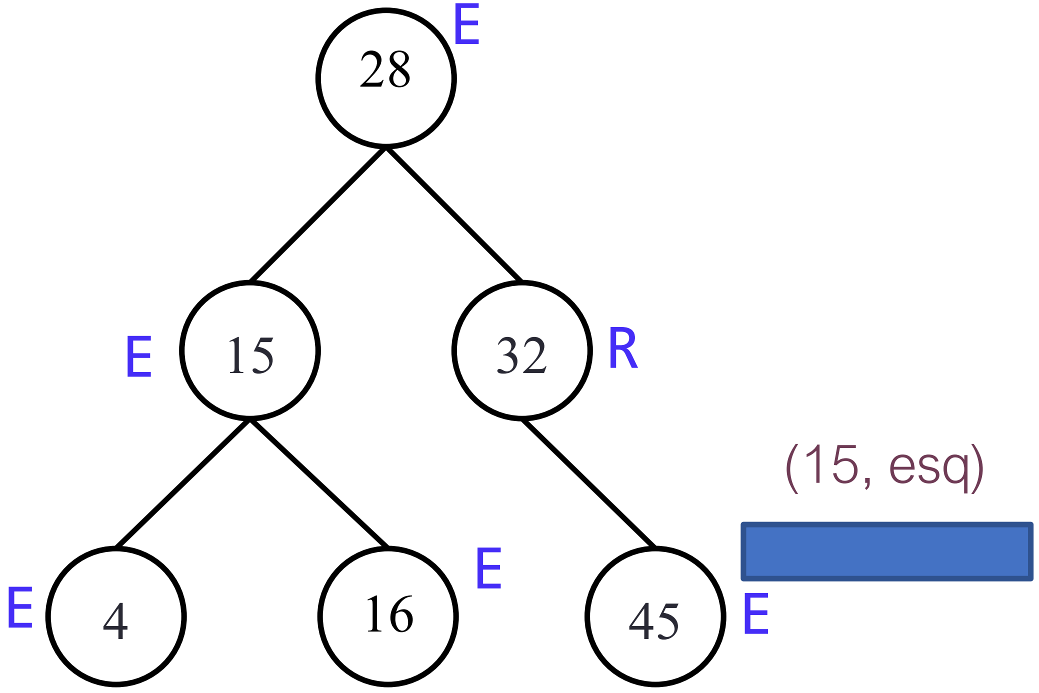
Passo: (15, esq)

Subárvore esquerda diminuiu

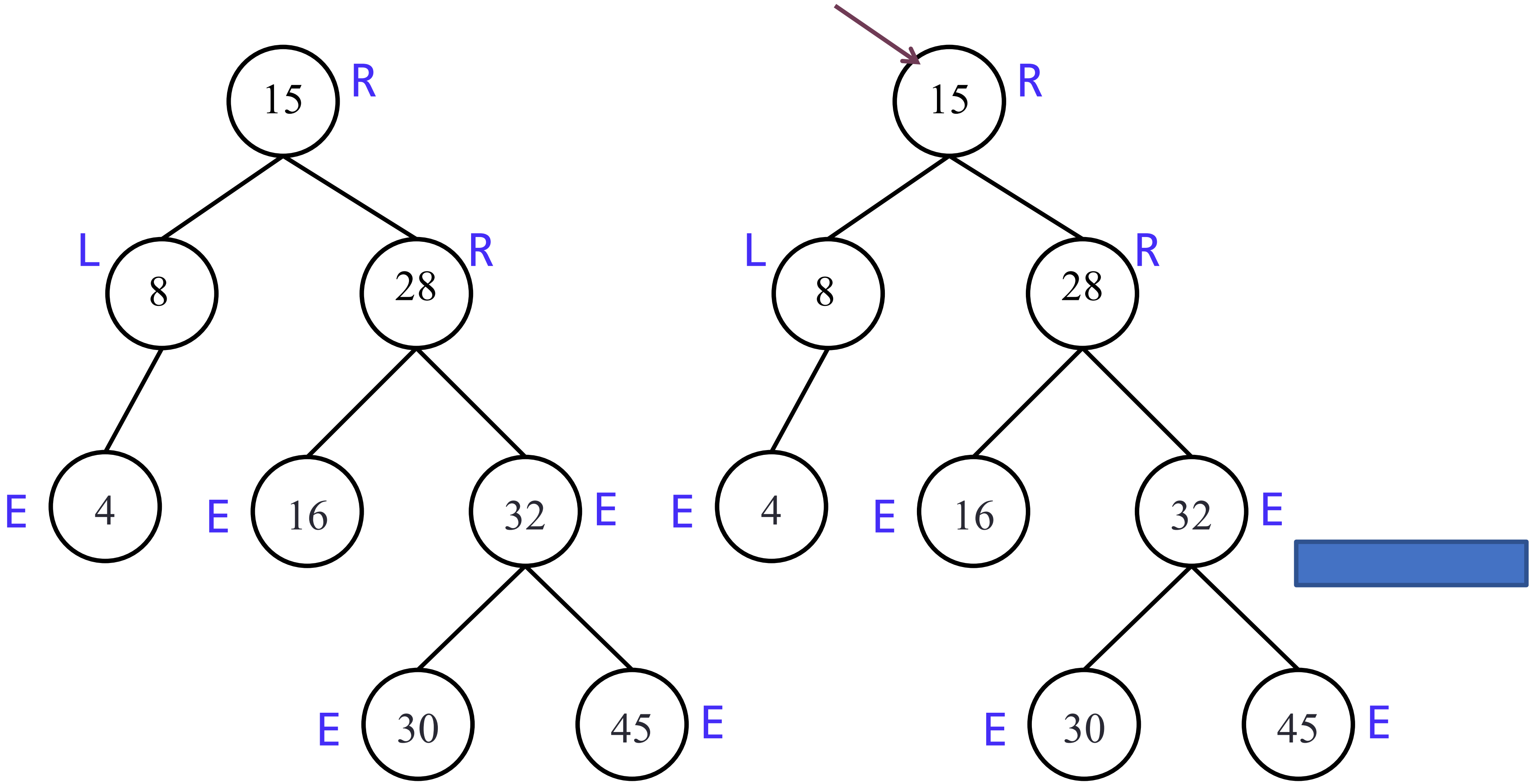
R → Rotação à Direita

fDir R → Simples

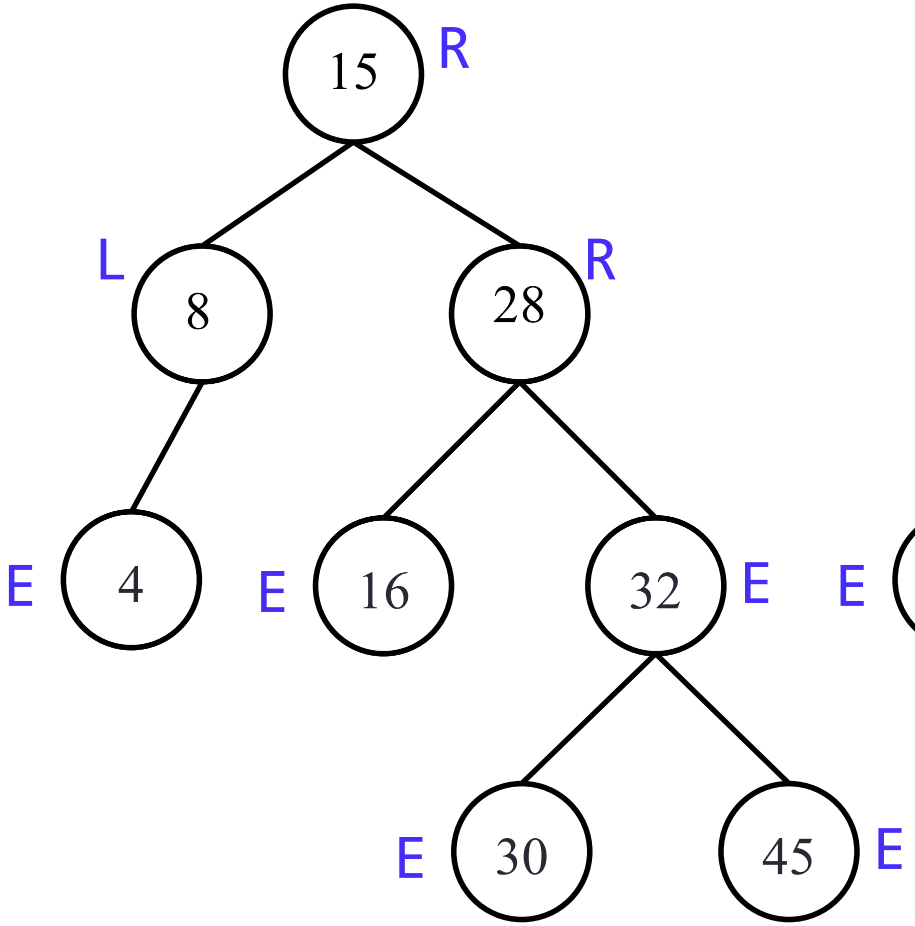
Árvore Diminuiu

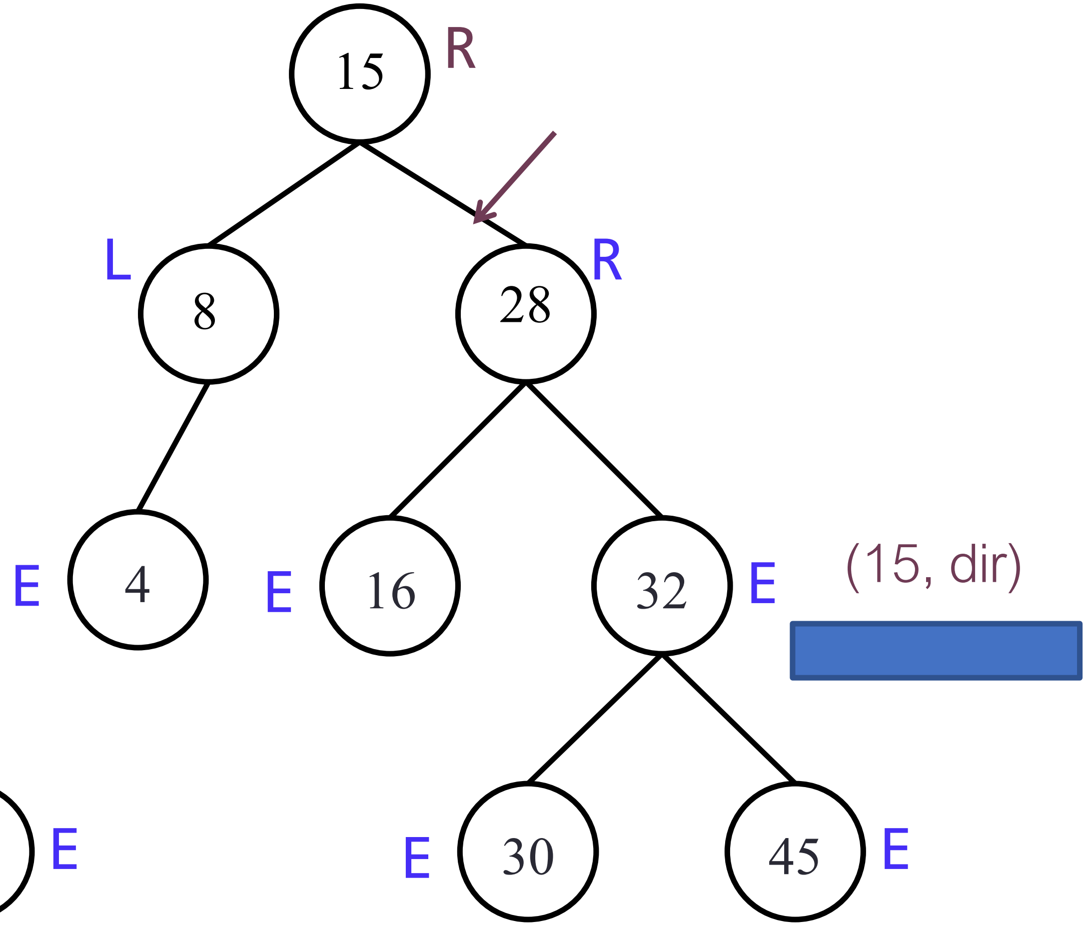


Remover nó com dois filhos (28)



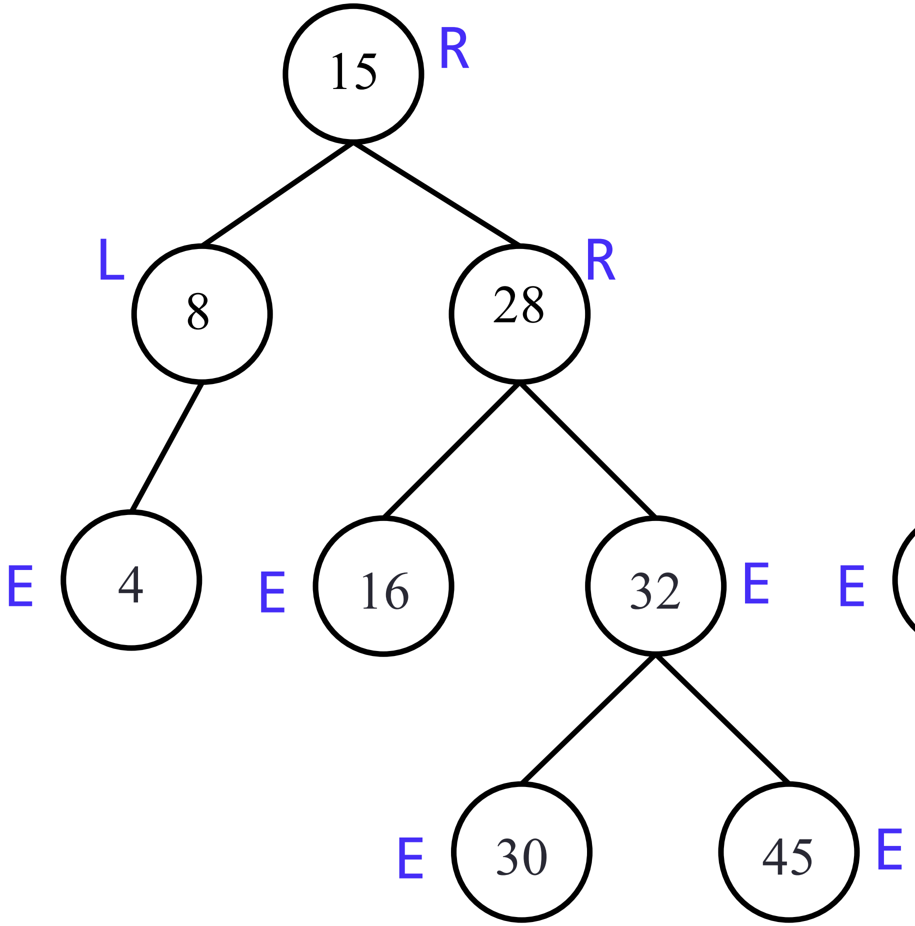
Remover nó com dois filhos (28)

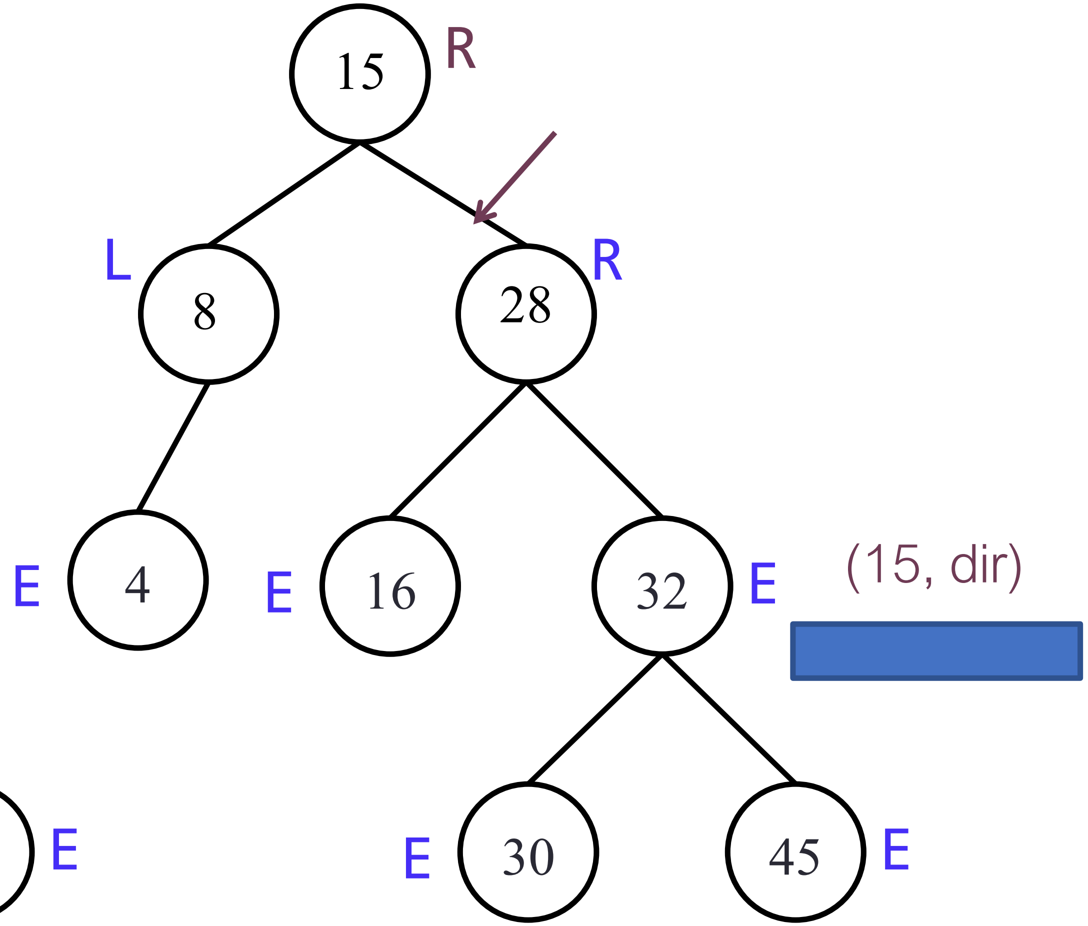




Remover nó com dois filhos (28)

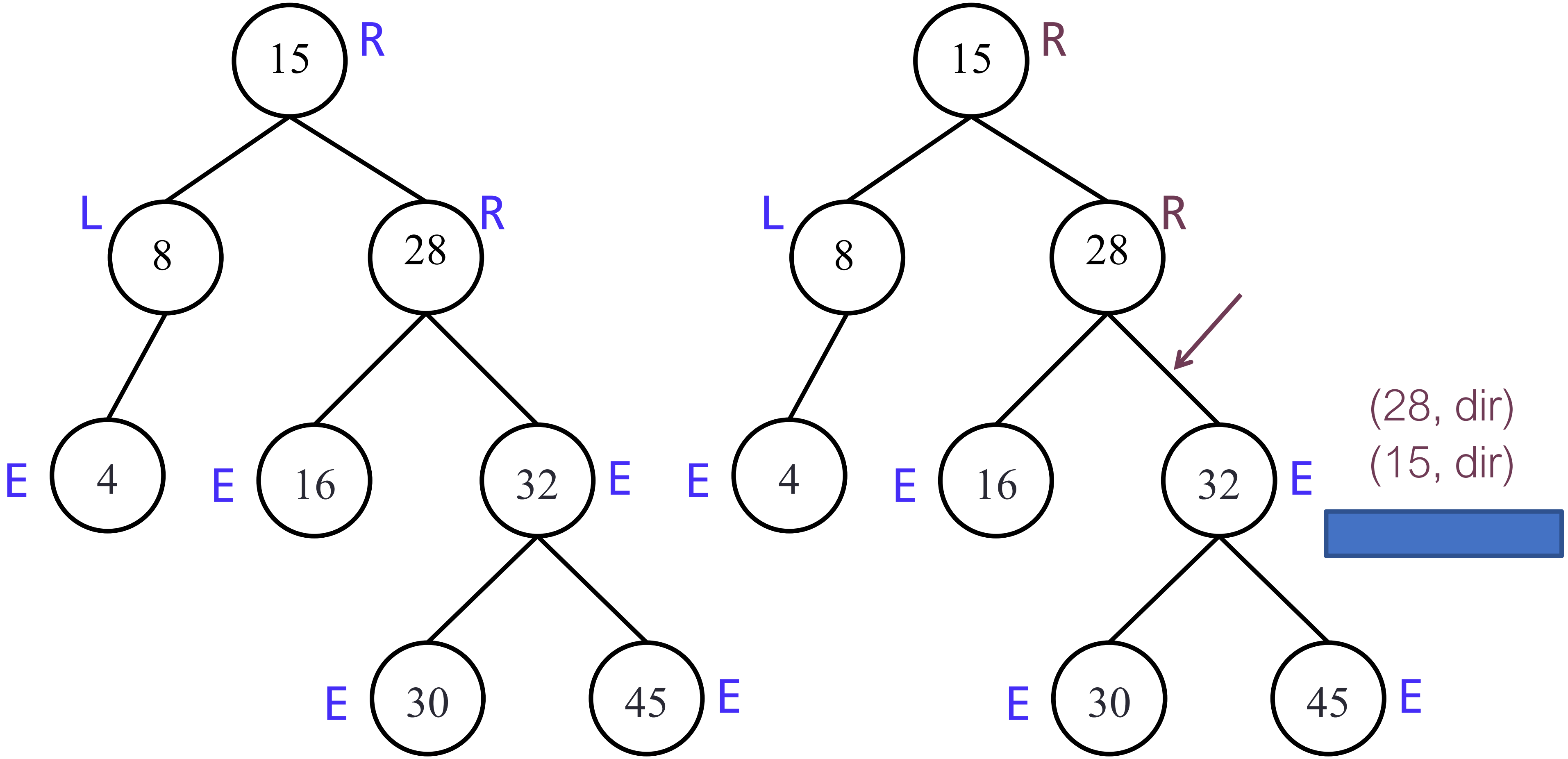
Encontrar mínimo da subárvore
direita e substituir 28





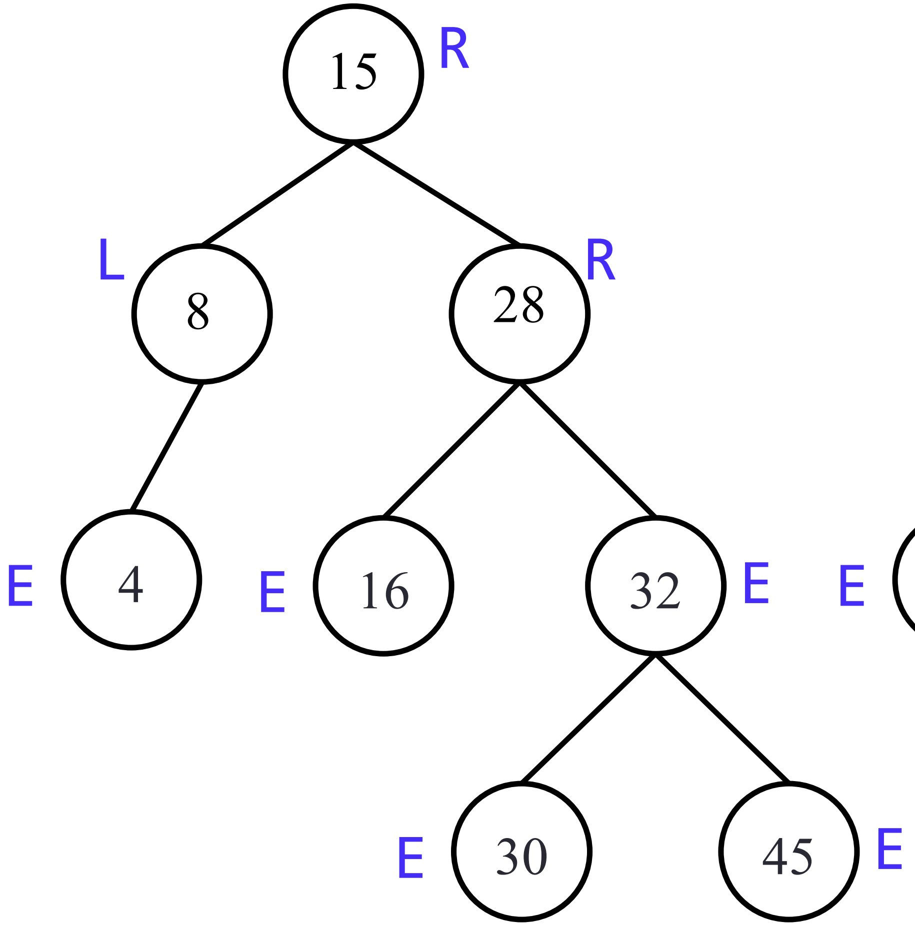
Remover nó com dois filhos (28)

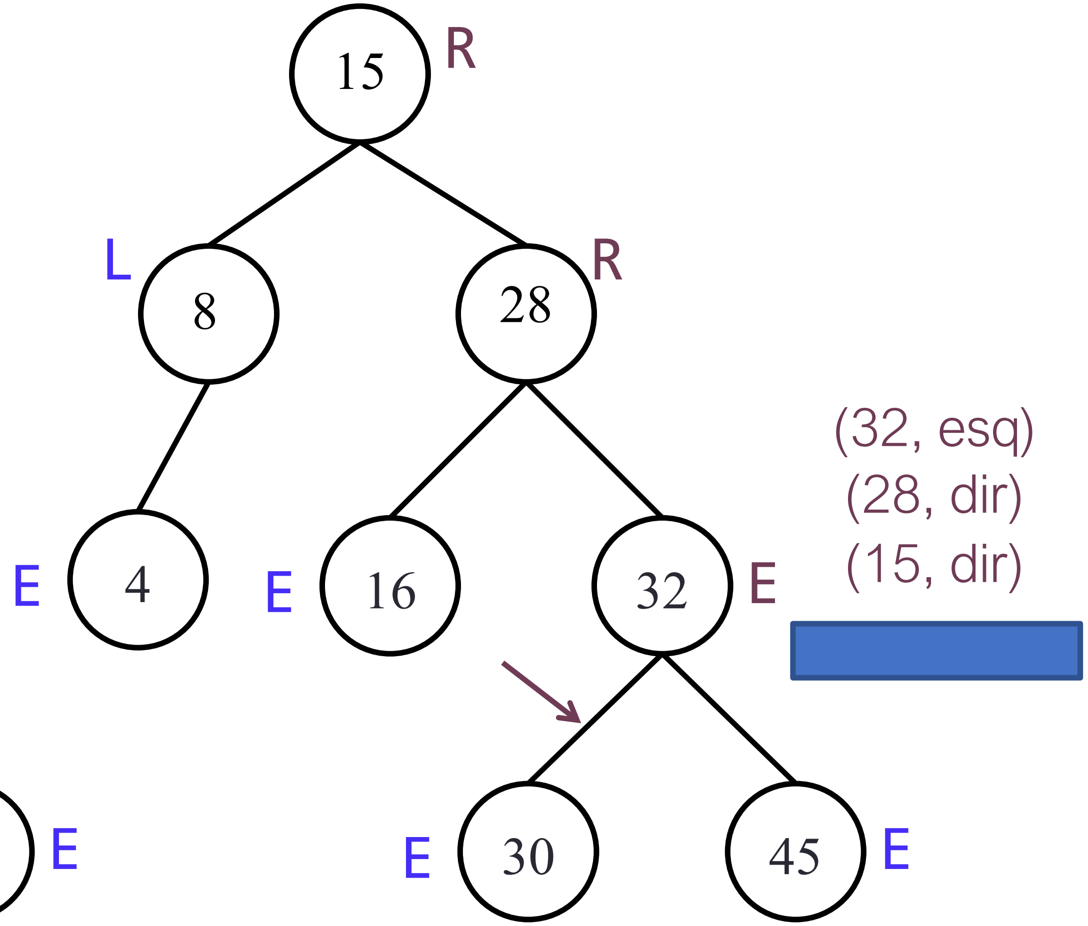
Encontrar mínimo da subárvore
direita e substituir 28



Remover nó com dois filhos (28)

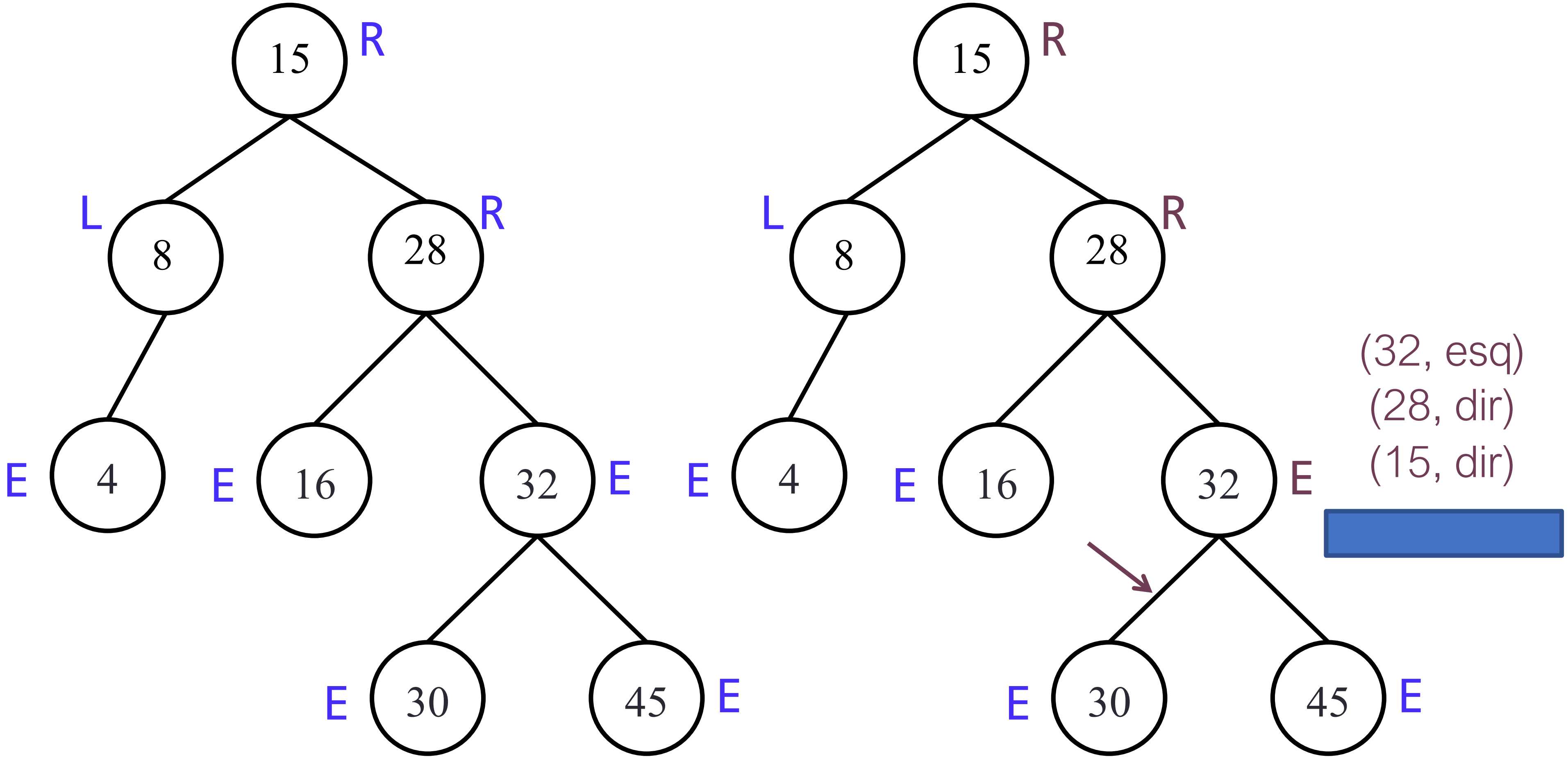
Encontrar mínimo da subárvore
direita e substituir 28





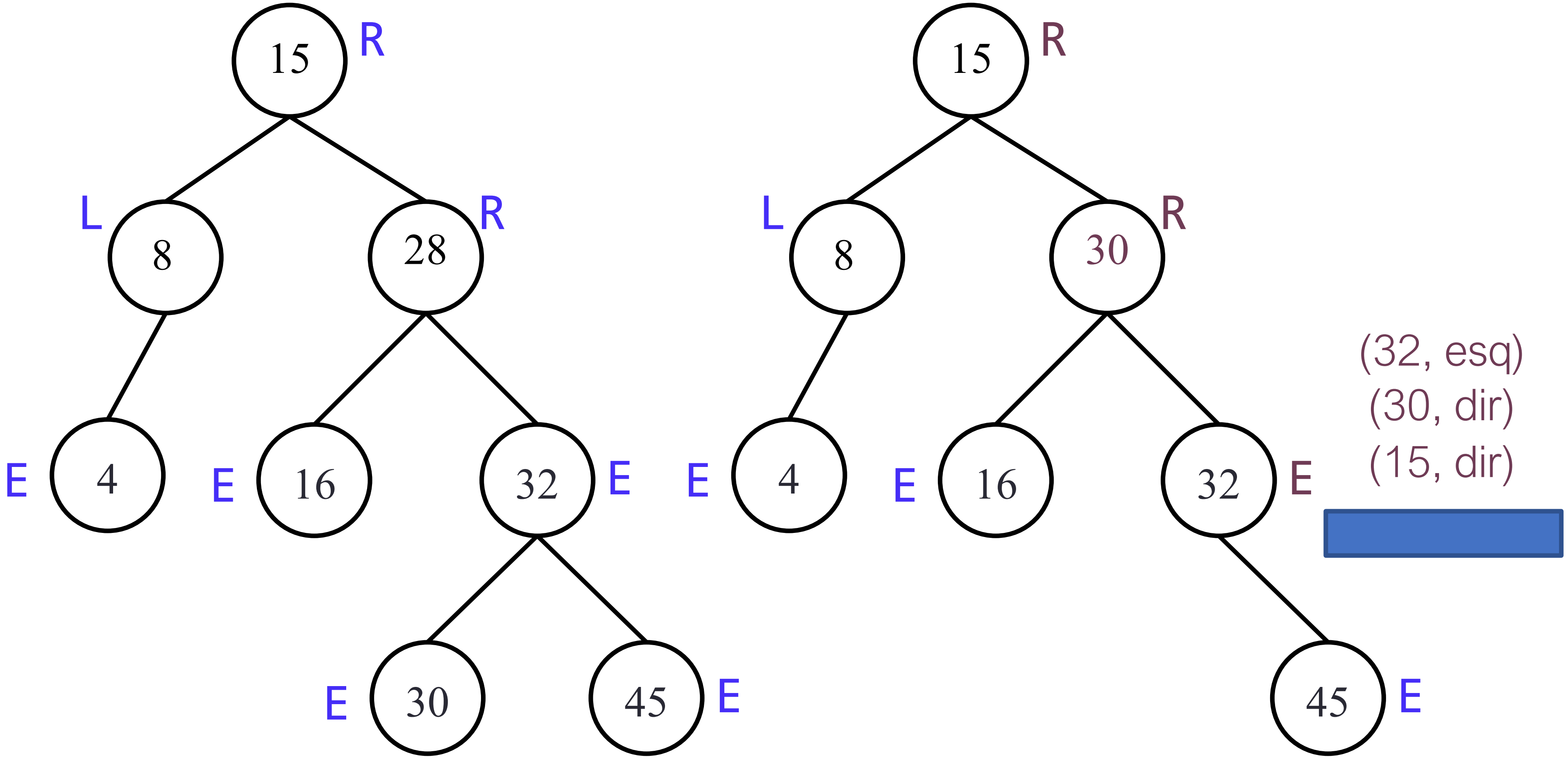
Remover nó com dois filhos (28)

Remover mínimo: pai herda subárvore direita



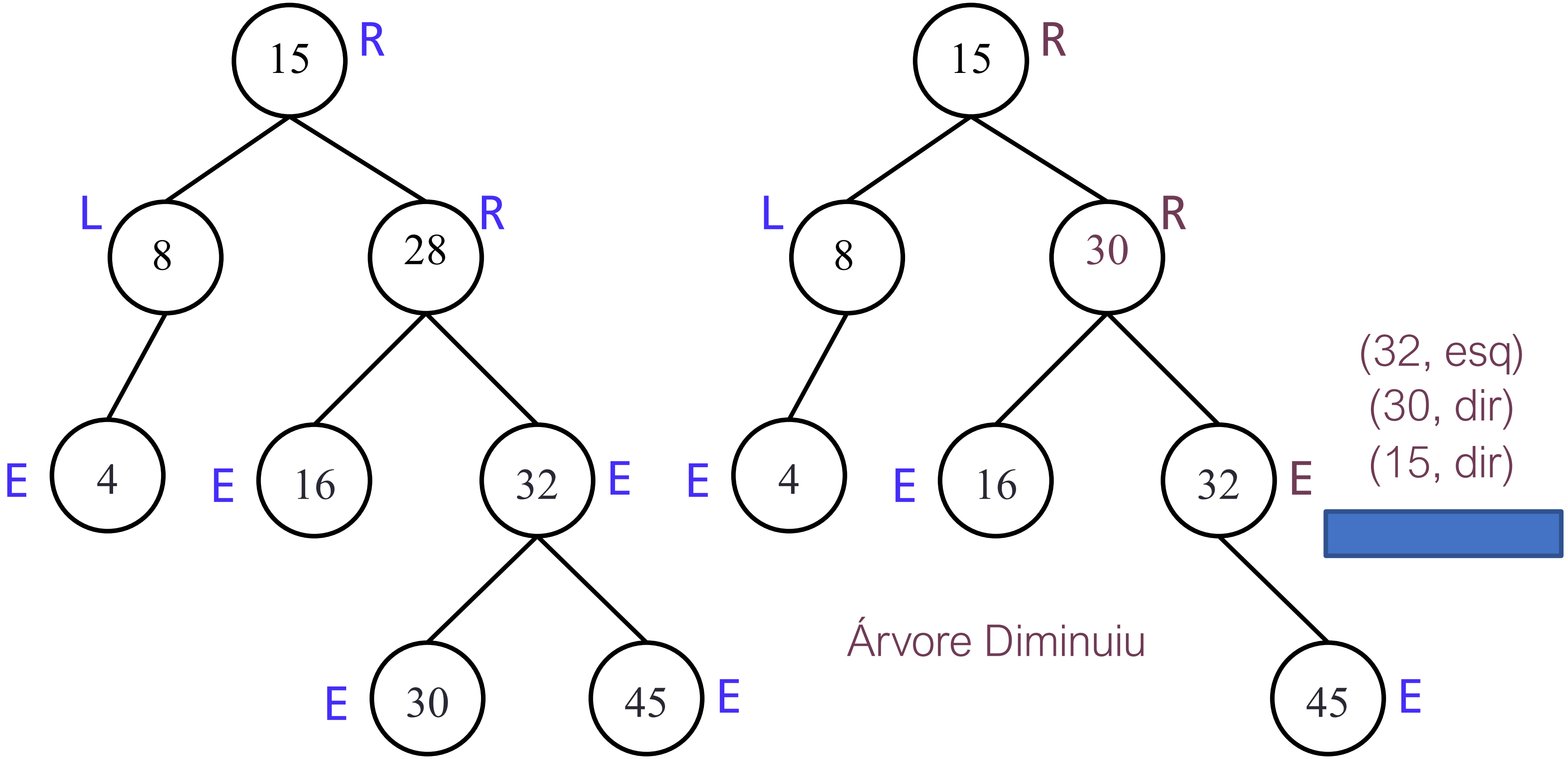
Remover nó com dois filhos (28)

Remover mínimo: pai herda
subárvore direita



Remover nó com dois filhos (28)

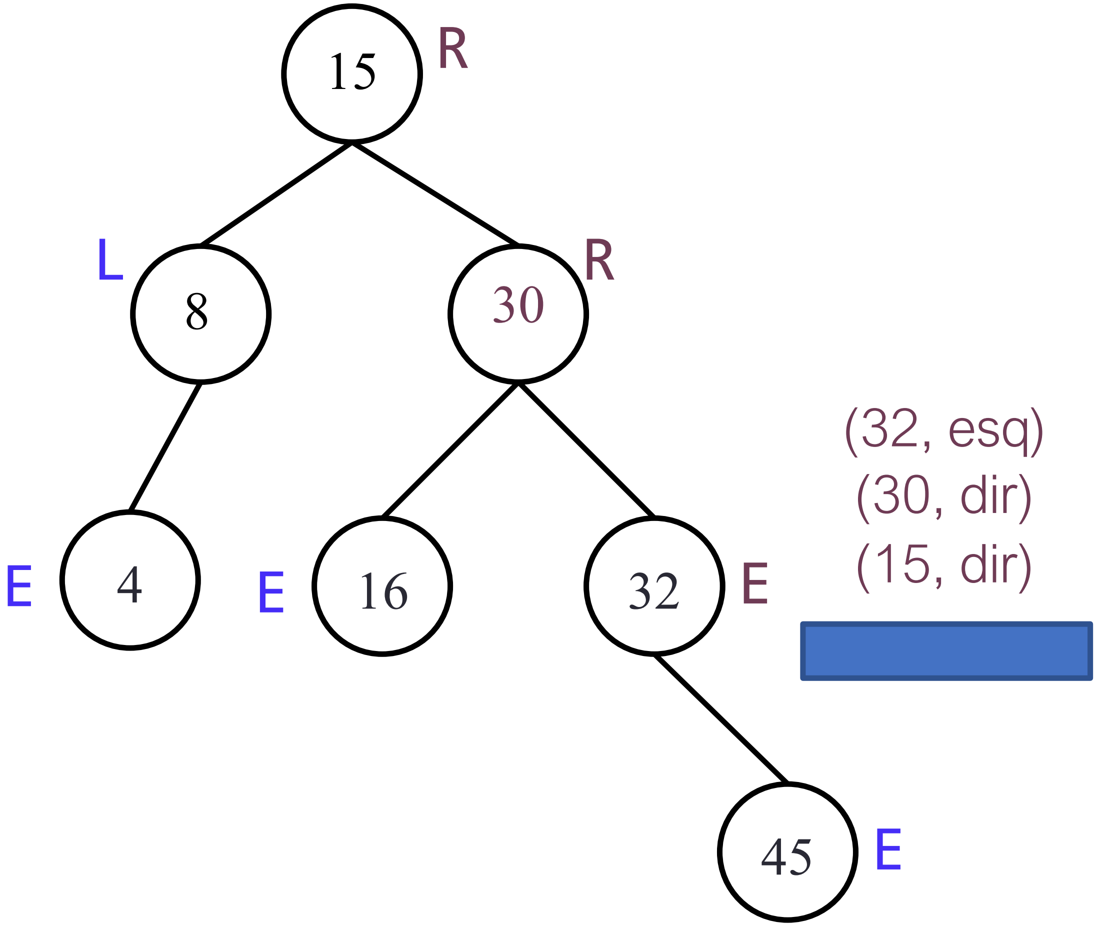
Remover mínimo: pai herda subárvore direita



Árvore Diminuiu

Remover nó com dois filhos (28)

Passo:

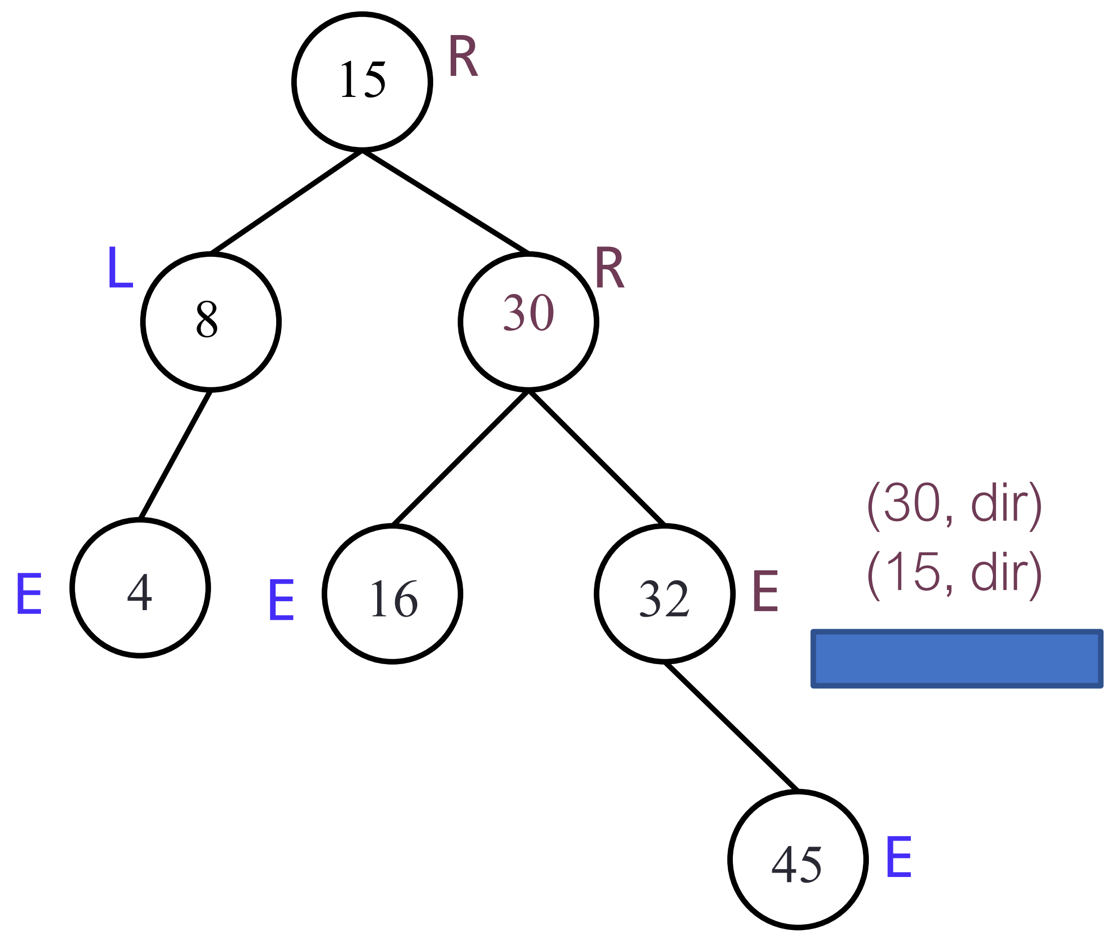


Remover nó com dois filhos (28)

Passo: (32, esq)

Subárvore esquerda diminuiu

E → R



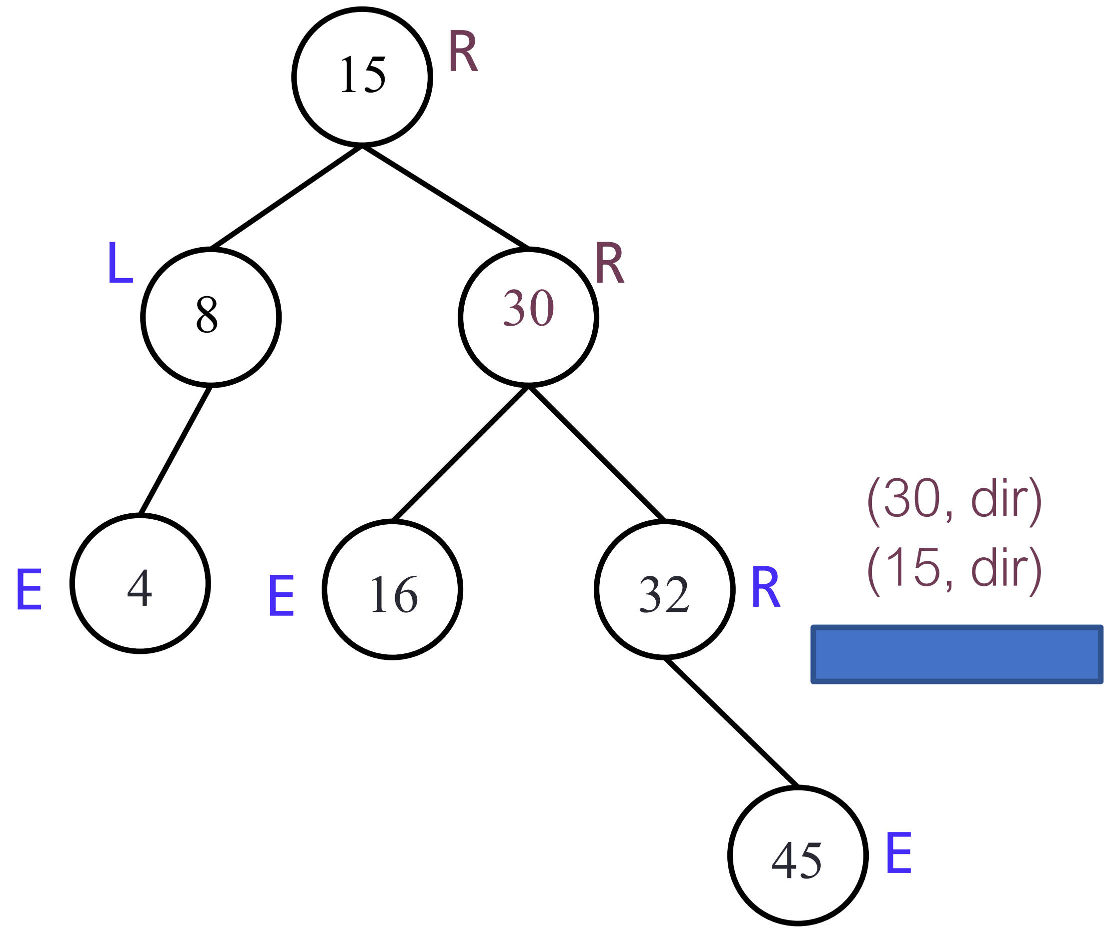
Remover nó com dois filhos (28)

Passo: (32, esq)

Subárvore esquerda diminuiu

E → R

Árvore não Diminuiu



Nó mínimo que guarda o caminho

```
// Returns the node with the smallest key
// in the tree rooted at the specified node.
// Moreover, stores the path into the stack.
// Requires: theRoot != null.
protected BSTNode<K,V> minNode( BSTNode<K,V> theRoot,
                                Stack<PathStep<K,V>> path ){

   BSTNode<K,V> node = theRoot;
   while ( node.getLeft() != null ){
      path.push( new PathStep<K,V>(node, true) );
      node = node.getLeft();
   }
   return node;
}
```

Classe Árvore AVL (10)

```
// If there is an entry in the dictionary whose key is the specified key,
// removes it from the dictionary and returns its value;
// otherwise, returns null.
public V remove( K key ) {

   Stack<PathStep<K,V>> path = new StackInList<PathStep<K,V>>();
   BSTNode<K,V> node = this.findNode(key, path);

   if ( node == null )
      return null;
   else {
      V oldValue = node.getValue();
      // Remover a entrada de node.
      currentSize--;
      this.reorganizeRem(path);
      return oldValue;
   }
}
```

Slide
Seguinte

Classe Árvore AVL (11) – Remover a entrada

```
if ( node.getLeft() == null )
   // The left subtree is empty.
   this.linkSubtree(node.getRight(), path.top());
else if ( node.getRight() == null )
        // The right subtree is empty.
        this.linkSubtree(node.getLeft(), path.top());
     else {
           // Node has 2 children. Replace the node's entry with
           // the 'minEntry' of the right subtree.
           path.push( new PathStep<K,V>(node, false) );
           BSTNode<K,V> minNode = this.minNode(node.getRight(), path);
           node.setEntry( minNode.getEntry() );
           // Remove the 'minEntry' of the right subtree.
           this.linkSubtree(minNode.getRight(), path.top());
         }
```

Complexidades da Árvore Binária de Pesquisa (com n nós)

Pesquisa Inserção Remoção Mínimo Máximo

	Pior Caso	Caso Esperado
Árvore sem restrições	n	$\log n$
AVL	$(1.44) \log n$	$\log n$

Percurso Percurso Ordenado

	"qualquer caso"
"qualquer" árvore	n