Algoritmos e Estruturas de Dados

TAD **SortedMap** (parte II) – Capítulo 11

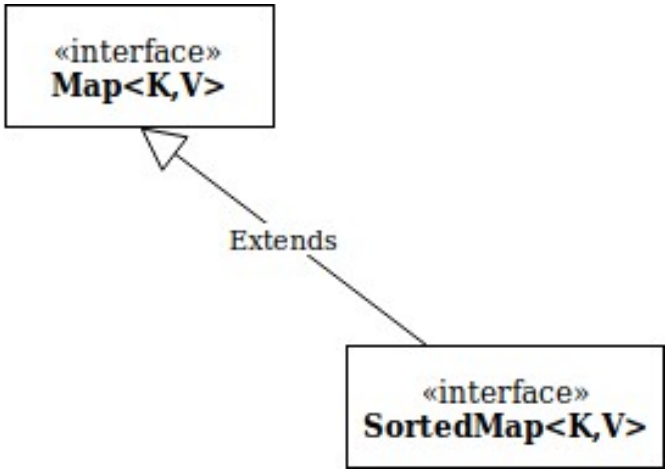
2019/20

Package dataStructures: interfaces (TADs)

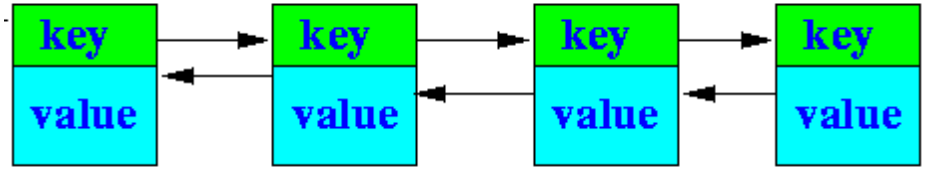
- Dicionário Ordenado (interface ***SortedMap***)
  - Dicionário, em que é necessário ordenação nas chaves.

TAD ***SortedMap***

```
package dataStructures;

public interface SortedMap<K, V> extends Map<K,V>{

    // Returns the entry with the smallest key in the SortedMap.
    // @throws NoElementException if isEmpty()
    Entry<K,V> minEntry( ) throws NoElementException;

    // Returns the entry with the largest key in the SortedMap.
    // @throws NoElementException if isEmpty()
    Entry<K,V> maxEntry( ) throws NoElementException;

}
```



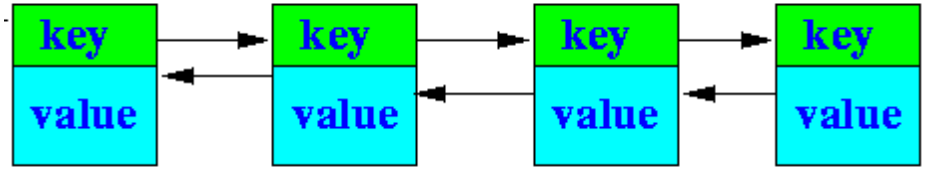
TAD ***Map*** (1)




```
package dataStructures;


public interface Map<K, V> {

    // Returns true iff the map contains no entries.
    boolean isEmpty( );

    // Returns the number of entries in the map.
    int size( );

    // Returns an iterator of the keys in the map.
    Iterator<K> keys( ) throws NoElementException;

    // Returns an iterator of the values in the map.
    Iterator<V> values( ) throws NoElementException;

    // Returns an iterator of the entries in the map.
    Iterator<Entry<K,V>> iterator() throws NoElementException;
    …
}
```

Estes iteradores dão
os elementos ordenados por chave.

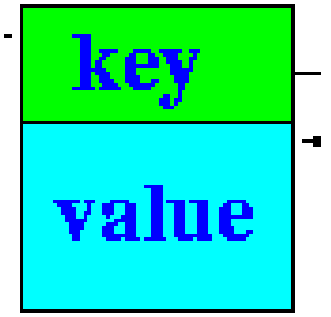
TAD ***Map*** (2)




```
package dataStructures;

public interface Map<K, V> {

    …

    // If there is an entry in the map whose key is the specified key,
    // returns its value; otherwise, returns null.
    V find( K key );

    // If there is an entry in the map 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.
    V insert( K key, V value );

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

}
```

Interface **Entry**

```
package dataStructures;

public interface Entry<K, V> {

    // Returns the key in the entry.
    K getKey( );

    // Returns the value in the entry.
    V getValue( );

}
```



TAD **SortMap<K,V>**
Opções de implementação

- As implementações realizadas usando as seguintes estruturas de dados:
  - Vector Ordenado;
  - Lista duplamente ligada ordenada;
  - Tabela de dispersão mais lista duplamente ligada ordenada
  - Tabela de dispersão mais Ordenação

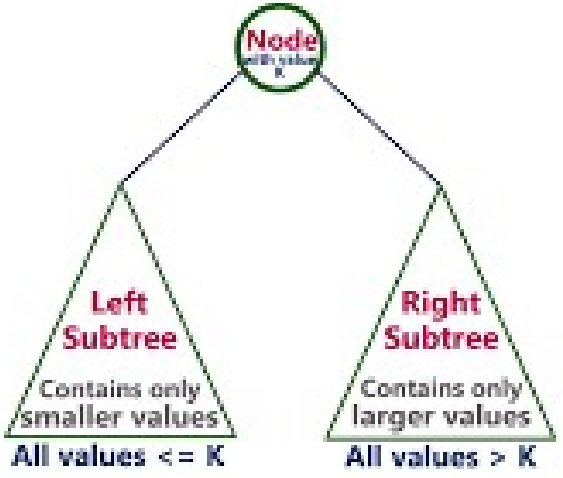
Não conseguimos melhor?

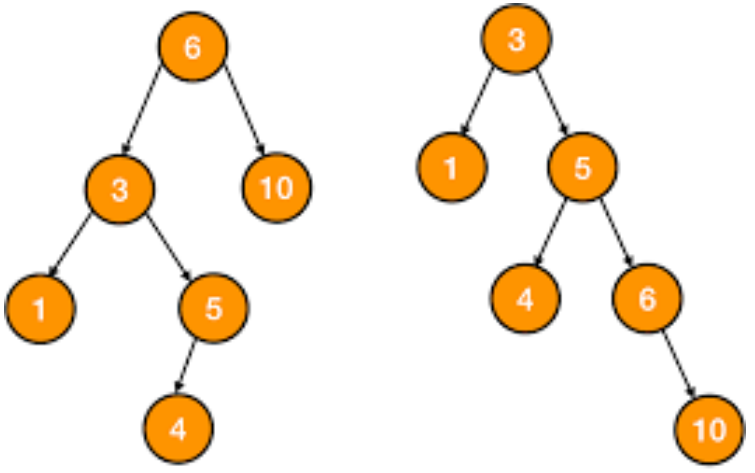
Uma estrutura de dados que consiga

O(log n) não só na pesquisa

Árvore binária de pesquisa

- Uma árvore binária de pesquisa é uma árvore ordenada binária em que todo nó X verifica as seguintes propriedades:
  - - Todo nó na sub-árvore esquerda, contem um elemento menor que o contido em X;
  - - Todo nó na sub-árvore direita, contem um elemento maior que o contido em X.

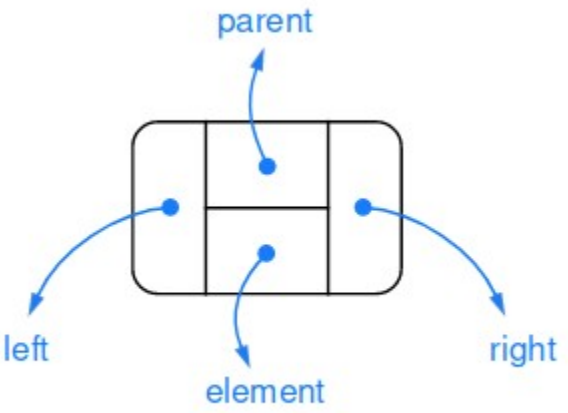




Two Binary Search Trees Representing Same Set

Classe **BinaryTreeNode**

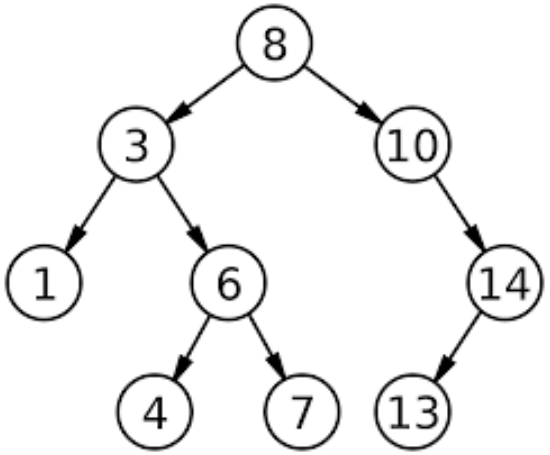
```
package dataStructures;

class BTNode<E> implements Node<E>{

    protected BTNode<E> parent;
    protected BTNode<E> left;
    protected BTNode<E> right;
    protected E element;

    public BTNode(E elem){
        parent=null;
        left=null;
        right=null;
        element=elem;
    }
    public BTNode(E elem,BTNode<E> parent,BTNode<E> left,BTNode<E> right){
        this.parent=parent;
        this.left=left;
        this.right=right;
        element=elem;
    }
...
}
```

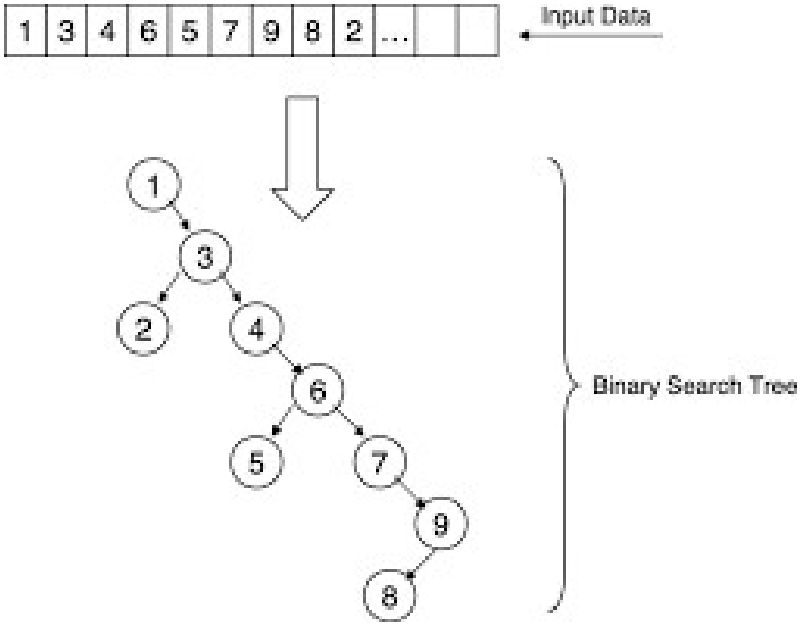


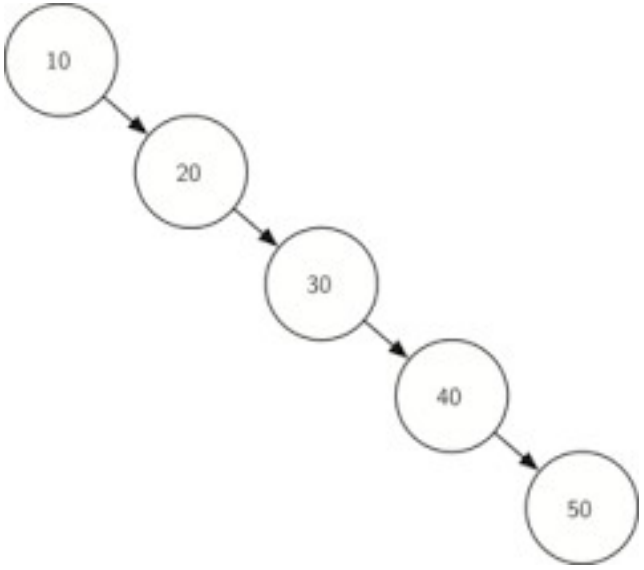
Árvore binária de pesquisa

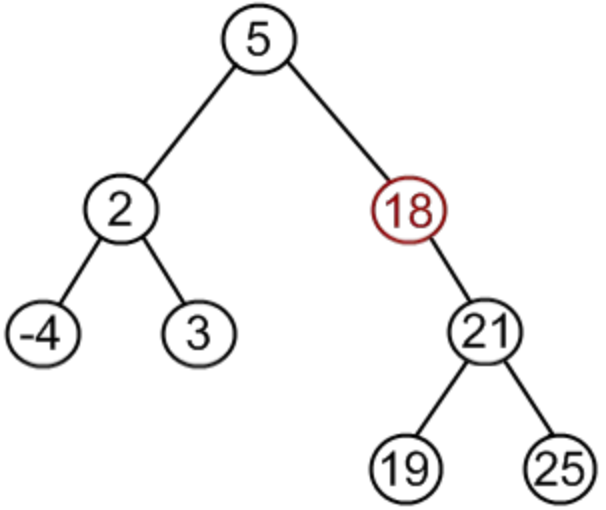
```
package dataStructures;

public class BinarySearchTree<K extends Comparable<K>,V>
                    extends BinaryTree<Entry<K,V>> implements SortedMap<K,V>{

    protected BinarySearchTree(Node<Entry<K,V>> n) {
        root=n;
    }
    public BinarySearchTree() {
        this(null);
    }
    @Override
    protected Node<Entry<K, V>> left(Node<Entry<K, V>> n) {
        return ((BTNode<Entry<K, V>>)n).getLeft();
    }
    @Override
    protected Node<Entry<K, V>> right(Node<Entry<K, V>> n) {
        return ((BTNode<Entry<K, V>>)n).getRight();
    }
    @Override
    protected Node<Entry<K, V>> parent(Node<Entry<K, V>> n) {
        return ((BTNode<Entry<K, V>>)n).getParent();
    }
    …
}
```

Árvore binária de pesquisa
(exemplos)

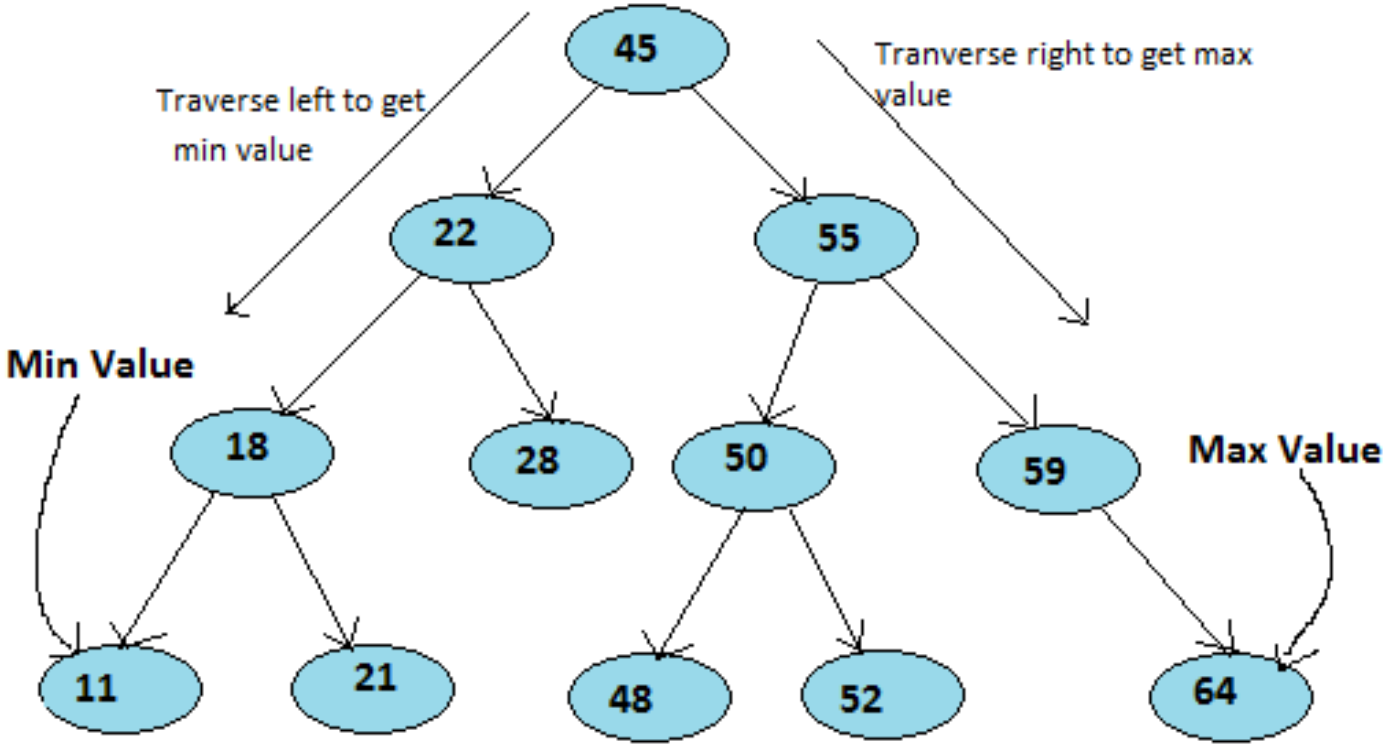






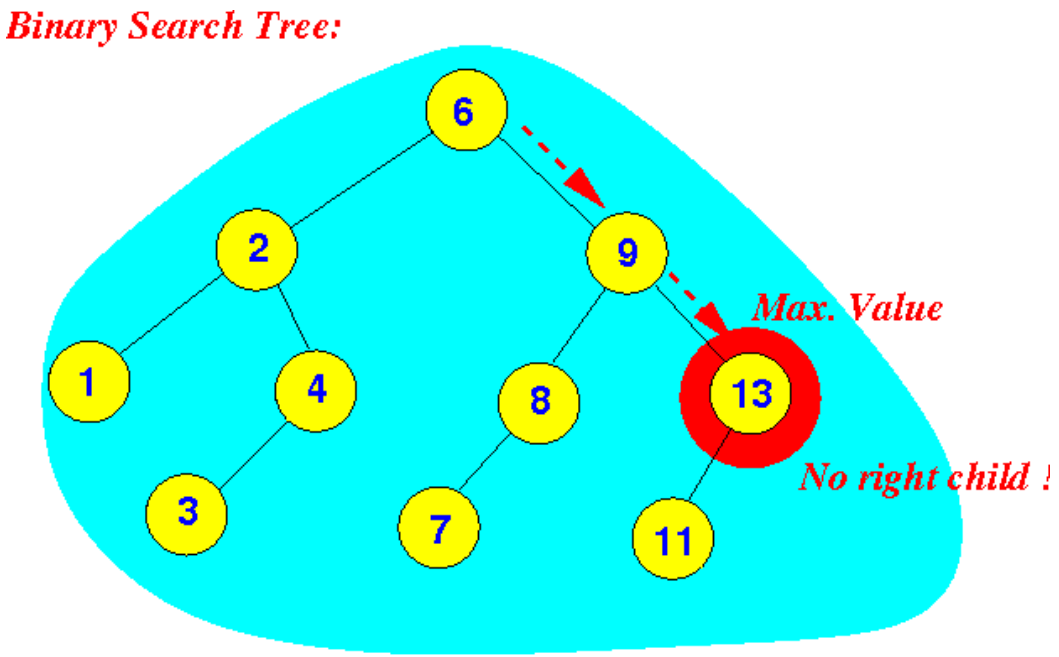


Árvore binária de pesquisa
(máximo e mínimo)



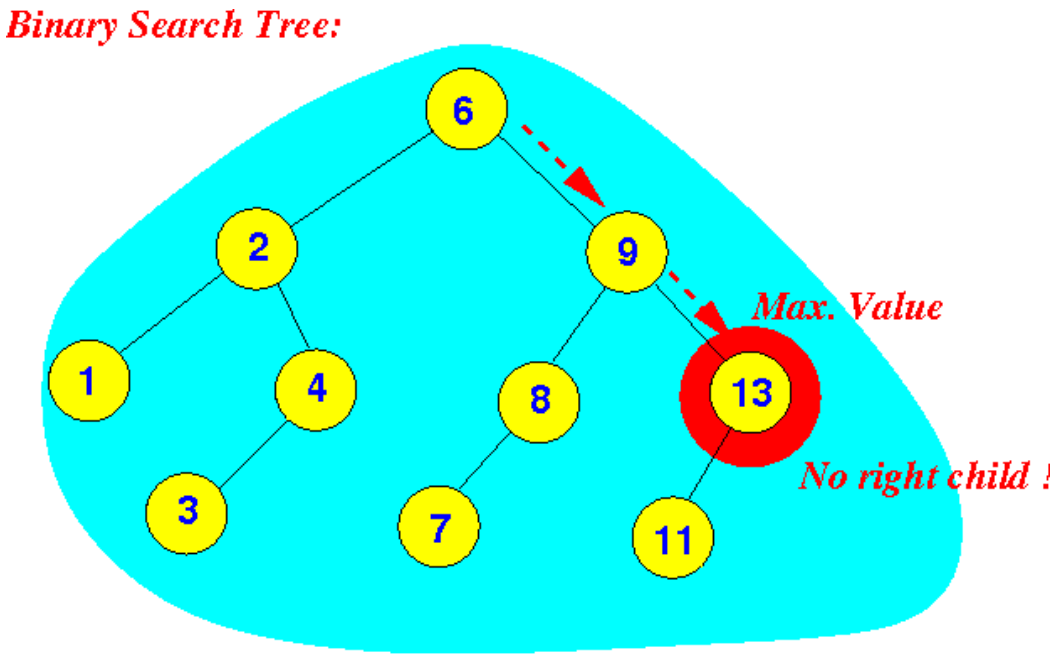
Árvore binária de pesquisa (máximo)

```
@Override
public Entry<K, V> maxEntry() throws NoElementException {
    if ( this.isEmpty() )
        throw new NoElementException();
    return maxNode(root).getElement();
}
// Precondition: node != null.
protected Node<Entry<K,V>> maxNode( Node<Entry<K,V>> node ){
    ...
}
```



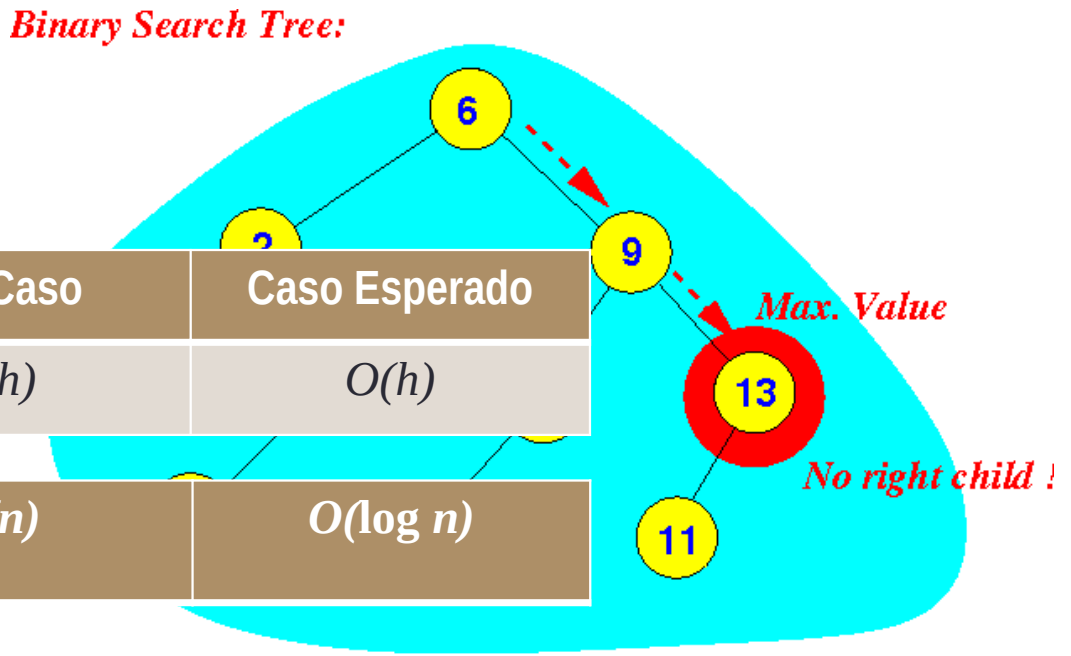
Árvore binária de pesquisa (máximo)

```
@Override
public Entry<K, V> maxEntry() throws NoElementException {
    if ( this.isEmpty() )
        throw new NoElementException();
    return maxNode(root).getElement();
}
// Precondition: node != null.
protected Node<Entry<K,V>> maxNode( Node<Entry<K,V>> node ){
    if ( ((BTNode<Entry<K, V>>) node).getRight() == null )
        return node;
    return maxNode(( (BTNode<Entry<K, V>>) node).getRight());
}
```



Árvore binária de pesquisa (máximo)

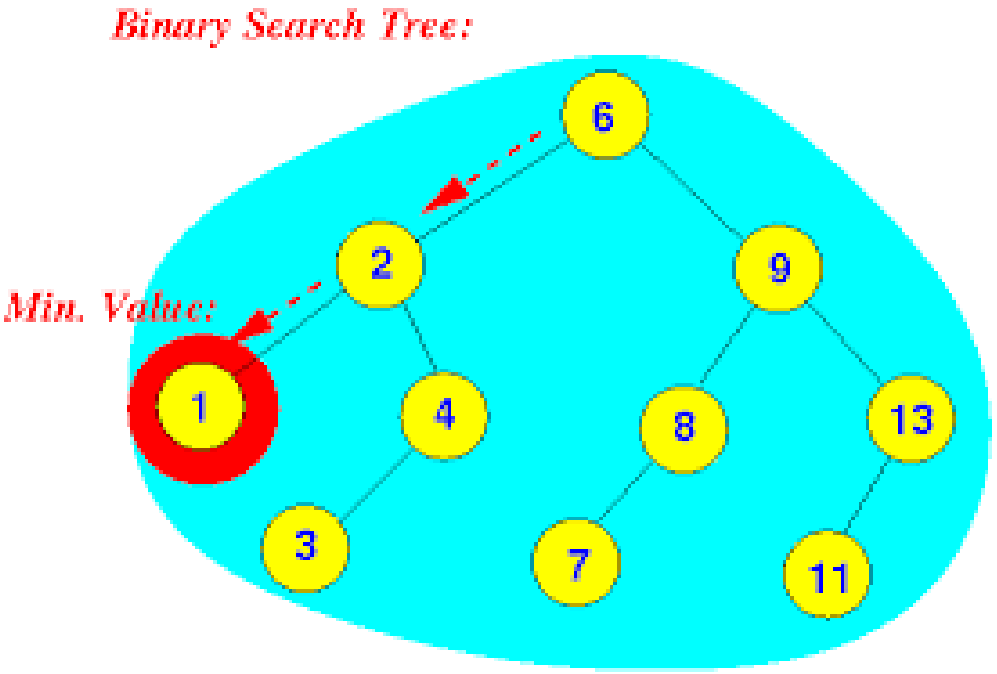
```
@Override
public Entry<K, V> maxEntry() throws NoElementException {
    if ( this.isEmpty() )
        throw new NoElementException();
    return this.maxNode(root).getElement();
}
// Precondition: node != null.
protected Node<Entry<K,V>> maxNode( Node<Entry<K,V>> node ){
    if ( ((BTNode<Entry<K, V>>) node).getRight() == null )
        return node;
    return this.maxNode(( (BTNode<Entry<K, V>>) node).getRight());
}
```



	Melhor Caso	Pior Caso	Caso Esperado
Máximo	*O(1)*	*O(h)*	*O(h)*
***h* (altura)**	***O*(log *n*)**	***O*(*n*)**	***O*(log *n*)**

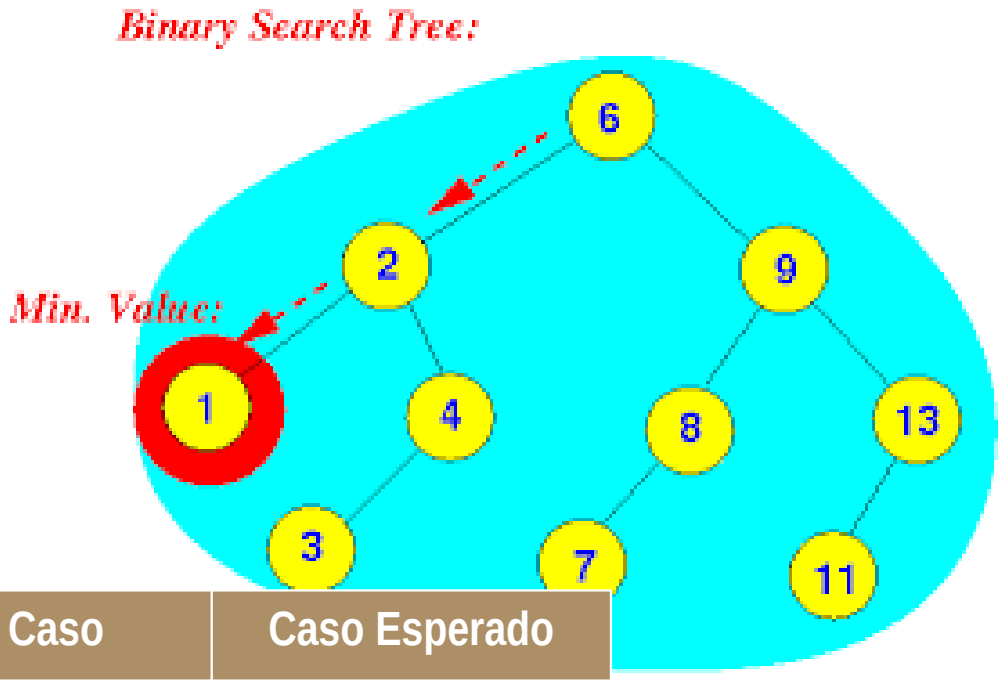
Árvore binária de pesquisa (mínimo)

```
@Override
public Entry<K, V> minEntry() throws NoElementException {
    // TODO
    return null;
}
```



Árvore binária de pesquisa (mínimo)

```java
@Override
public Entry<K, V> minEntry() throws NoElementException {
    // TODO
    return null;
}
```

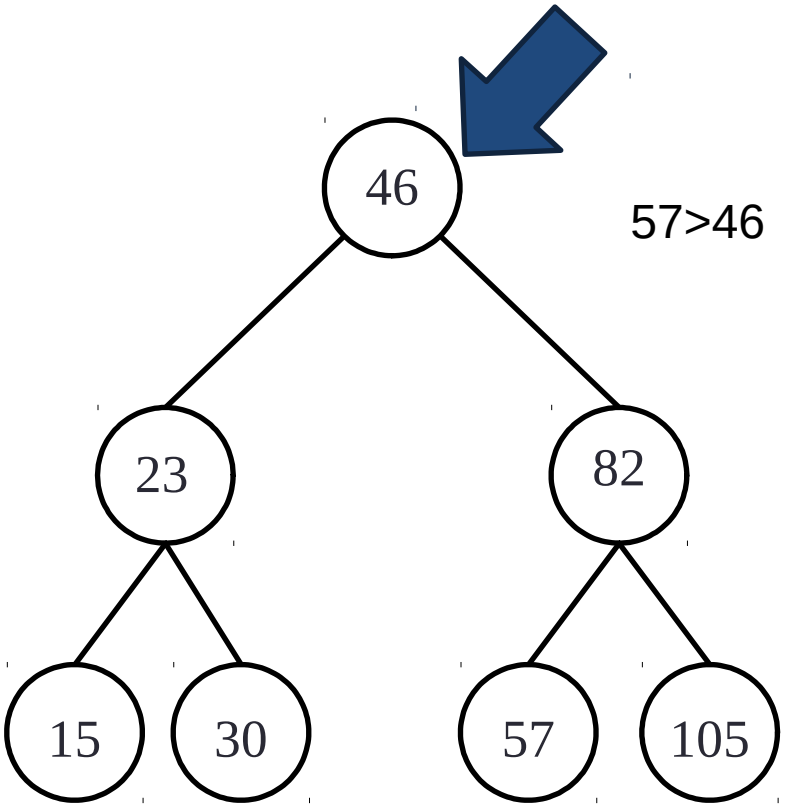


	Melhor Caso	Pior Caso	Caso Esperado
Mínimo	$O(1)$	$O(h)$	$O(h)$

h (altura)	$O(\log n)$	$O(n)$	$O(\log n)$

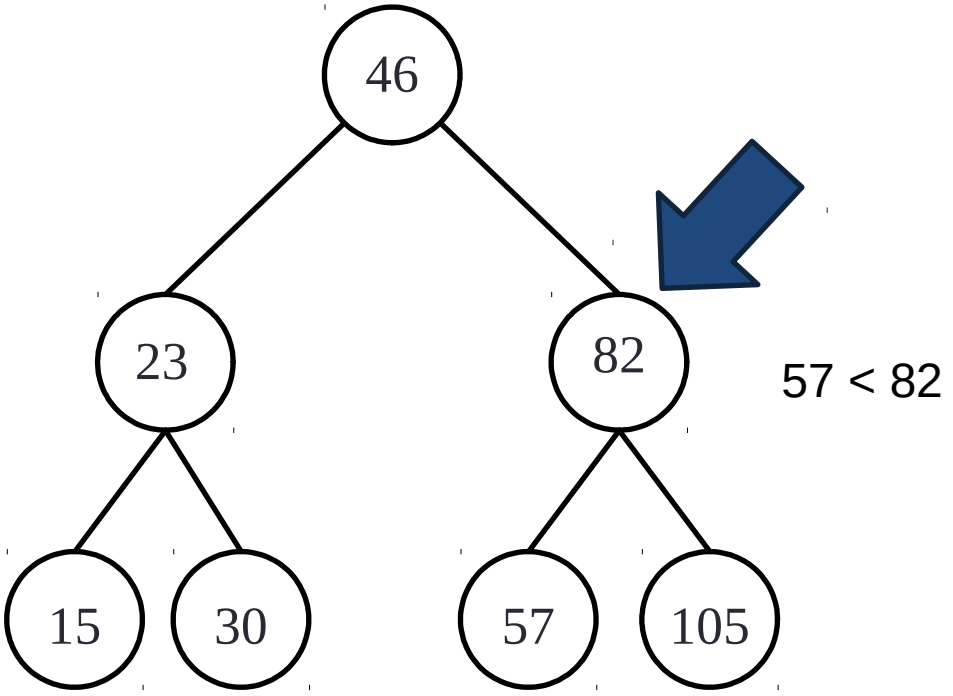
Árvore binária de pesquisa
(pesquisa)

find(57)



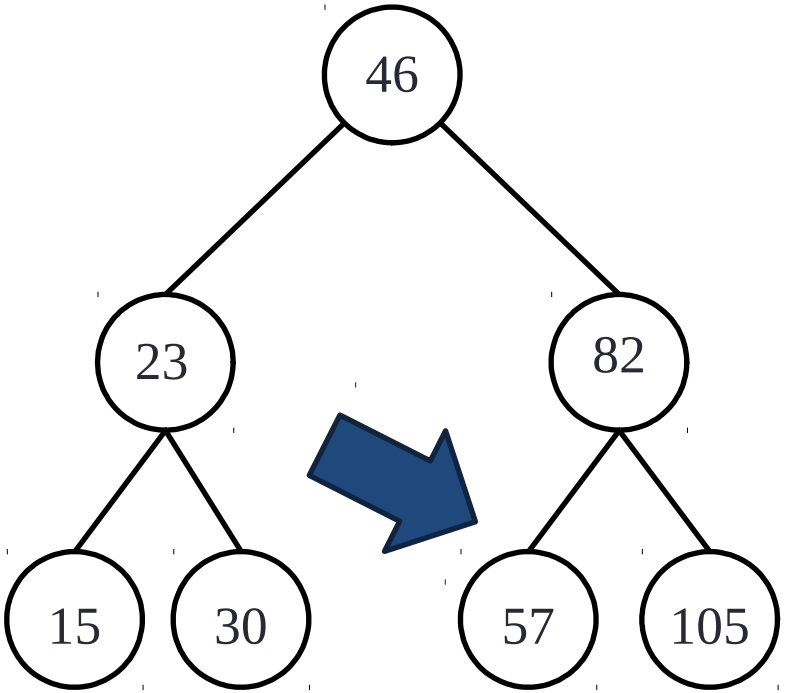
Árvore binária de pesquisa
(pesquisa)

find(57)



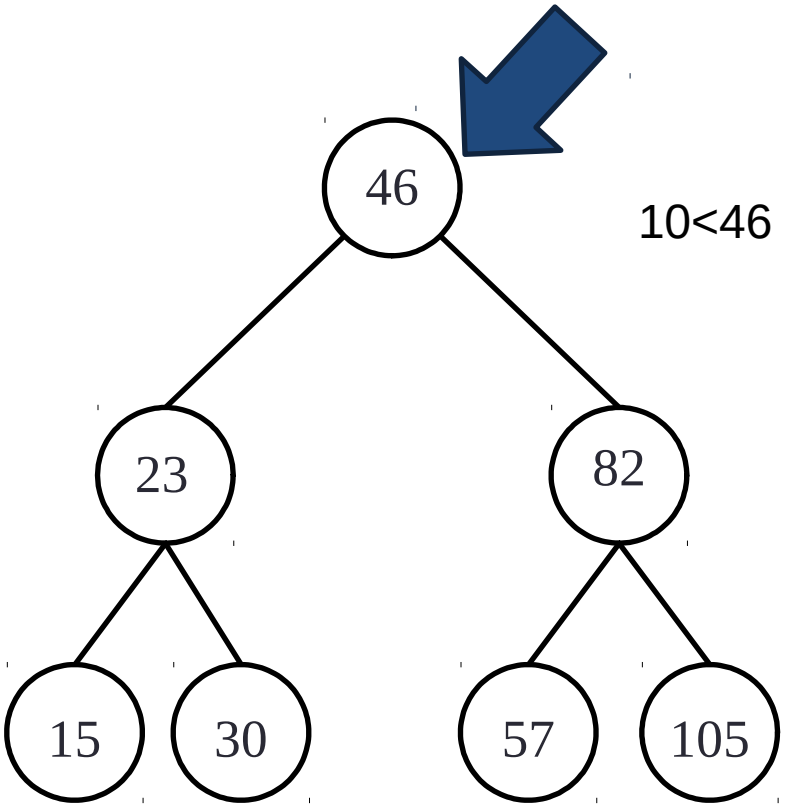
Árvore binária de pesquisa
(pesquisa)

find(57)



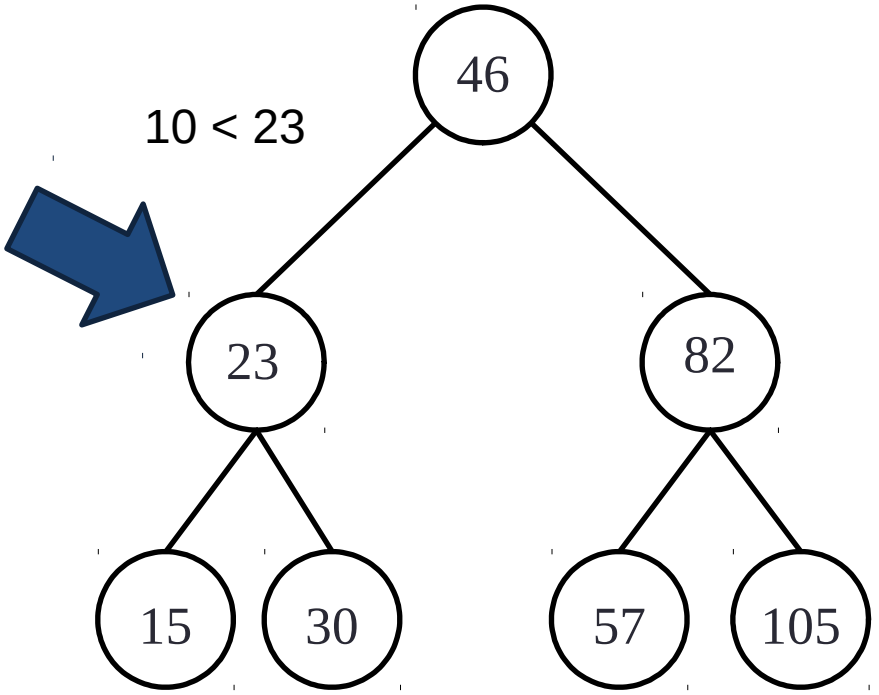
Árvore binária de pesquisa
(pesquisa)

find(10)



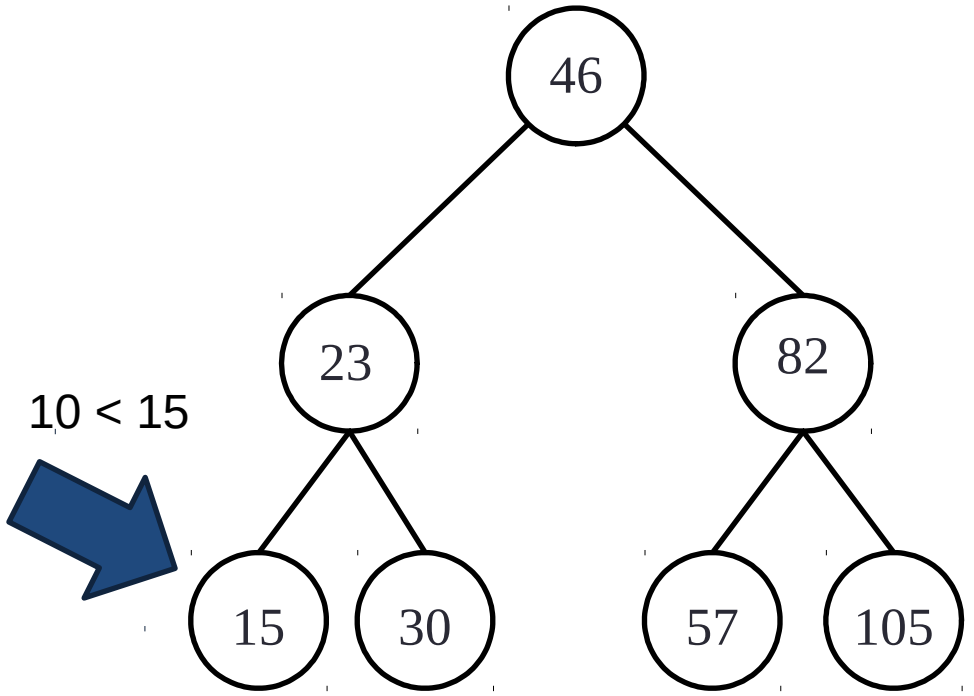
Árvore binária de pesquisa
(pesquisa)

find(10)



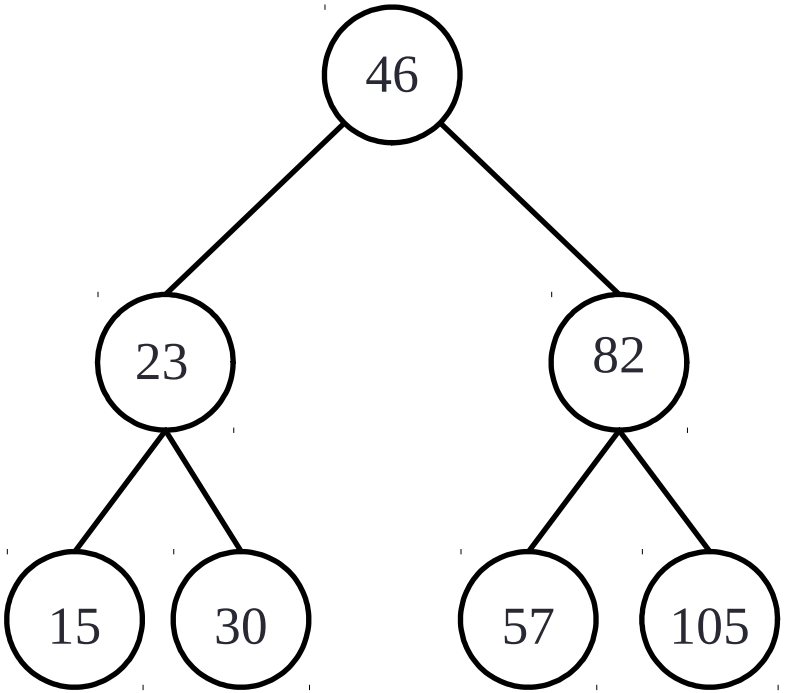
Árvore binária de pesquisa
(pesquisa)

find(10)



Árvore binária de pesquisa
(pesquisa)

find(10)



10 < 15 && não tem filho esquerdo

Árvore binária de pesquisa

```
package dataStructures;

public class BinarySearchTree<K extends Comparable<K>,V>
             extends BinaryTree<Entry<K,V>> implements SortedMap<K,V>{

    ...

    @Override
    public V find(K key) {
        Node<Entry<K,V>> res=findNode(root,key);
        if (res==null)
            return null;
        return res.getElement().getValue();
    }
…
}
```

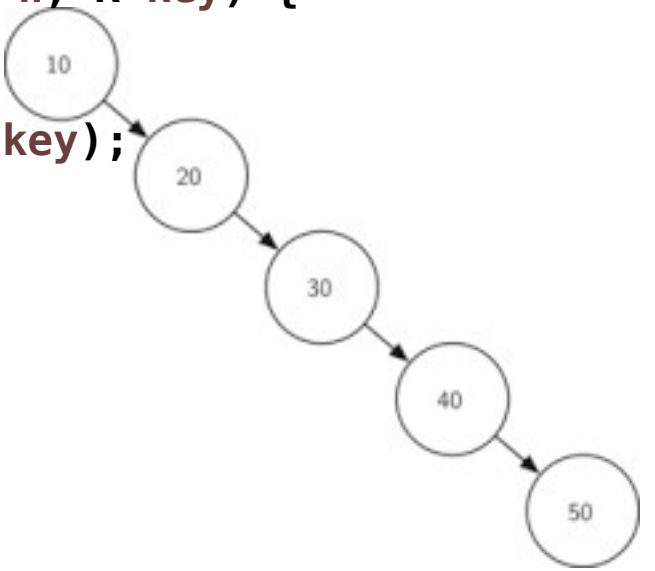
Árvore binária de pesquisa

```
package dataStructures;

public class BinarySearchTree<K extends Comparable<K>,V>
                    extends BinaryTree<Entry<K,V>> implements SortedMap<K,V>{

    ...
    protected Node<Entry<K,V>> findNode(Node<Entry<K,V>> n, K key) {
        Node<Entry<K,V>> res=null;
        if (n!=null) {
            int num= n.getElement().getKey().compareTo(key);
            if (num==0)
                res=n;
            else if (num<0)
                res=findNode(right(n),key);
            else
                res=findNode(left(n),key);
        }
        return res;
    }

    …
}
```

Árvore binária de pesquisa

Complexidade da Pesquisa Recursiva

```
private Node<Entry<K,V>> findNode(Node<Entry<K,V>> n, K key) {
    Node<Entry<K,V>> res=null;
    if (n!=null) {
        int num= n.getElement().getKey().compareTo(key);
        if (num==0)
            res=n;
        else if (num<0)
            res=findNode(right(n),key);
        else
            res=findNode(left(n),key);
    }
    return res;
}
```



No pior caso, a entrada está numa folha ou não ocorre na árvore.
Se a árvore "for uma lista":

Número de Chamadas Recursivas

$$\text{numCR}(n) = \begin{cases} 0 & n = 1 \\ \text{numCR}(n-1) + 1 & n \geq 2 \end{cases}$$

$O(n)$

$$T(n) = \begin{cases} a & n = 0 \\ b\,T(n-1) + c & n \geq 1 \end{cases} \quad \text{ou} \quad n = 1,\ n \geq 2$$

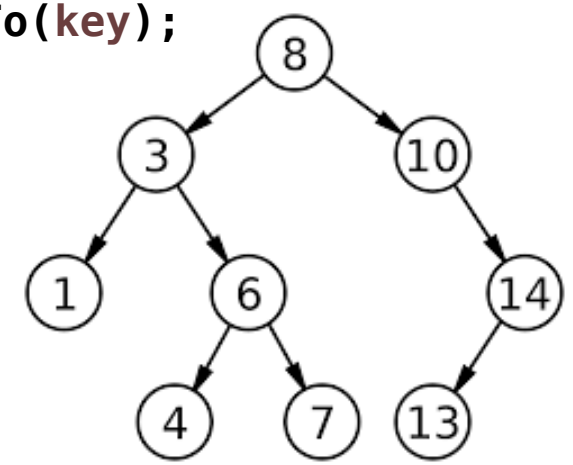
com $a \geq 0$, $b \geq 1$, $c \geq 1$ constantes

$$T(n) = \begin{cases} O(n) & b = 1 \\ O(b^n) & b > 1 \end{cases}$$

Árvore binária de pesquisa

Complexidade da Pesquisa Recursiva

```
private Node<Entry<K,V>> findNode(Node<Entry<K,V>> n, K key) {
    Node<Entry<K,V>> res=null;
    if (n!=null) {
        int num= n.getElement().getKey().compareTo(key);
        if (num==0)
            res=n;
        else if (num<0)
            res=findNode(right(n),key);
        else
            res=findNode(left(n),key);
    }
    return res;
}
```



No pior caso, a entrada está numa folha ou não ocorre na árvore.
Se a árvore "estiver equilibrada":

$$\text{numCR}(n) = \begin{cases} 0 & n = 0 \\ \text{numCR}(\frac{n}{2}) + 1 & n \geq 1 \end{cases}$$

O(log n)

$$T(n) = \begin{cases} a & n = 0 \\ & \text{ou} \\ b\,T(\frac{n}{2}) + O(1) & n \geq 1 \end{cases} \qquad n = 1 \quad n \geq 2$$

com $a \geq 0, \quad b = 1, 2$ constantes

$$T(n) = \begin{cases} O(\log n) & b = 1 \\ O(n) & b = 2 \end{cases}$$

Árvore binária de pesquisa

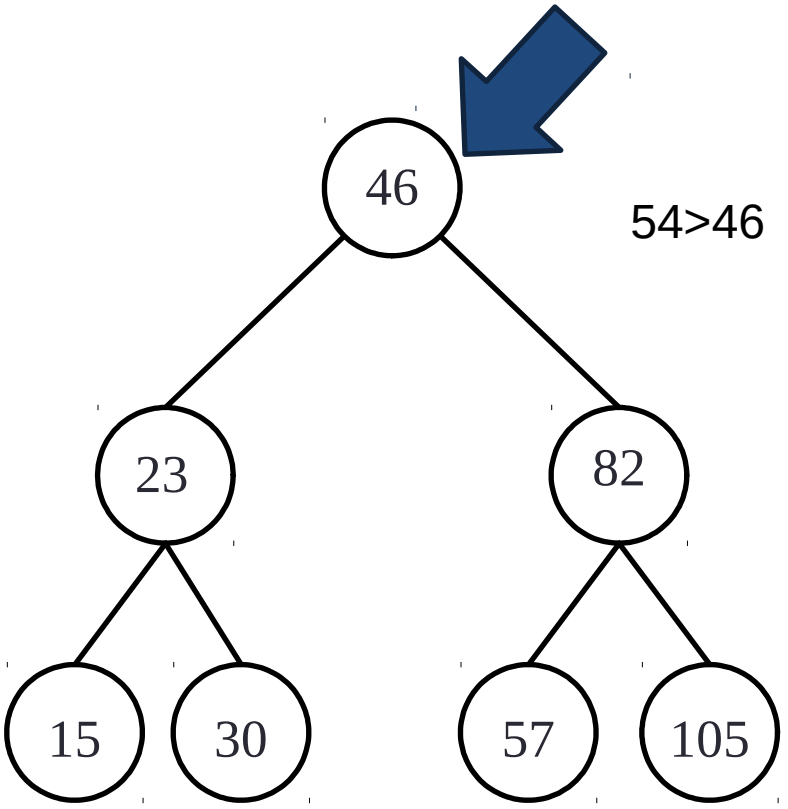
Complexidade da Pesquisa Recursiva

```
private Node<Entry<K,V>> findNode(Node<Entry<K,V>> n, K key) {
    Node<Entry<K,V>> res=null;
    if (n!=null) {
        int num= n.getElement().getKey().compareTo(key);
        if (num==0)
            res=n;
        else if (num<0)
            res=findNode(right(n),key);
        else
            res=findNode(left(n),key);
    }
    return res;
}
```

	Melhor Caso	Pior Caso	Caso Esperado
Pesquisa	$O(1)$	$O(h)$	$O(h)$
h (altura)	$O(\log n)$	$O(n)$	$O(\log n)$

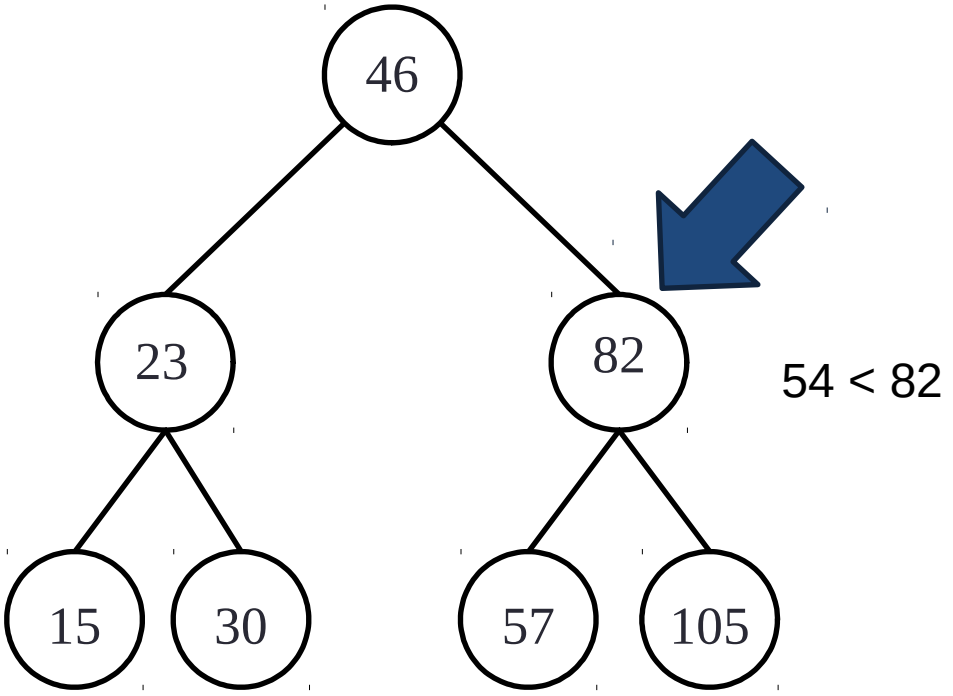
Árvore binária de pesquisa (inserção)

insert(54, _)



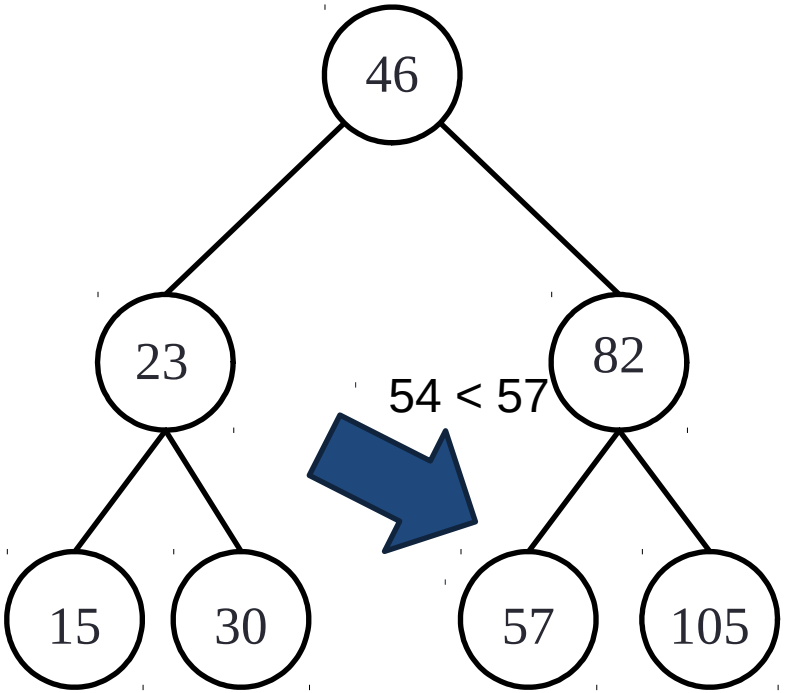
Árvore binária de pesquisa (inserção)

insert(54, _)



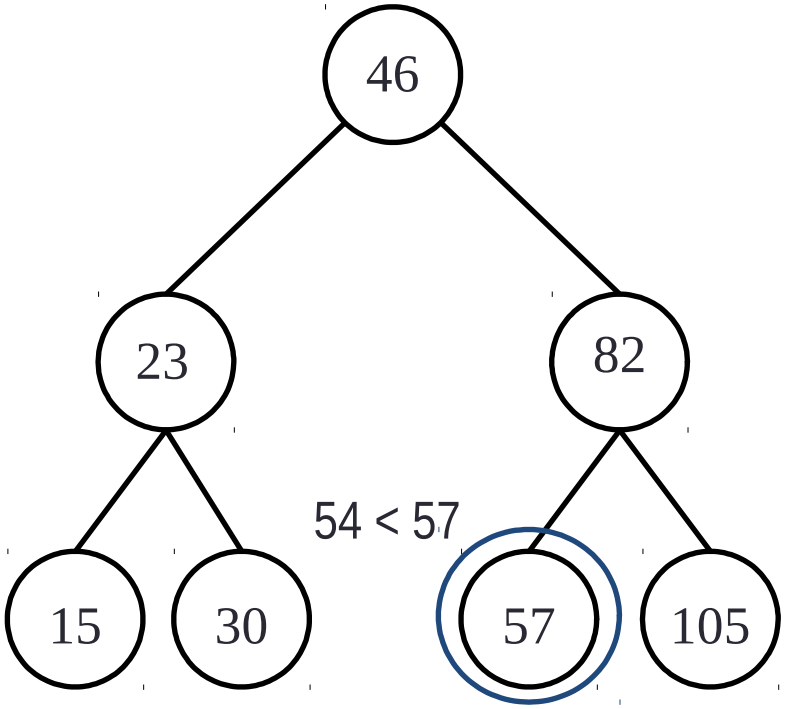
Árvore binária de pesquisa (inserção)

insert(54, _)



Árvore binária de pesquisa (inserção)

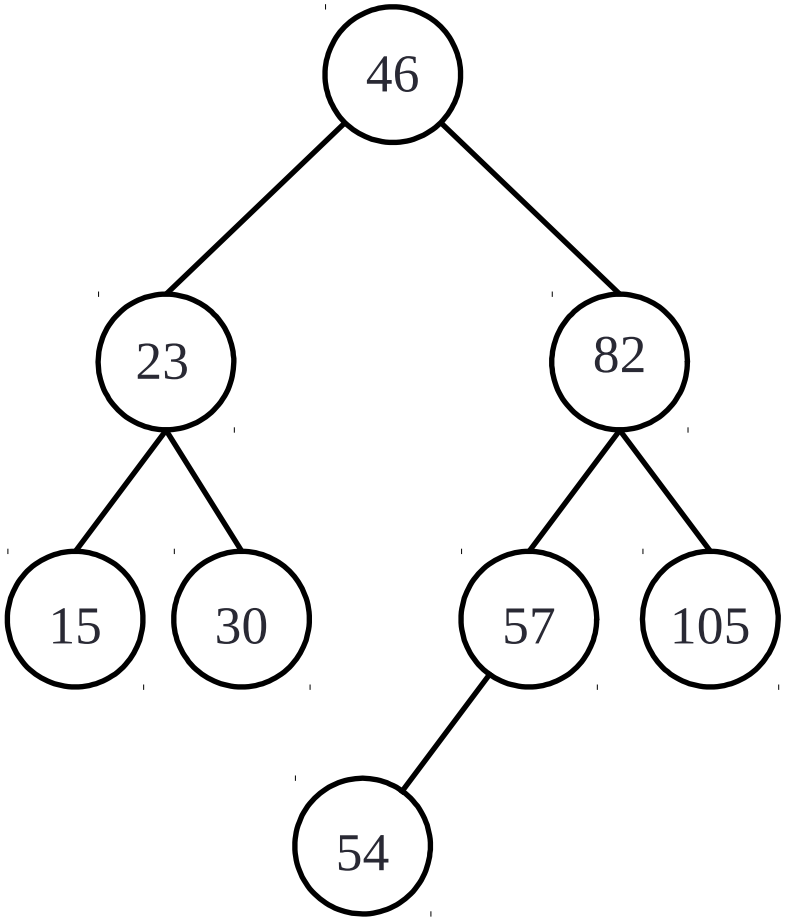
insert(54, _)



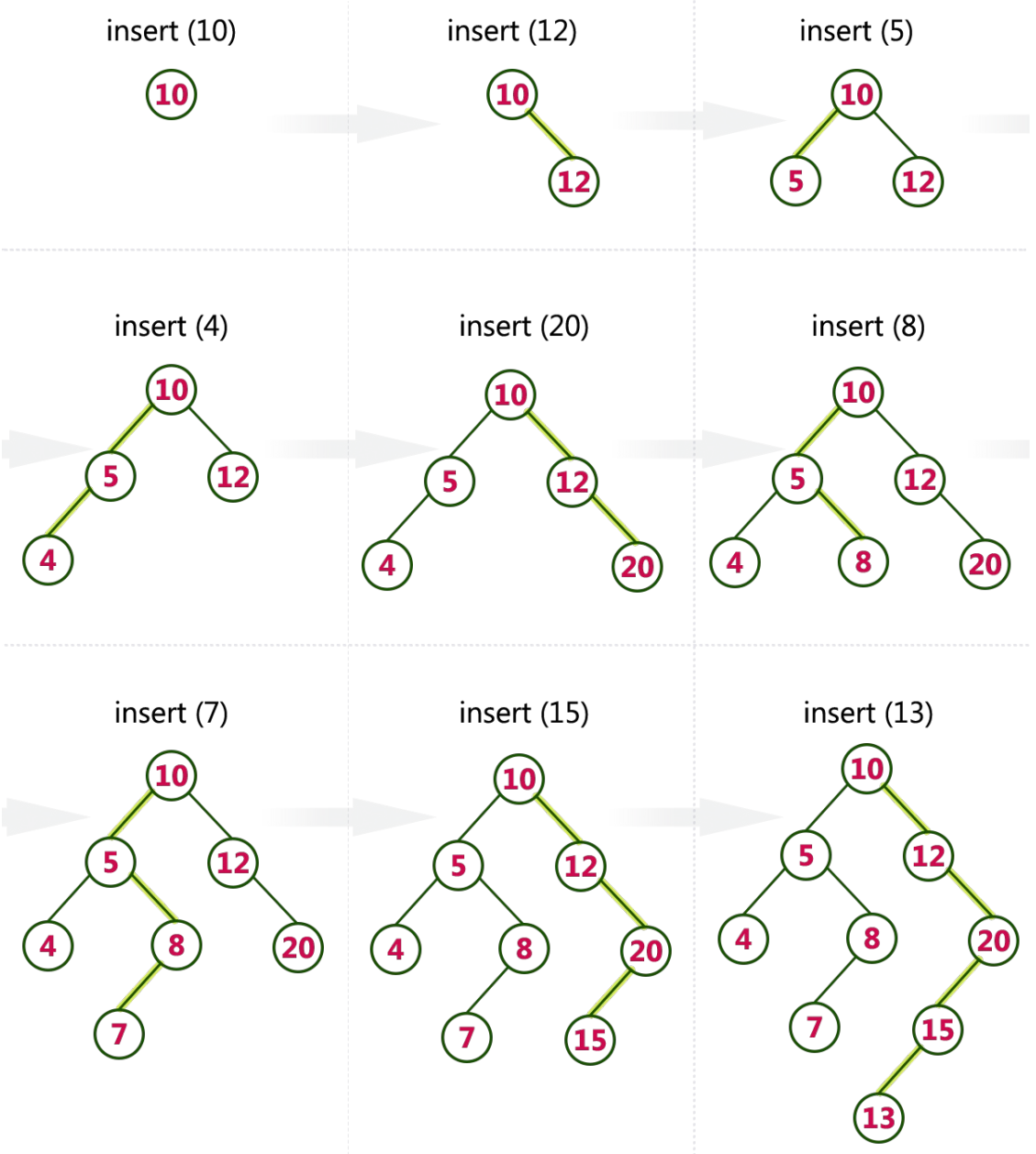
- 57 será o pai do novo nó
- Como 57 é maior que 54, o novo nó será filho esquerdo do 57

Árvore binária de pesquisa (inserção)

insert(54, _)



Árvore binária de pesquisa (inserção)



Árvore binária de pesquisa (inserção)

```java
@Override
public V insert(K key, V value) {
    // TODO
    return null;
}
```

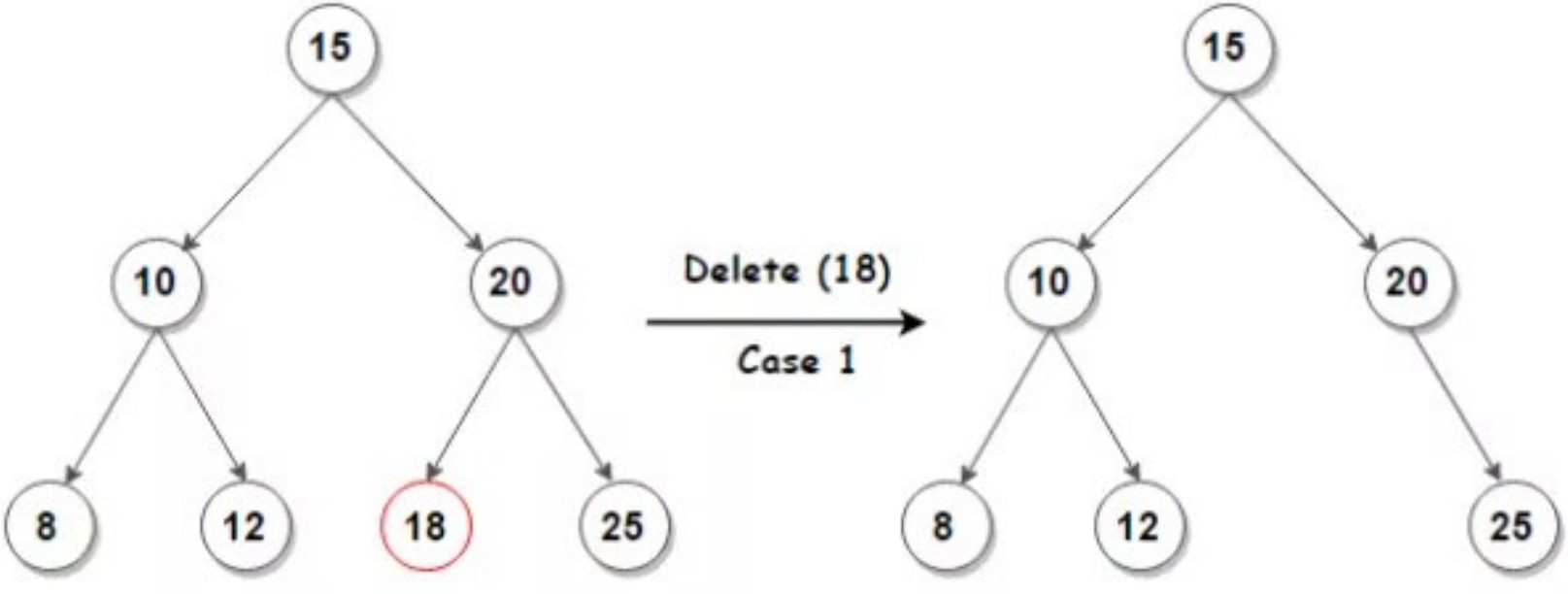
	Melhor Caso	Pior Caso	Caso Esperado
Inserção	$O(1)$	$O(h)$	$O(h)$

h (altura)	$O(\log n)$	$O(n)$	$O(\log n)$

Árvore binária de pesquisa (remoção)

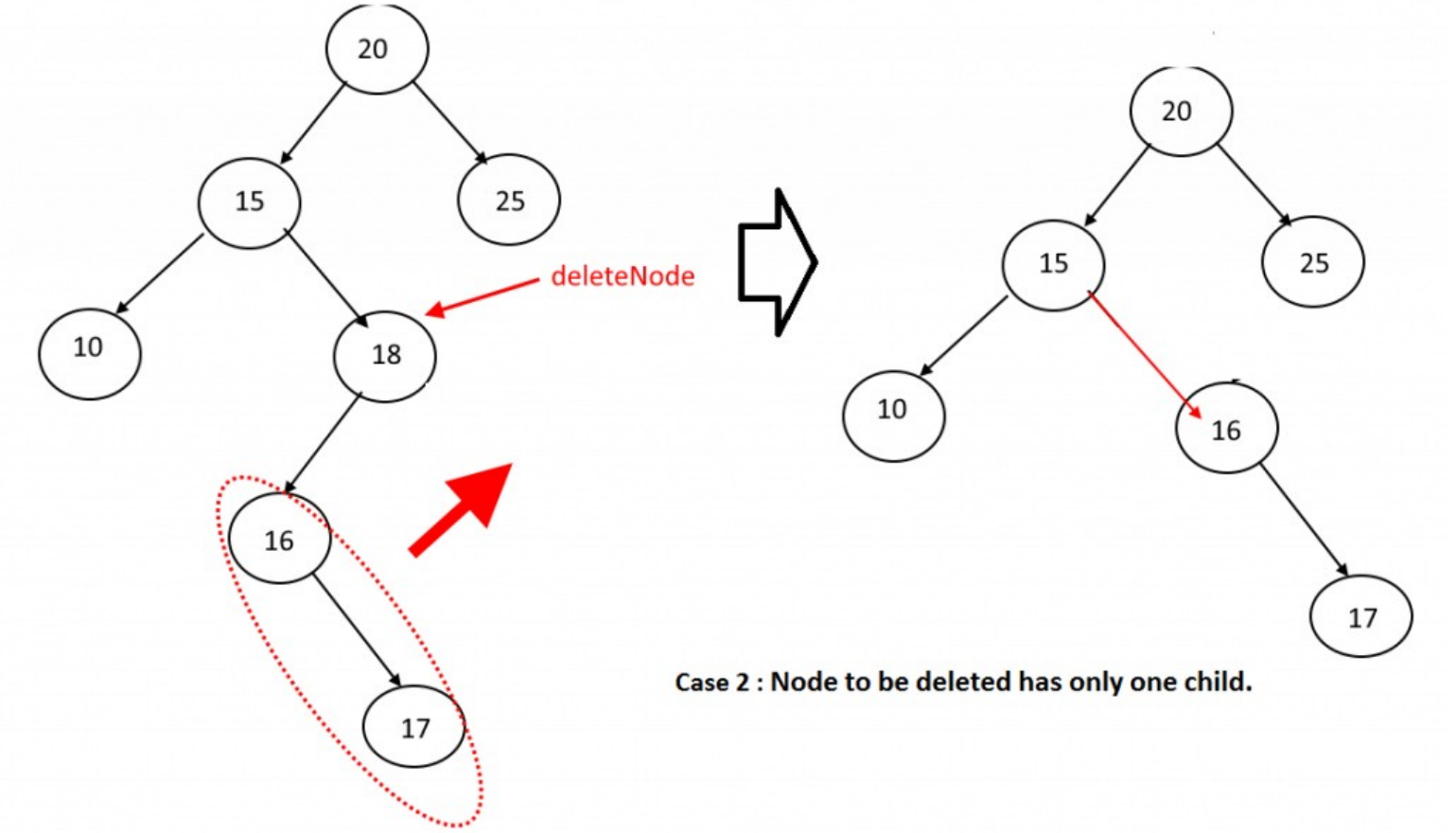
- Casos possíveis:

1) O nó onde está o elemento não tem filhos;



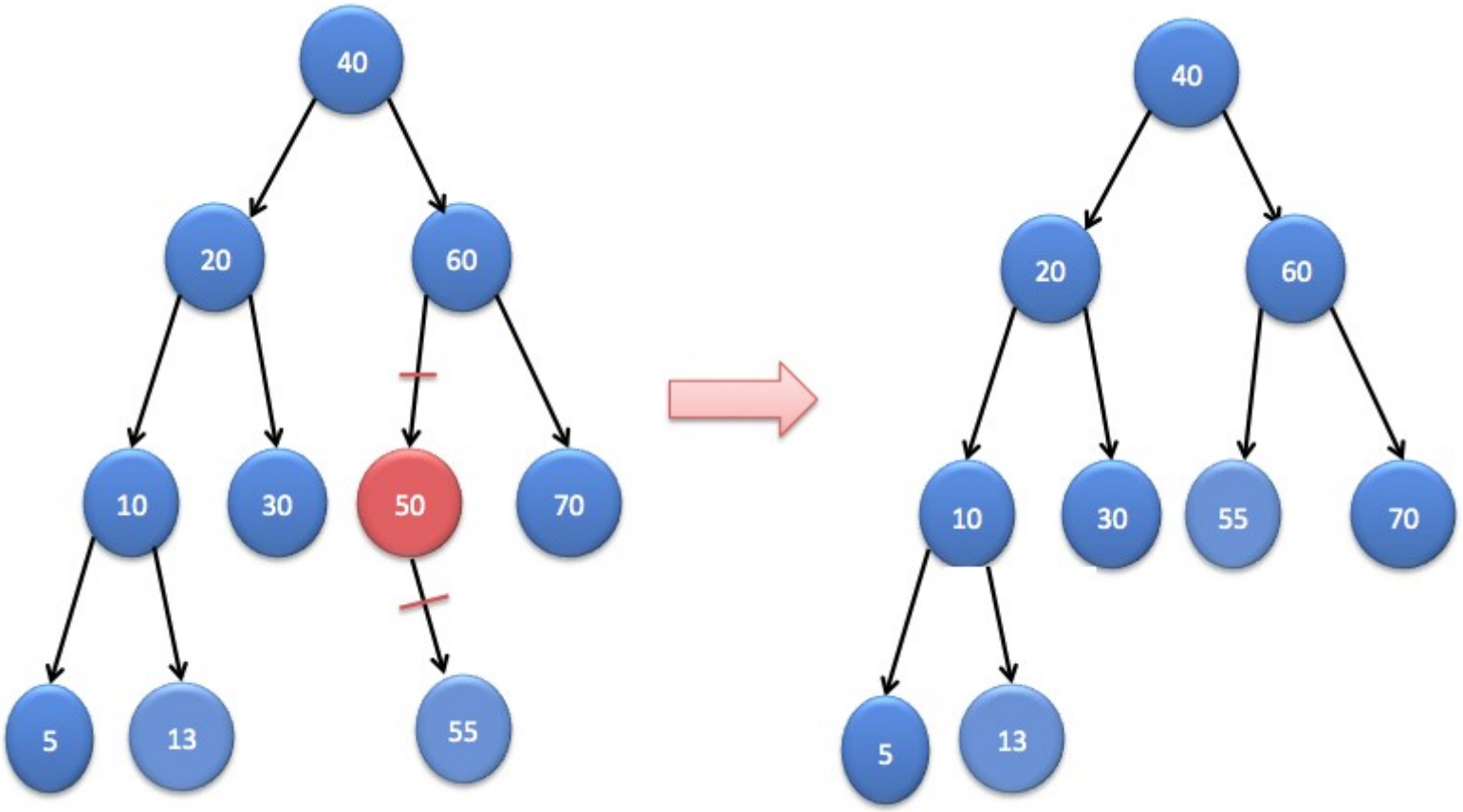
Árvore binária de pesquisa (remoção)

- Casos possíveis:
  - 2) O nó onde está o elemento tem um filho;



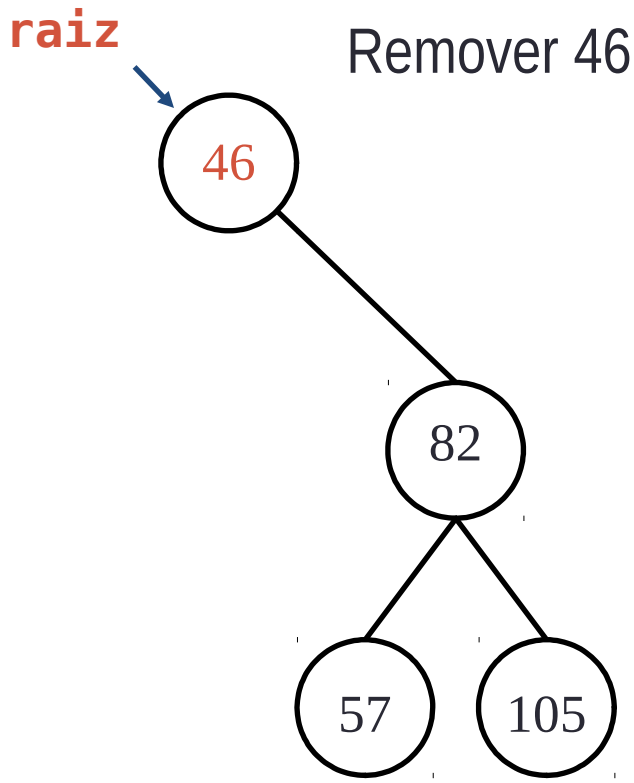
Árvore binária de pesquisa (remoção)

- Casos possíveis:
  - 2) O nó onde está o elemento tem um filho;



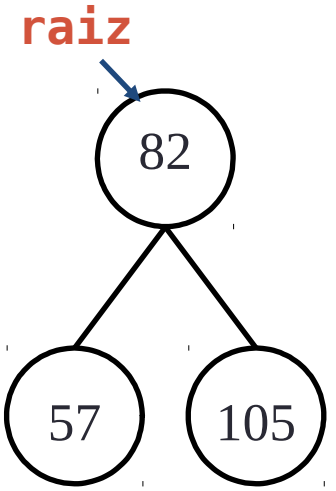
Árvore binária de pesquisa (remoção)

- Casos possíveis:
  - 2) O nó onde está o elemento tem um filho;



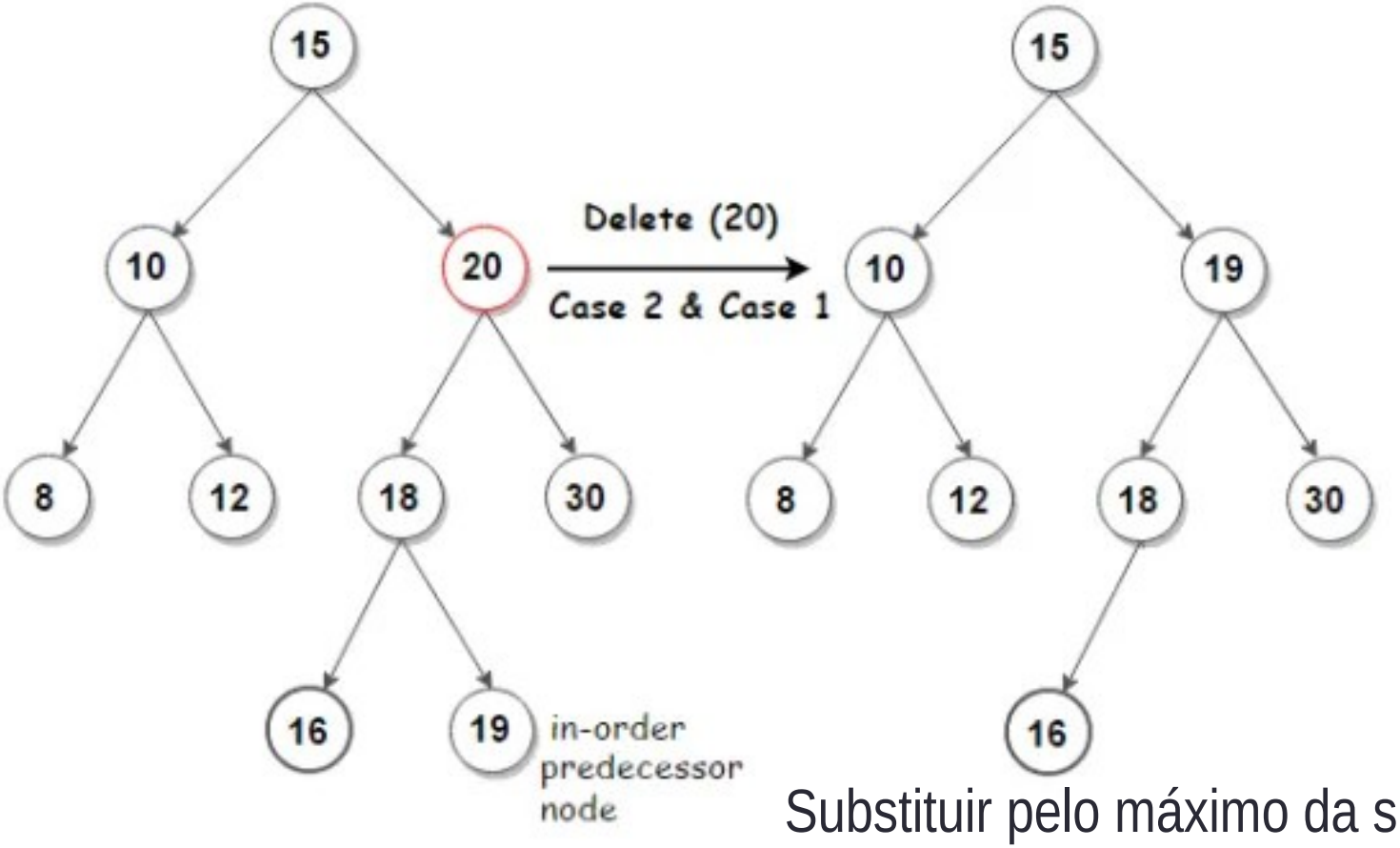
Remover 46 (raiz)

Raiz herda filho, neste caso direito



Árvore binária de pesquisa (remoção)

- Casos possíveis:
  - 3) O nó onde está o elemento tem dois filhos.

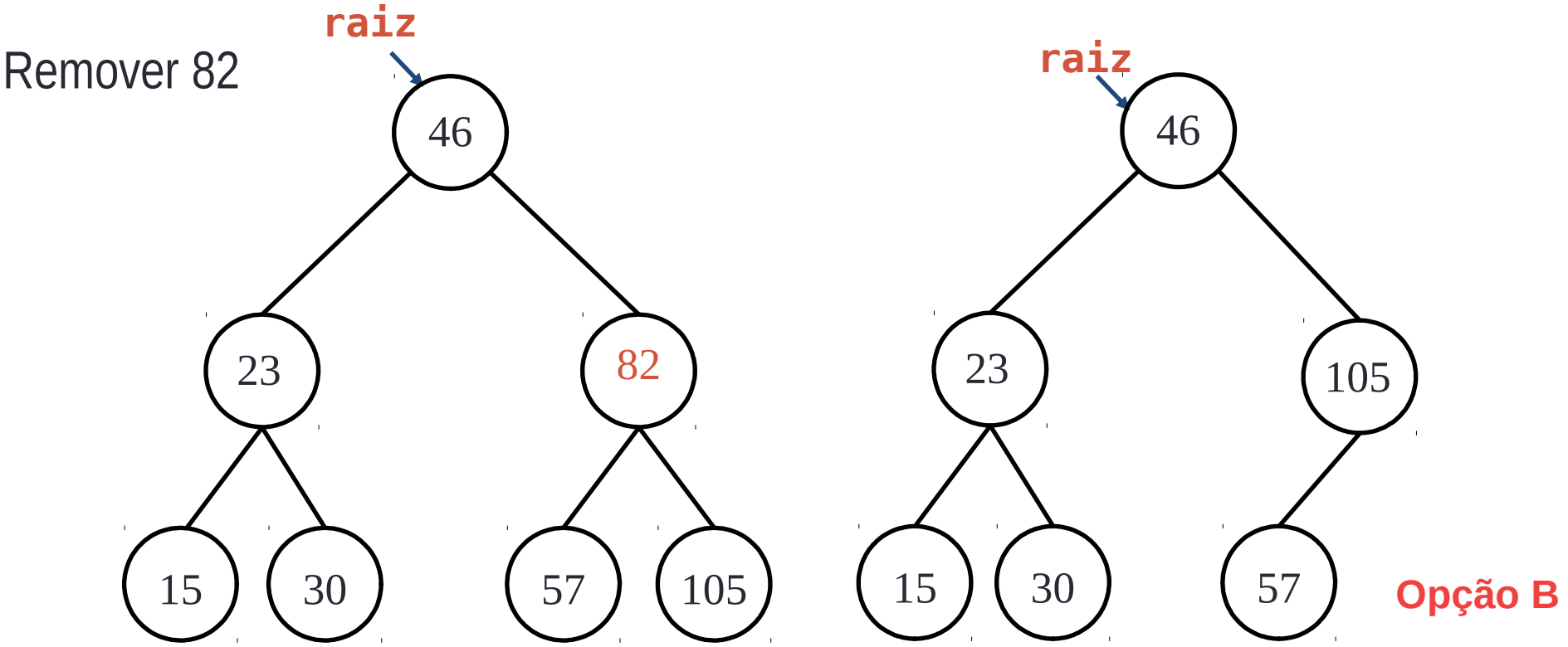


Opção A

Substituir pelo máximo da subárvore esquerda, removendo-o

Árvore binária de pesquisa (remoção)

- Casos possíveis:
  - 3) O nó onde está o elemento tem dois filhos.



Substituir pelo mínimo da subárvore direita, removendo-o

Árvore binária de pesquisa (remoção)

```
@Override
    public V remove(K key) {
    // TODO
    return null;
}
```

	Melhor Caso	Pior Caso	Caso Esperado
Remoção	$O(1)$	$O(h)$	$O(h)$

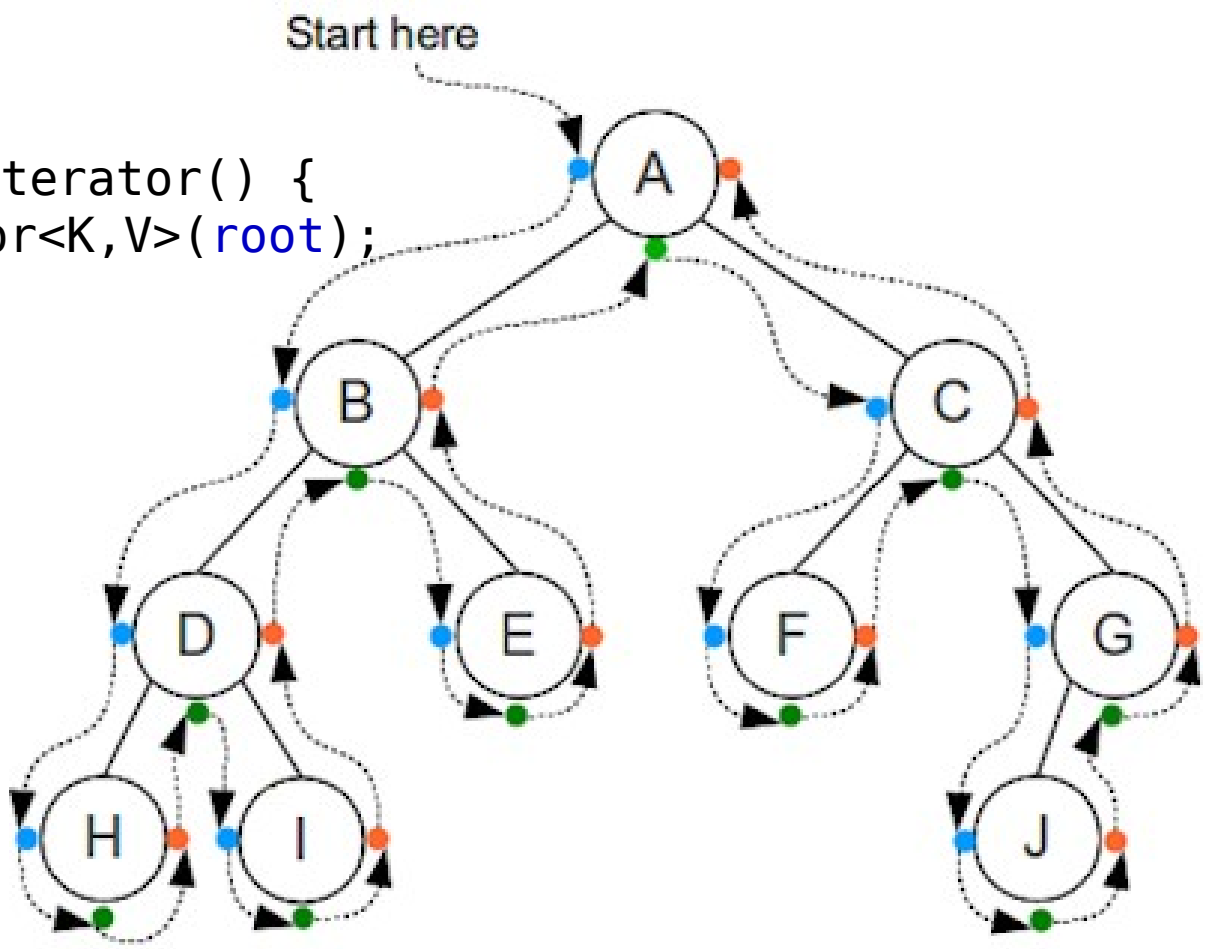
h (altura)	$O(\log n)$	$O(n)$	$O(\log n)$

Árvore binária de pesquisa
(percurso ordenado)

- Percurso infixo

```
@Override
public Iterator<Entry<K,V>> iterator() {
    return new BSTOrderIterator<K,V>(root);
}
```

Classe a implementar



Pre-Order	ABDHIECFGJ
In-Order	HDIBEAFCJG
Post-Order	HIDEBFJGCA

Que percurso ? Queremos ordenação!

Árvore binária de pesquisa

	Melhor Caso	Pior Caso	Caso Esperado
Pesquisa	$O(1)$	$O(h)$	$O(h)$
Inserção	$O(1)$	$O(h)$	$O(h)$
Remoção	$O(1)$	$O(h)$	$O(h)$
Mínimo	$O(1)$	$O(h)$	$O(h)$
Máximo	$O(1)$	$O(h)$	$O(h)$
Percurso Ordenado	$O(n)$	$O(n)$	$O(n)$

h (altura)	$O(\log n)$	$O(n)$	$O(\log n)$

Diagrama de classes
(interface **SortedMap**)