

ALGORITMOS E ESTRUTURAS DE DADOS 2023/2024 DICIONÁRIO ORDENADO

Armanda Rodrigues

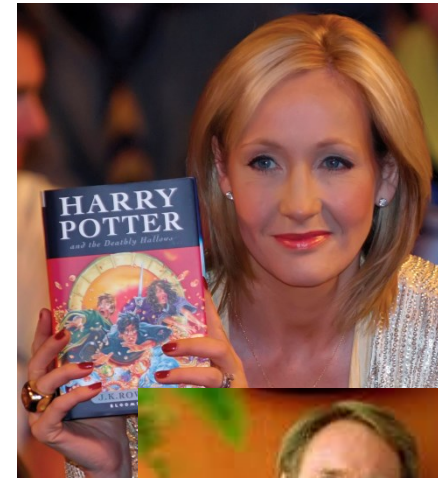
27 de outubro de 2023

Autores, Escritores

- Existem pesquisas que são habitualmente feitas nas bibliotecas e que não estão diretamente relacionadas com a chave de um documento
 - Procurar um documento cujo Título contém uma determinada String
 - Procurar todos os documentos de um determinado Autor
 - Procurar os documentos que estão associados a um determinado assunto
- Pode também ser necessário ordenar todos os objetos de um determinado tipo, utilizando a ordem dada por um dos atributos, para criar listagens formatadas
 - Listar todos os Leitores com documentos em atraso, por ordem alfabética
 - Listar todos os documentos de um determinado autor, por ordem alfabética
- Quando uma operação tem subjacente uma ordenação de objetos, a informação associada deverá ser armazenada através de uma implementação de um **Dicionário Ordenado**

Interface Biblioteca

- Vamos adicionar ao TAD Biblioteca o seguinte método:
 - `Iterator<Book> listBooksByAuthor(String author) throws NonExistingAuthorException;`
- O método deve possibilitar a pesquisa no conjunto completo de autores da Biblioteca, de um autor, por nome do mesmo
- A pesquisa do nome do autor deverá devolver a lista, ordenada alfabeticamente, de todos os livros, escritos pelo mesmo autor



Classe LibraryClass

- A implementação deste método vai implicar:
 - Adicionar uma nova estrutura de dados à `LibraryClass`, para conter todos os autores de livros da biblioteca
 - Alterar a implementação de outros métodos da classe, dada a necessidade da atualização da nova estrutura de dados
- Requisitos:
 - Chave da Pesquisa: Nome do Autor (este deverá ser único)
 - Muitos autores
 - Associado a cada autor, todos os livros por ele escritos, ordenados alfabeticamente
 - Possibilidade de inserção e remoção de livros, que se vai refletir na nova estrutura
 - Depois da remoção de todos os livros de um autor, este deverá também ser removido

TAD Dicionário Ordenado

Interface Dicionário Ordenado (K,V)

```
package dataStructures;
```

```
public interface OrderedDictionary<K extends Comparable<K>, V> extends Dictionary<K,V>{
```

```
    // Returns the entry with the smallest key in the dictionary.
```

```
    Entry<K,V> minEntry( ) throws EmptyDictionaryException;
```

```
    // Returns the entry with the largest key in the dictionary.
```


```
    Entry<K,V> maxEntry( ) throws EmptyDictionaryException;
```

```
    // Returns an iterator of the entries in the dictionary
```

```
    // which preserves the key order relation.
```

```
    // Iterator<Entry<K,V>> iterator( );
```

```
}
```

- 
- Já estava na interface Dictionary
 - Para o dicionário ordenado, este iterador deverá devolver as entradas por ordem da chave

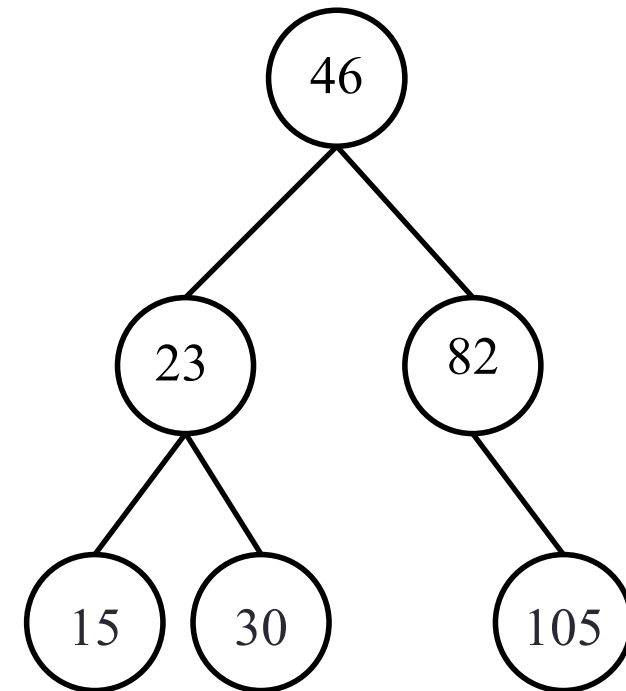
Classe de Exceções do Dicionário Ordenado

```
package dataStructures;
```

```
public class EmptyDictionaryException extends RuntimeException{  
}
```

Árvore Binária de Pesquisa (ou ordenada)

- Numa árvore binária de pesquisa, todo o nó X verifica as seguintes propriedades:
 - Qualquer nó da sub-árvore **esquerda** de X é **menor** que X ; e
 - Qualquer nó da sub-árvore **direita** de X é **maior** que X .



Classe Nó de Árvore Binária de Pesquisa (1)

```
package dataStructures;
import java.io.Serializable;

class BSTNode<K,V> implements Serializable{

    // Entry stored in the node.
    private EntryClass<K,V> entry;

    // (Pointer to) the left child.
    private BSTNode<K,V> leftChild;

    // (Pointer to) the right child.
    private BSTNode<K,V> rightChild;

    ...
}
```

Classe Nó de Árvore Binária de Pesquisa (2)

```
public BSTNode( K key, V value, BSTNode<K,V> left,  
                BSTNode<K,V> right ){  
  
    entry = new EntryClass<K,V>(key, value);  
    leftChild = left;  
    rightChild = right;  
}  
  
public BSTNode( K key, V value ){  
    this(key, value, null, null);  
}
```

Classe Nó de Árvore Binária de Pesquisa (3)

```
public EntryClass<K,V> getEntry( ){  
    return entry;  
}
```

```
public K getKey( ){  
    return entry.getKey();  
}
```

```
public V getValue( ){  
    return entry.getValue();  
}
```

Classe Nó de Árvore Binária de Pesquisa (4)

```
public BSTNode<K,V> getLeft( ){  
    return leftChild;  
}
```

```
public BSTNode<K,V> getRight( ){  
    return rightChild;  
}
```

```
public void setEntry( EntryClass<K,V> newEntry ){  
    entry = newEntry;  
}
```

Classe Nó de Árvore Binária de Pesquisa (5)

```
public void setEntry( K newKey, V newValue ){  
    entry.setKey(newKey);  
    entry.setValue(newValue);  
}
```

```
public void setKey( K newKey ){  
    entry.setKey(newKey);  
}
```

```
public void setValue( V newValue ){  
    entry.setValue(newValue);  
}
```

Classe Nó de Árvore Binária de Pesquisa (6)

```
public void setLeft( BSTNode<K,V> newLeft ){
    leftChild = newLeft;
}

public void setRight( BSTNode<K,V> newRight ){
    rightChild = newRight;
}

// Returns true iff the node is a leaf.
public boolean isLeaf( ){
    return leftChild == null && rightChild == null;
}
} // End of BSTNode.
```

Classe Árvore Binária de Pesquisa (1)

```
package dataStructures;

public class BinarySearchTree<K extends Comparable<K>, V>
    implements OrderedDictionary<K,V>{

    // The root of the tree.
    protected BSTNode<K,V> root;

    // Number of entries in the tree.
    protected int currentSize;
```

Classe Árvore Binária de Pesquisa (2)

```
public BinarySearchTree( ){  
    root = null;  
    currentSize = 0;  
}  
  
public boolean isEmpty( ){  
    return root == null;  
}  
  
public int size( ){  
    return currentSize;  
}  
.....  
}
```


Pesquisa Recursiva (1)

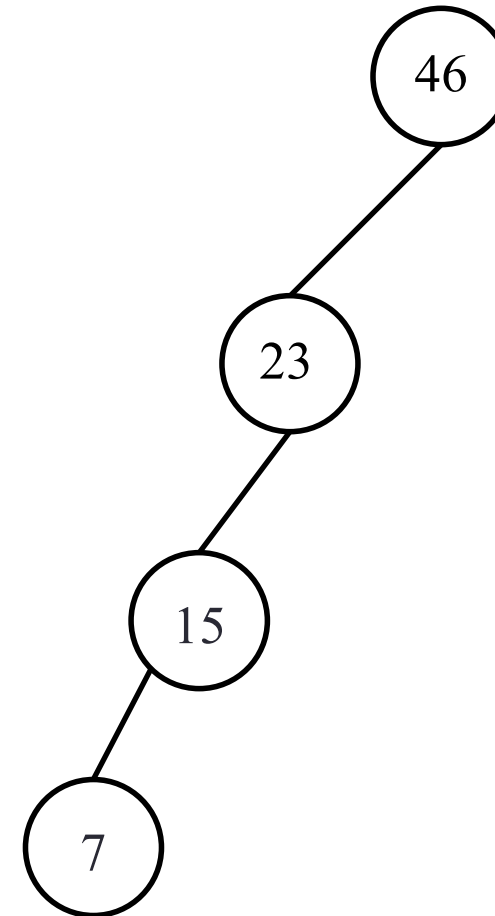
```
// If there is an entry in the dictionary whose key is the
// specified key, returns its value; otherwise, returns null.
public V find( K key ){
    BSTNode<K,V> node = this.findNode(root, key);
    if ( node == null )
        return null;
    else
        return node.getValue();
}
```

Pesquisa Recursiva (2)

```
protected BSTNode<K,V> findNode( BSTNode<K,V> node, K key ){
    if ( node == null )
        return null;
    else {
        int compResult = key.compareTo( node.getKey() );
        if ( compResult == 0 )
            return node;
        else if ( compResult < 0 )
            return this.findNode(node.getLeft(), key);
        else
            return this.findNode(node.getRight(), key);
    }
}
```

Complexidade da Pesquisa Recursiva

- Pior Caso:
 - A entrada que se procura está numa folha
 - A entrada que se procura (menor que 7) não ocorre na árvore.
- Número de chamadas recursivas do algoritmo
- Se a árvore for “uma lista”:
 - Se a árvore for vazia: zero chamadas recursivas
 - Senão, o algoritmo desce para o único filho e chama o método de pesquisa, recursivamente



Complexidade da Pesquisa Recursiva

- Pior Caso:
 - A entrada que se procura está numa folha
 - A entrada que se procura não ocorre na árvore.

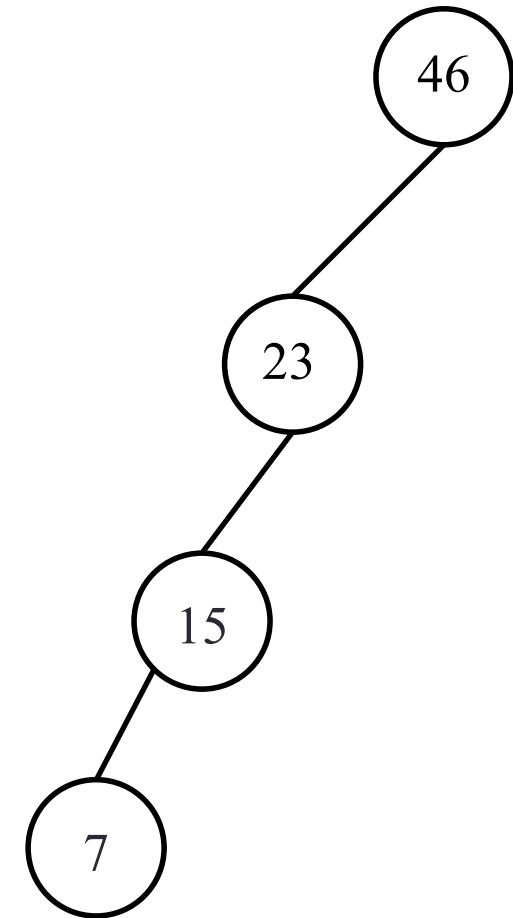
$$\text{numCR}(n) = \begin{cases} 0, & n = 0 \\ \text{numCR}(n - 1) + 1, & n \geq 1 \end{cases}$$

Aplica-se a Recorrência 1

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

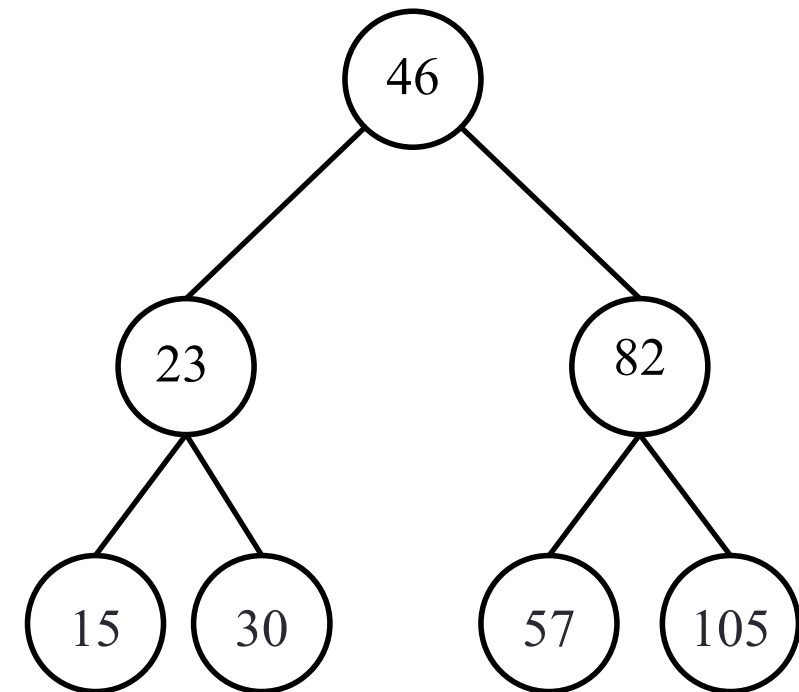
com $a \geq 0$, $b \geq 1$, $c \geq 1$ constantes

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



Complexidade da Pesquisa Recursiva

- Pior Caso:
 - A entrada que se procura está numa folha
 - A entrada que se procura não ocorre na árvore.
- Número de chamadas recursivas do algoritmo
- Se a árvore “estiver equilibrada”:
 - Se a árvore for vazia: zero chamadas recursivas
 - Senão, o algoritmo desce para o filho adequado e chama o método de pesquisa, recursivamente
 - Neste caso, a zona de pesquisa da árvore é reduzida a metade



Complexidade da Pesquisa Recursiva

- Pior Caso:

- A entrada que se procura está numa folha
- A entrada que se procura não ocorre na árvore.

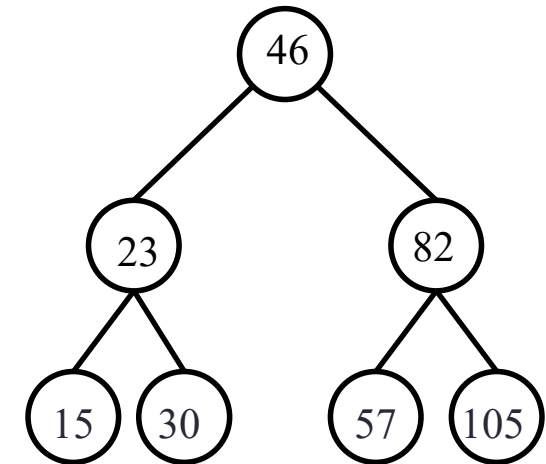
$$numCR(n) = \begin{cases} 0, & n = 0 \\ numCR\left(\frac{n}{2}\right) + 1 & n \geq 1 \end{cases}$$

Aplica-se a Recorrência 2 (a)

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

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

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



Mínimo Recursivo

```
// Returns the entry with the smallest key in the dictionary.
public Entry<K,V> minEntry( ) throws EmptyDictionaryException{
    if ( this.isEmpty() )
        throw new EmptyDictionaryException();
    return this.minNode(root).getEntry();
}

// Requires: node != null.
protected BSTNode<K,V> minNode( BSTNode<K,V> node ){
    if ( node.getLeft() == null )
        return node;
    else
        return this.minNode( node.getLeft() );
}
```

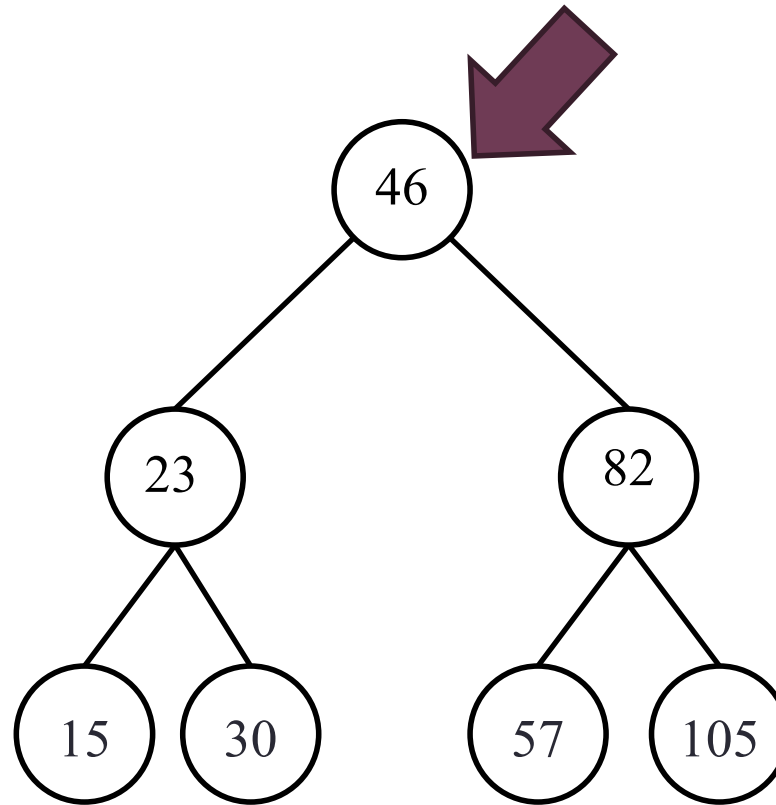
Máximo Recursivo

```
// Returns the entry with the largest key in the dictionary.
public Entry<K,V> maxEntry( ) throws EmptyDictionaryException{
    if ( this.isEmpty() )
        throw new EmptyDictionaryException();
    return this.maxNode(root).getEntry();
}

// Requires: node != null.
protected BSTNode<K,V> maxNode( BSTNode<K,V> node ){
    if ( node.getRight() == null )
        return node;
    else
        return this.maxNode( node.getRight() );
}
```

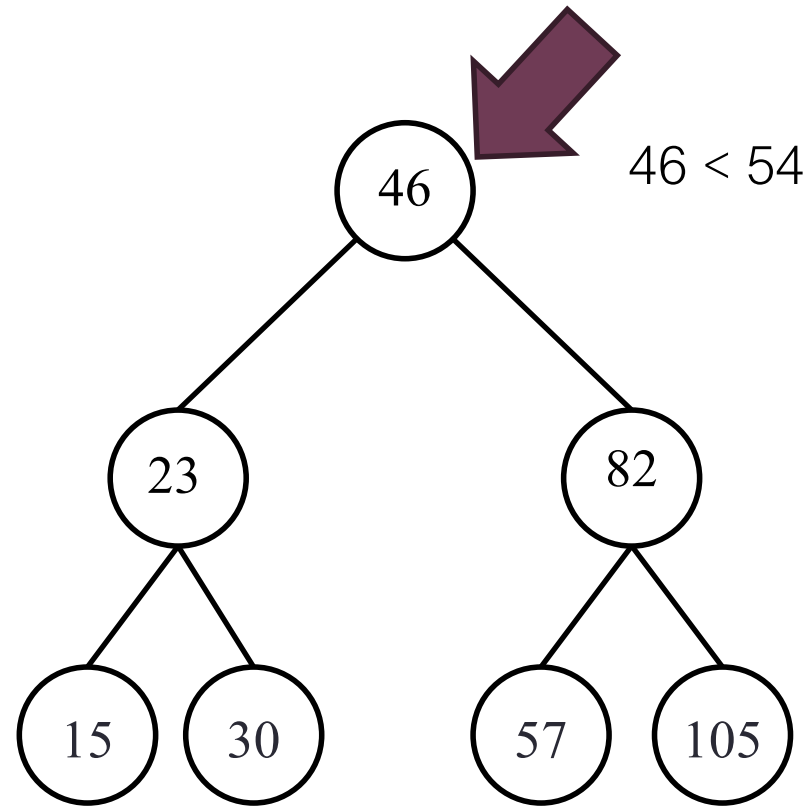

Inserção

Inserir 54



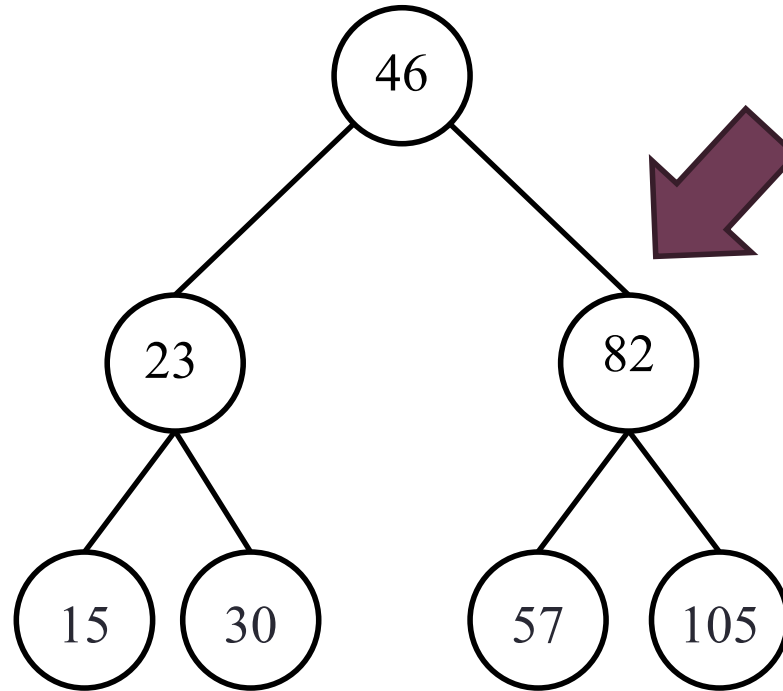
Inserção

Inserir 54



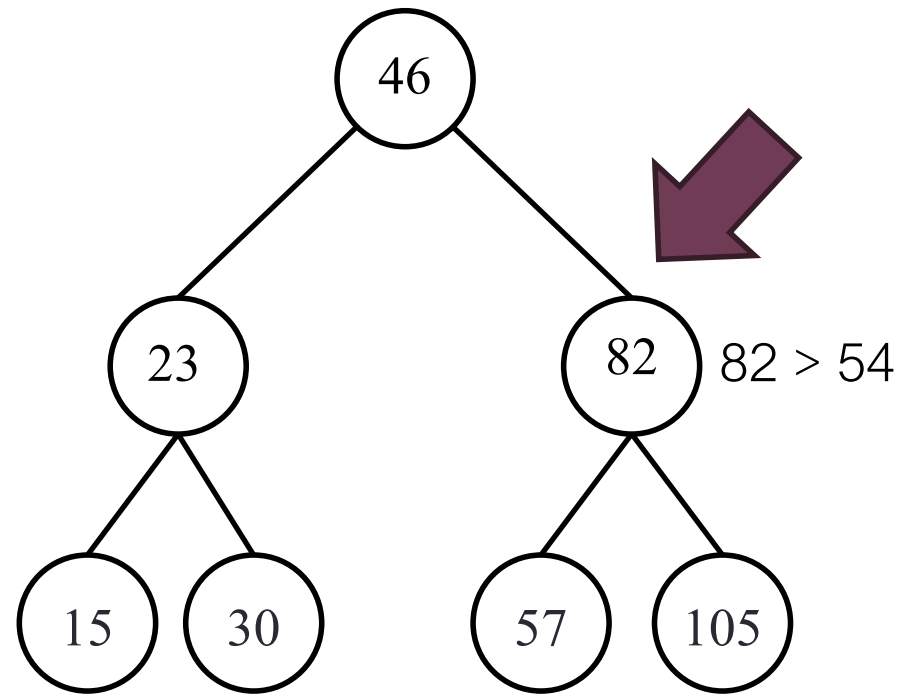
Inserção

Inserir 54



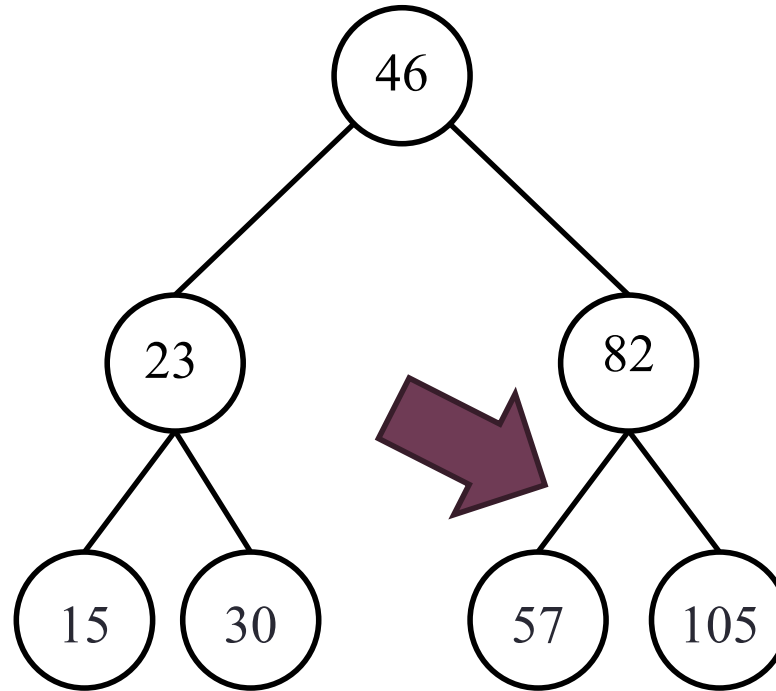
Inserção

Inserir 54



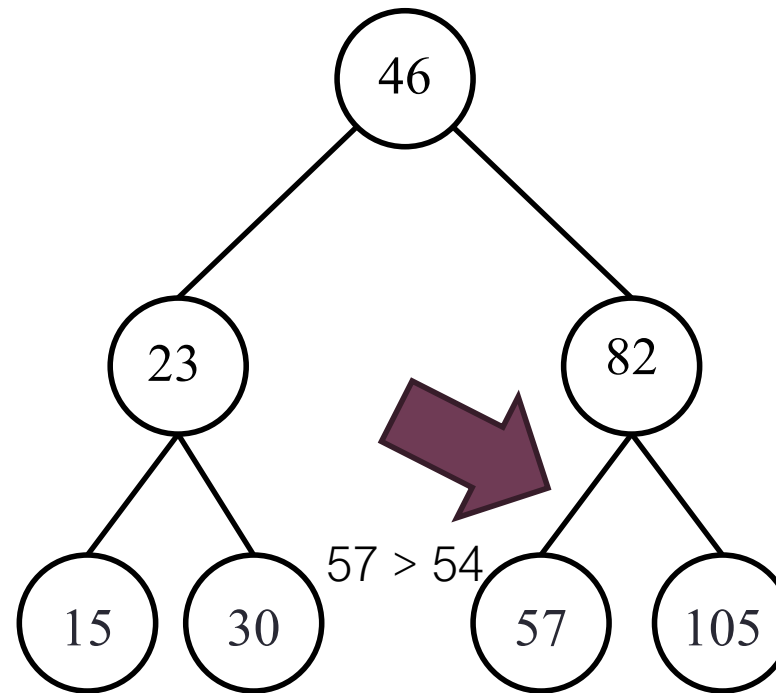
Inserção

Inserir 54



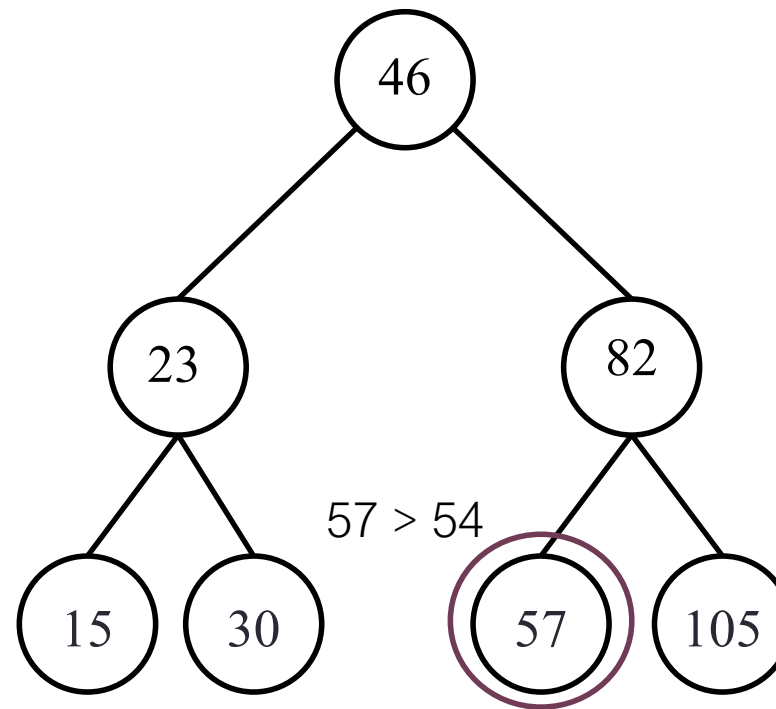
Inserção

Inserir 54



Inserção

Inserir 54



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

Classe Auxiliar Passo do Caminho Percorrido

```
protected static class PathStep<K,V>{
```

```
// The parent of the node.
```

```
public BSTNode<K,V> parent;
```

```
// The node is the left or the right child of parent.
```

```
public boolean isLeftChild;
```

```
public PathStep( BSTNode<K,V> theParent, boolean toTheLeft ){
    parent = theParent; isLeftChild = toTheLeft;
}
```

```
public void set( BSTNode<K,V> newParent, boolean toTheLeft ){
    parent = newParent; isLeftChild = toTheLeft;
}
```

```
}
```

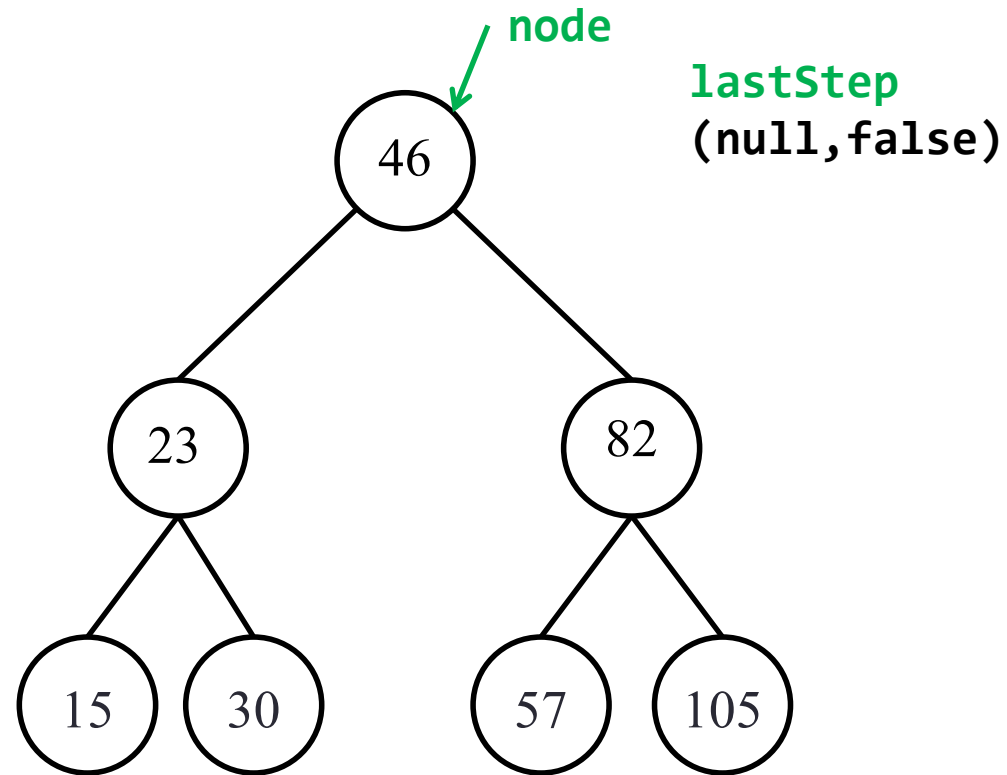
- O objeto irá referir-se a um nó.
- **parent** será o pai do nó,
- **isLeftChild** será a direção tomada (filho esquerdo ou direito) a partir de parent.

Pesquisa Iterativa que Guarda o Último Passo

```
protected BSTNode<K,V> findNode( K key, PathStep<K,V> lastStep ){
    BSTNode<K,V> node = root;
    while ( node != null ){
        int compResult = key.compareTo( node.getKey() );
        if ( compResult == 0 )
            return node;
        else if ( compResult < 0 )
            { lastStep.set(node, true); node = node.getLeft(); }
        else
            { lastStep.set(node, false); node = node.getRight(); }
    }
    return null;
}
```

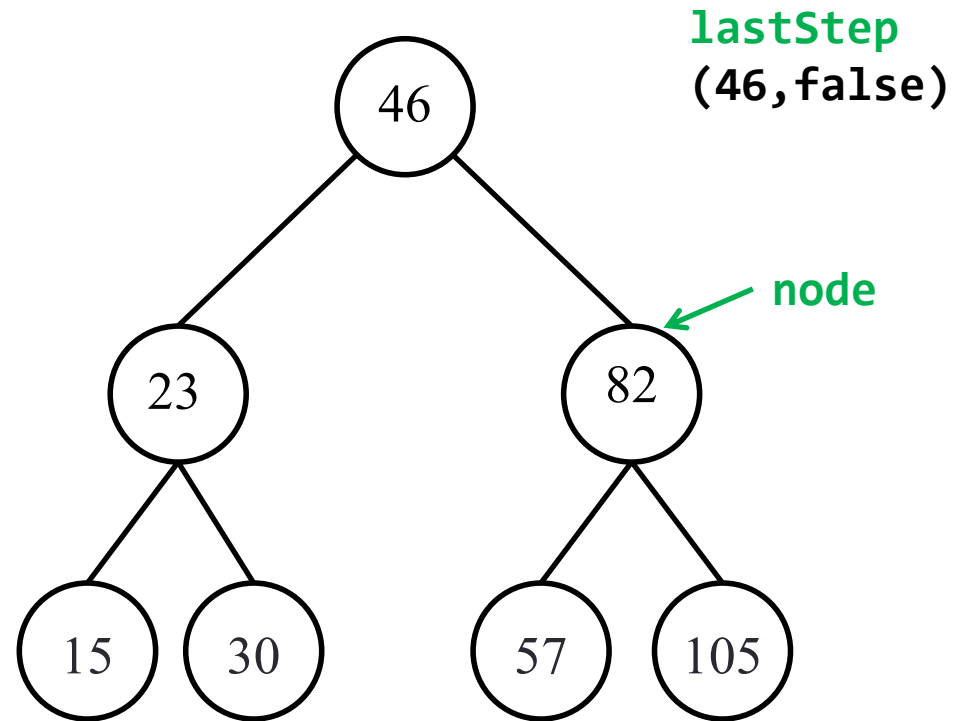
Pesquisa que guarda o último passo

Procurar 54



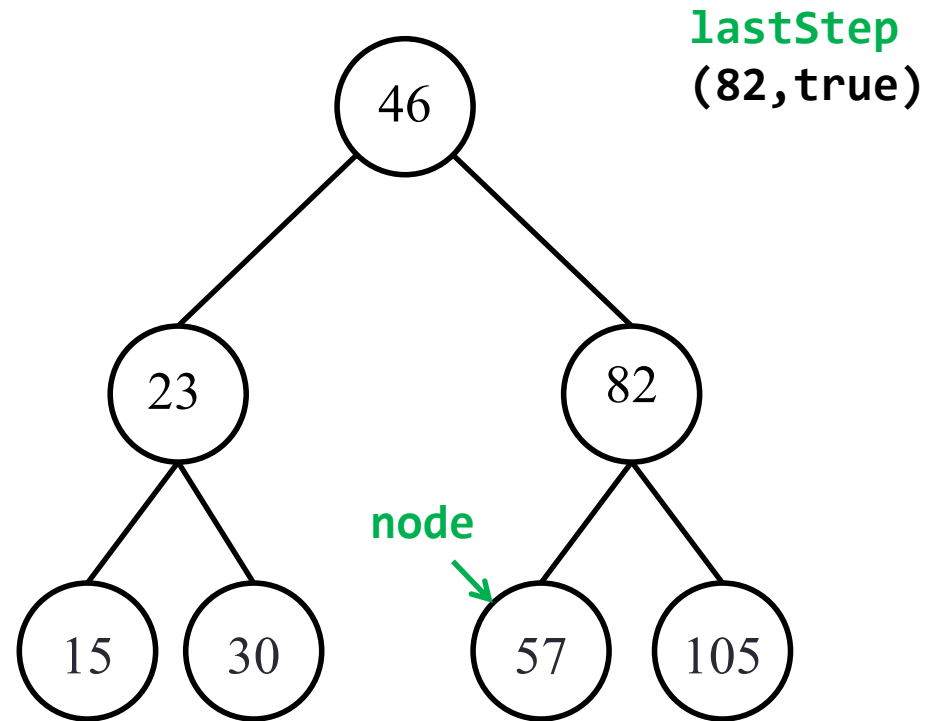
Pesquisa que guarda o último passo

Procurar 54



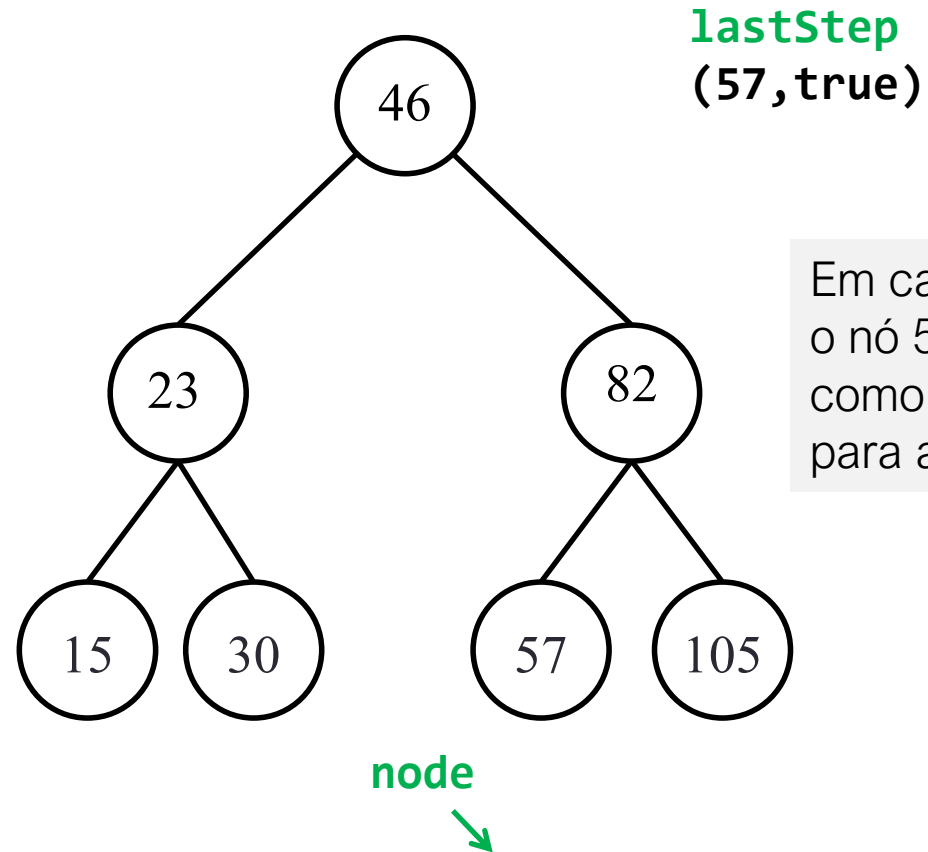
Pesquisa que guarda o último passo

Procurar 54



Pesquisa que guarda o último passo

Procurar 54



Inserção

```
public V insert( K key, V value ){
    PathStep<K,V> lastStep = new PathStep<K,V>(null, false);
    BSTNode<K,V> node = this.findNode(key, lastStep);
    if ( node == null ){
        BSTNode<K,V> newLeaf = new BSTNode<K,V>(key, value);
        this.linkSubtree(newLeaf, lastStep);
        currentSize++;
        return null;
    }
    else{
        V oldValue = node.getValue();
        node.setValue(value);
        return oldValue;
    }
}
```

Método que insere newLeaf como descendente do antecessor guardado em lastStep com a direção (esquerda ou direita) guardada

Ligar uma Subárvore a Árvore

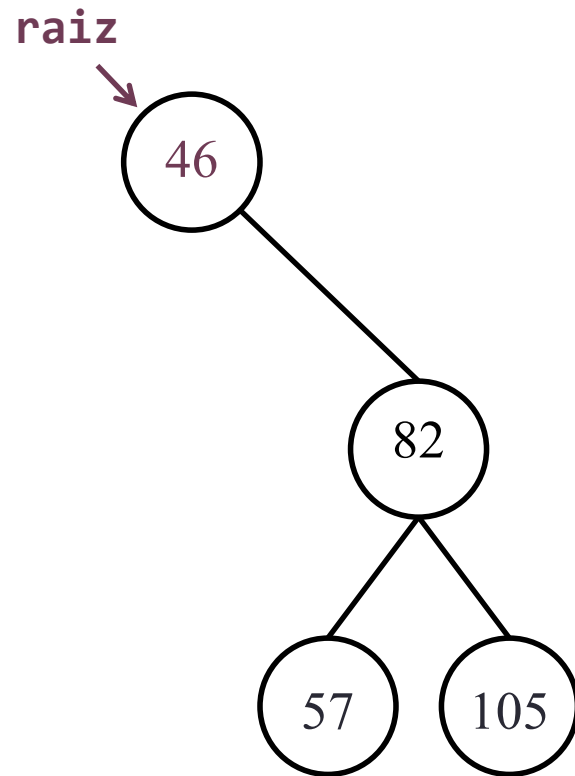
```
// Links a new subtree, rooted at the specified node, to the tree.
// The parent of the old subtree is stored in lastStep.
protected void linkSubtree( BSTNode<K,V> node,
                           PathStep<K,V> lastStep){
    if ( lastStep.parent == null )
        // Change the root of the tree.
        root = node;
    else
        // Change a child of parent.
        if ( lastStep.isLeftChild )
            lastStep.parent.setLeft(node);
        else
            lastStep.parent.setRight(node);
}
```

Remoção de nó

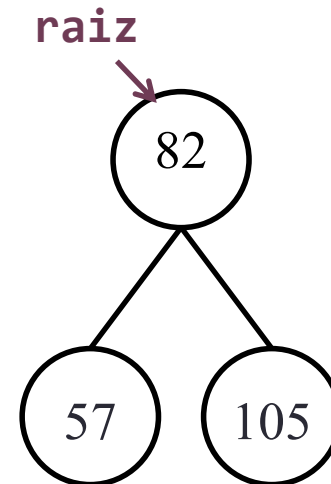
- Existem várias possibilidades que devem ser consideradas
 - Remoção da raiz, quando esta não tem filho esquerdo
 - Remoção de nó (sendo o mesmo um filho esquerdo de um outro nó), quando este não tem filho esquerdo
 - Remoção de nó (sendo o mesmo um filho direito de outro nó), quando este não tem filho esquerdo
 - Remoção da raiz, quando esta não tem filho direito
 - Remoção de nó (sendo o mesmo um filho esquerdo de um outro nó), quando este não tem filho direito
 - Remoção de nó (sendo o mesmo um filho direito de outro nó), quando este não tem filho direito
 - Remoção da raiz, quando esta tem dois filhos
 - Remoção de um nó (sem ser a raiz), quando este tem dois filhos

Remover raiz sem filho esquerdo

Remover 46 (raiz)

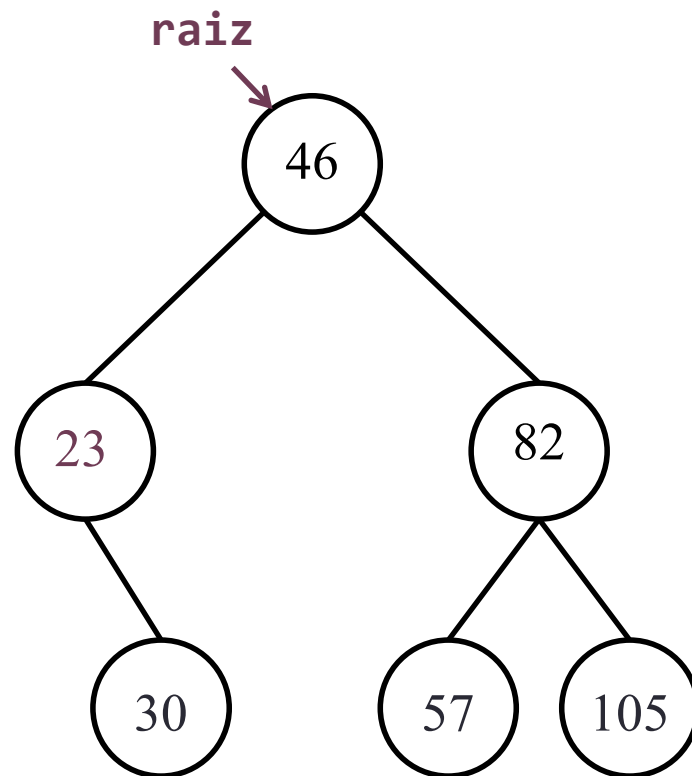


Raiz herda filho direito

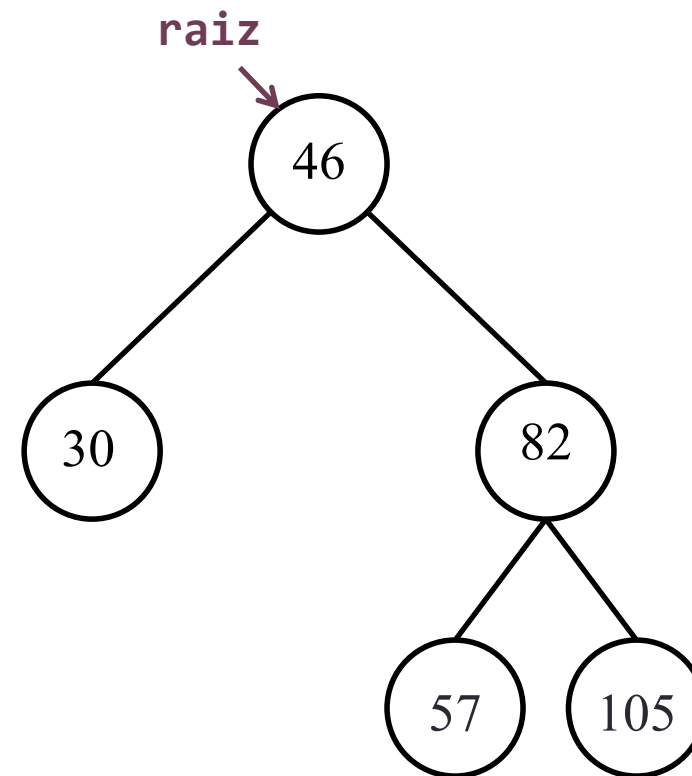


Remover Nó FE sem filho esquerdo

Remover 23 (filho esquerdo de 46)

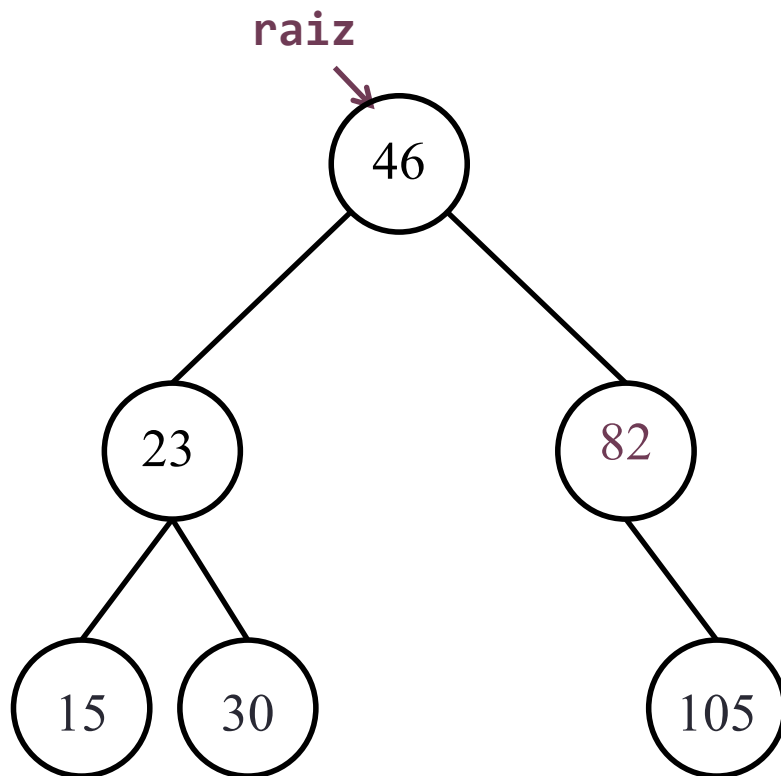


Pai herda filho direito

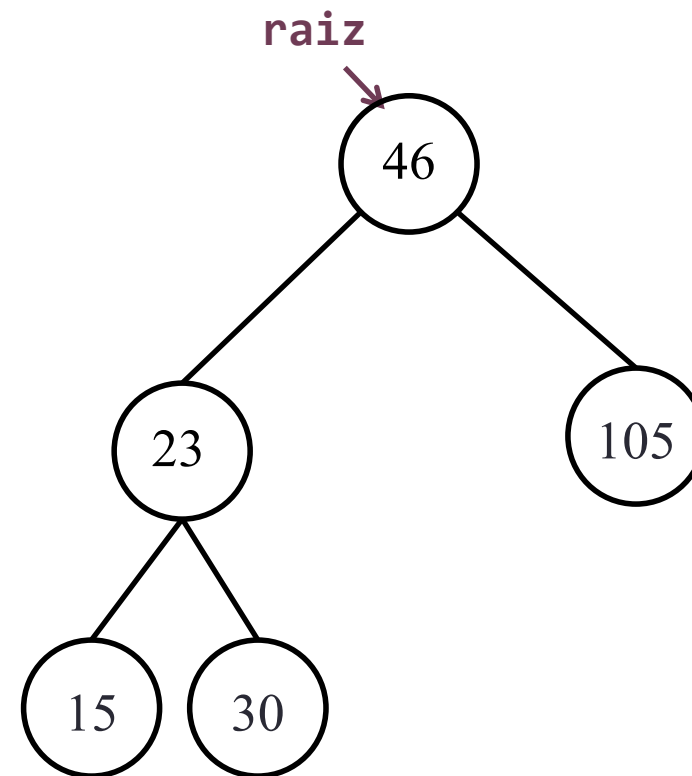


Remover Nó FD sem filho esquerdo

Remover 82 (filho direito de 46)



Pai herda filho direito



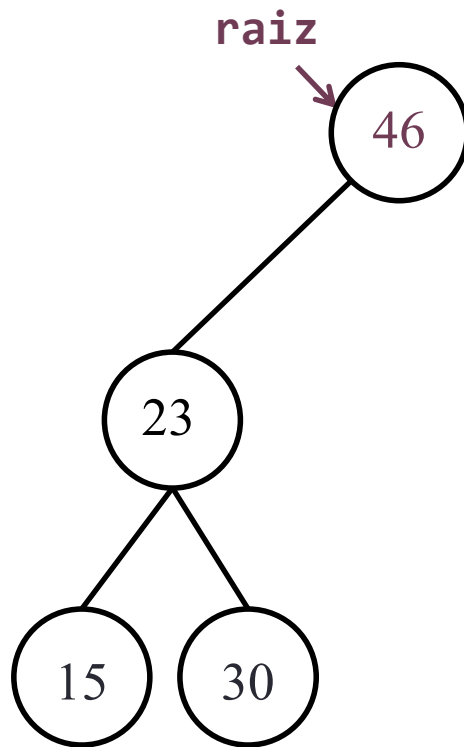
Remoção de nó

- Remoção da raiz, quando esta não tem filho esquerdo
- Remoção de nó (sendo o mesmo um filho esquerdo de um outro nó), quando este não tem filho esquerdo
- Remoção de nó (sendo o mesmo um filho direito de outro nó), quando este não tem filho esquerdo

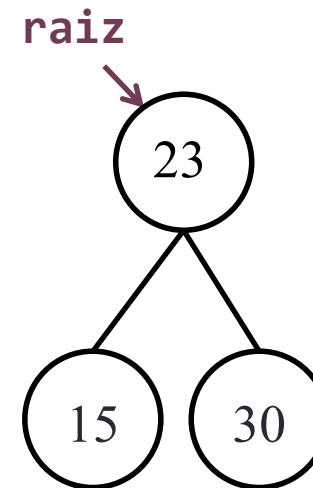
Solução: Pai (ou raiz)
herda filho direito

Remover raiz sem filho direito

Remover 46 (raiz)

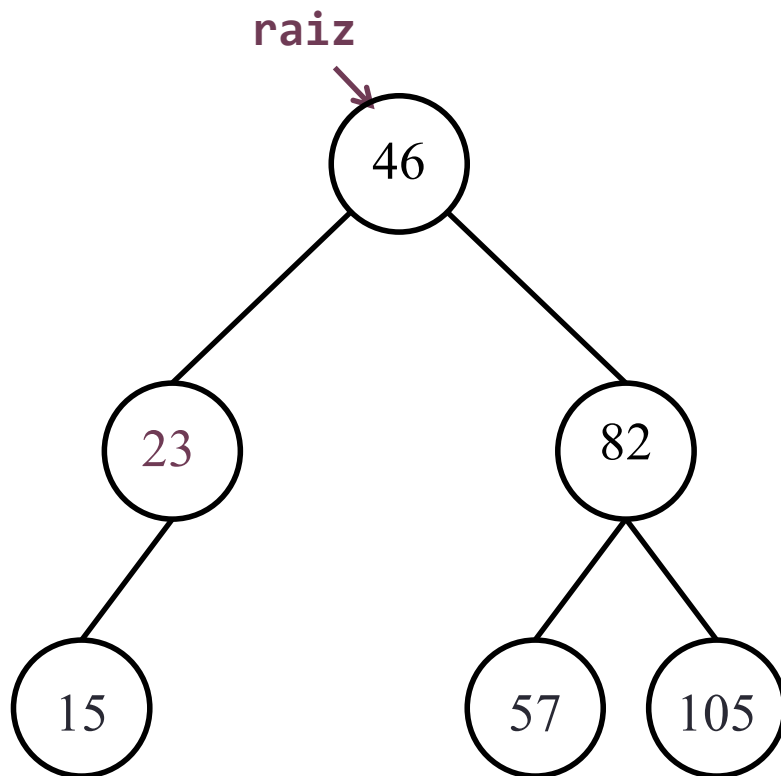


Raiz herda filho esquerdo

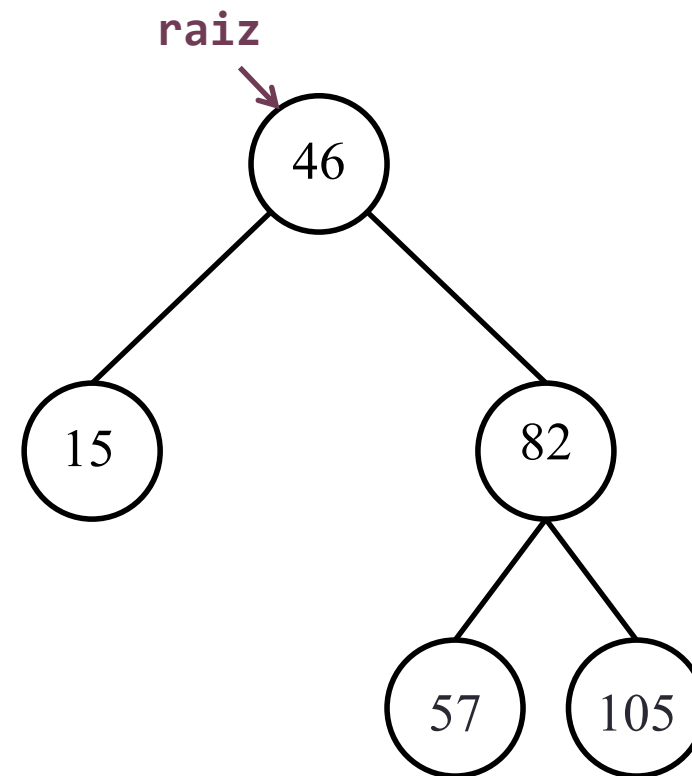


Remover Nó FE sem filho direito

Remover 23 (filho esquerdo de 46)

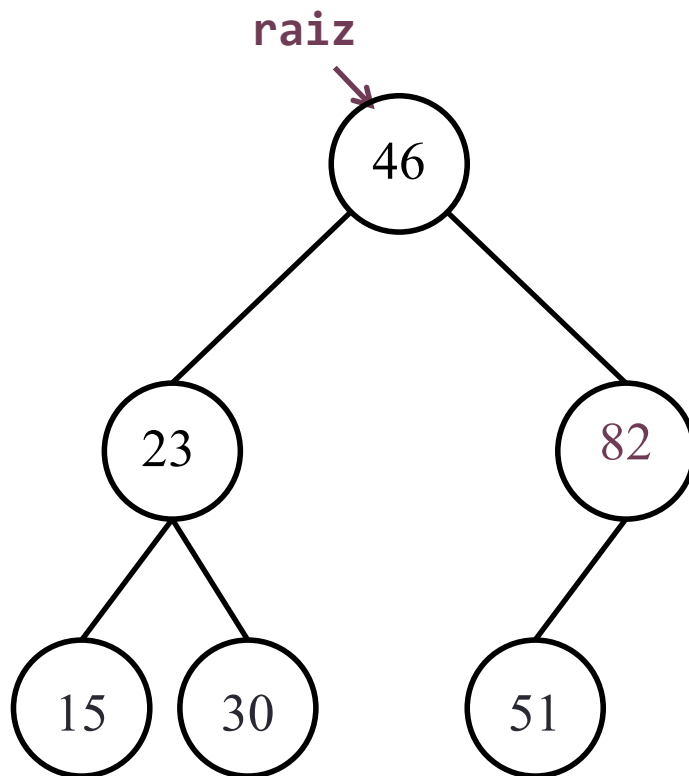


Pai herda filho esquerdo

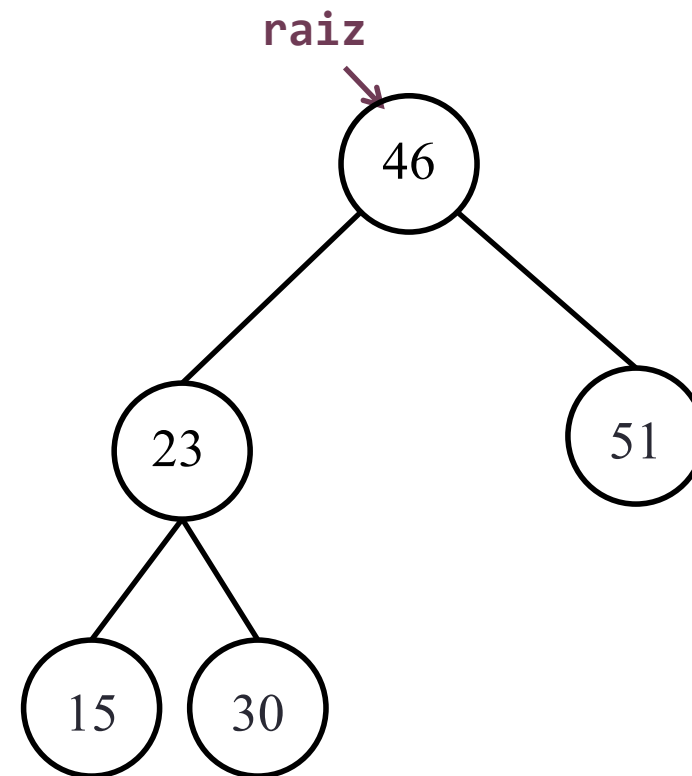


Remover Nó FD sem filho direito

Remover 82 (filho direito de 46)



Pai herda filho esquerdo



Remoção de nó

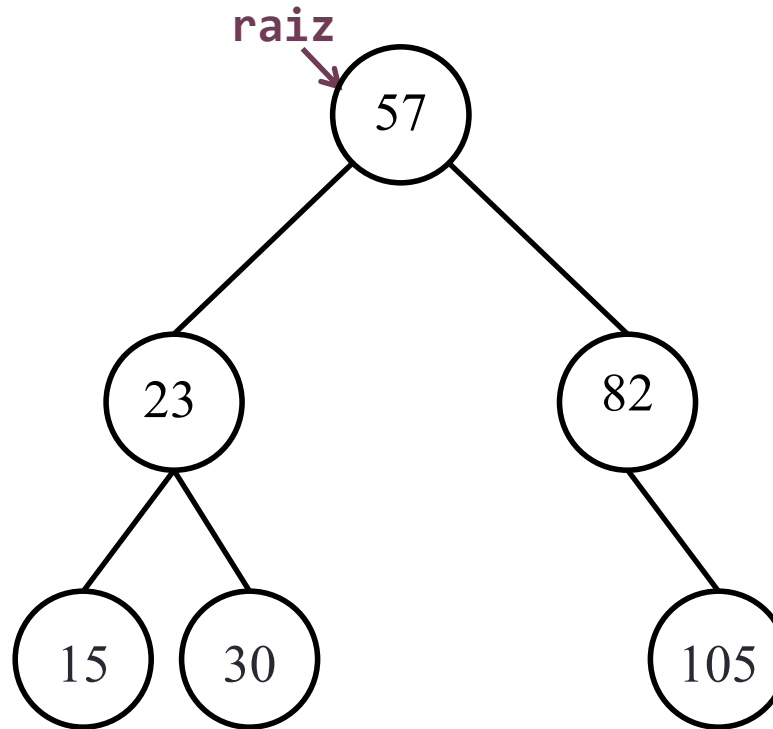
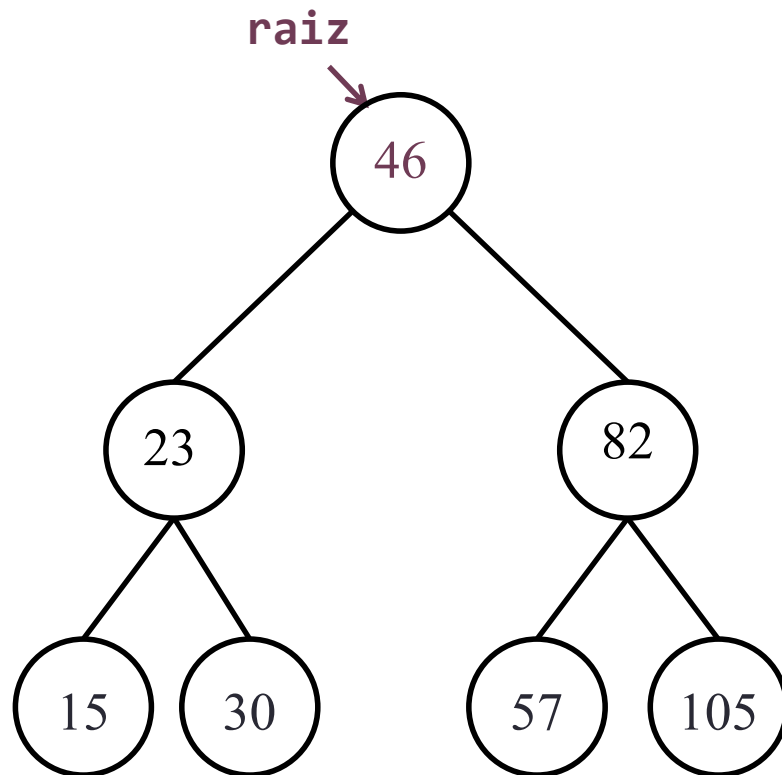
- Remoção da raiz, quando esta não tem filho direito
- Remoção de nó (sendo o mesmo um filho esquerdo de um outro nó), quando este não tem filho direito
- Remoção de nó (sendo o mesmo um filho direito de outro nó), quando este não tem filho direito

Solução: Pai (ou raiz)
herda filho esquerdo

Remover Nó com dois filhos – Caso 1

Remover 46 (raiz)

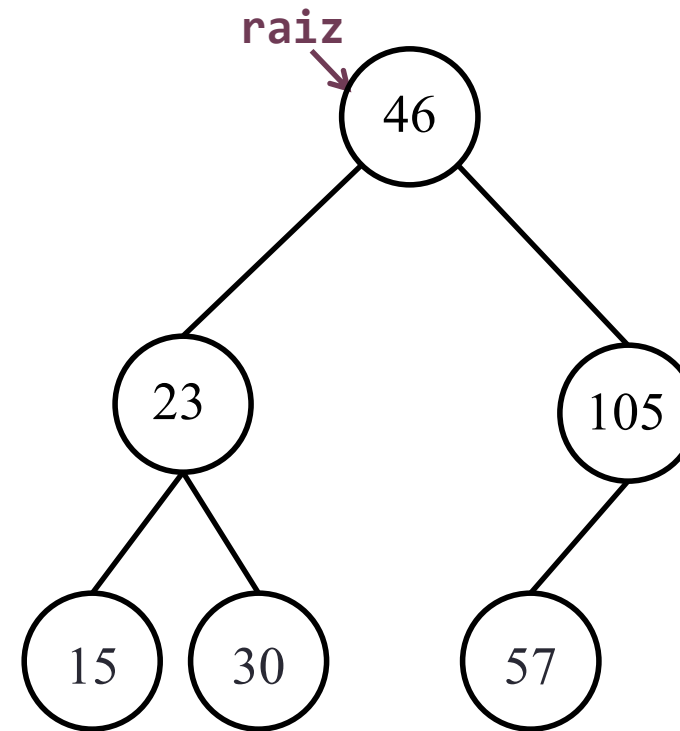
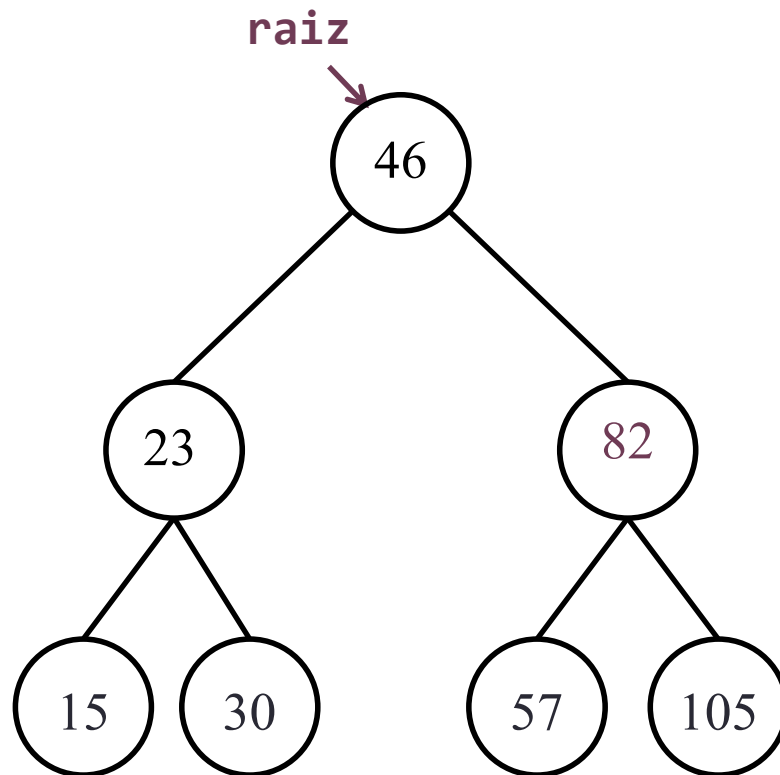
Substituir pelo mínimo da subárvore
direita, removendo-o



Remover Nó com dois filhos – Caso 2

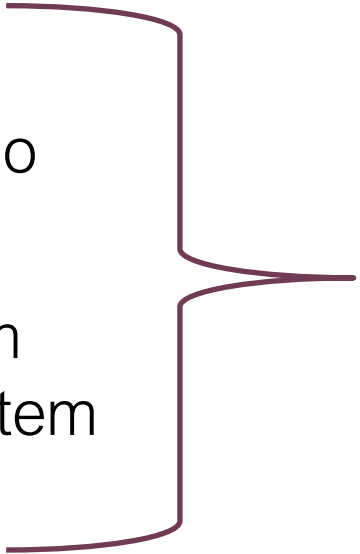
Remover 82

Substituir pelo mínimo da subárvore direita, removendo-o



Remoção de nó

- Remoção da raiz, quando esta tem dois filhos
- Remoção de um nó (sem ser a raiz), quando este tem dois filhos



Solução: Substituir pelo mínimo da subárvore direita, removendo-o.

Mínimo Iterativo que Guarda o Último Passo

```
// Returns the node with the smallest key
// in the tree rooted at the specified node.
// Moreover, stores the last step of the path in lastStep.
// Requires: theRoot != null.
protected BSTNode<K,V> minNode( BSTNode<K,V> theRoot,
                                PathStep<K,V> lastStep ){
    BSTNode<K,V> node = theRoot;
    while ( node.getLeft() != null ){
        lastStep.set(node, true);
        node = node.getLeft();
    }
    return node;
}
```

Remoção (1)

```
// 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 ){
    PathStep<K,V> lastStep = new PathStep<K,V>(null, false);
    BSTNode<K,V> node = this.findNode(key, lastStep);
    if ( node == null )
        return null;
    else {
        V oldValue = node.getValue();
        // Remover a entrada de node.
        currentSize--;
        return oldValue;
    }
}
```

Slide Seguinte

Remoção (2)

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

Liga-se o pai com o filho direito do nó a remover

Remoção (2)

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

Liga-se o pai com o filho esquerdo do nó a remover

Remoção (2)


```
...
else
{
    // Node has 2 children. Replace the node's entry with
    // the 'minEntry' of the right subtree.
    lastStep.set(node, false);
    BSTNode<K,V> minNode = this.minNode(node.getRight(), lastStep);
    node.setEntry( minNode.getEntry() );
    // Remove the 'minEntry' of the right subtree.
    this.linkSubtree(minNode.getRight(), lastStep);
}
```

node passa a ser o nó
pai, o seu conteúdo
será alterado

Remoção (2)

```
...  
else  
{  
    // Node has 2 children. Replace the node's entry with  
    // the 'minEntry' of the right subtree.  
    lastStep.set(node, false);  
    BSTNode<K,V> minNode = this.minNode(node.getRight(), lastStep);  
    node.setEntry( minNode.getEntry() );  
    // Remove the 'minEntry' of the right subtree.  
    this.linkSubtree(minNode.getRight(), lastStep);  
}
```

Procura-se o mínimo da subárvore direita do nó a remover, guardando o pai



Remoção (2)

```
...  
else  
{  
    // Node has 2 children. Replace the node's entry with  
    // the 'minEntry' of the right subtree.  
    lastStep.set(node, false);  
    BSTNode<K,V> minNode = this.minNode(node.getRight(), lastStep);  
    node.setEntry( minNode.getEntry() );  
    // Remove the 'minEntry' of the right subtree.  
    this.linkSubtree(minNode.getRight(), lastStep);  
}
```

O conteúdo de node é alterado com a entrada do mínimo encontrado (na subárvore direita)

Remoção (2)

```
...
else
{
    // Node has 2 children. Replace the node's entry with
    // the 'minEntry' of the right subtree.
    lastStep.set(node, false);
    BSTNode<K,V> minNode = this.minNode(node.getRight(), lastStep);
    node.setEntry( minNode.getEntry() );
    // Remove the 'minEntry' of the right subtree.
    this.linkSubtree(minNode.getRight(), lastStep);
}
```

Remove-se o mínimo da subárvore direita ligando a subárvore direita do mesmo à sua nova localização

Porque é que podemos fazer isto?

Remoção (2)

```
...
else
{
    // Node has 2 children. Replace the node's entry with
    // the 'minEntry' of the right subtree.
    lastStep.set(node, false);
    BSTNode<K,V> minNode = this.minNode(node.getRight(), lastStep);
    node.setEntry( minNode.getEntry() );
    // Remove the 'minEntry' of the right subtree.
    this.linkSubtree(minNode.getRight(), lastStep);
}
```

Remove-se o mínimo da subárvore direita ligando a sub-árvore direita do mesmo à sua nova localização

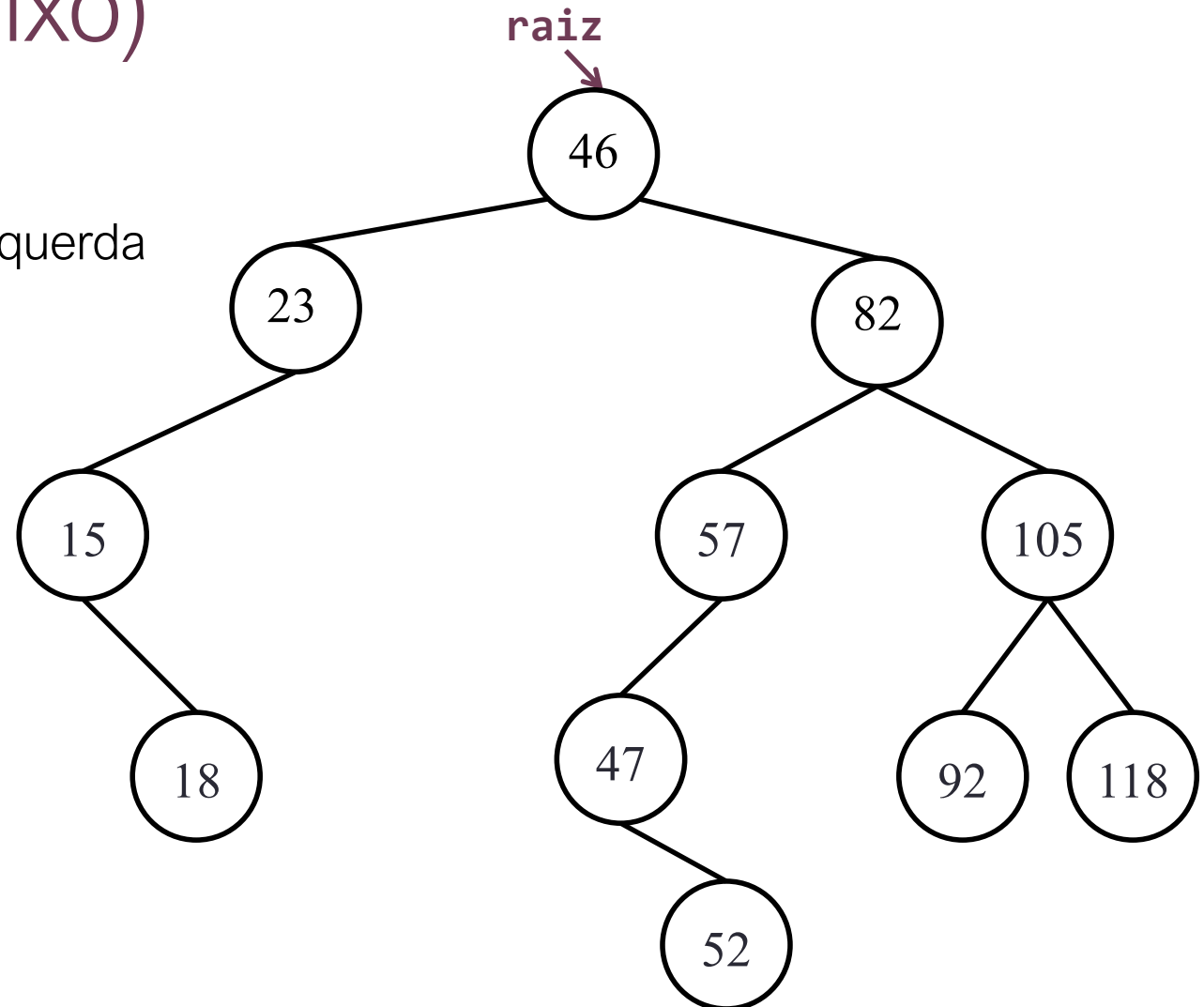
O mínimo da sub-árvore direita, tem sub-árvore esquerda ?

Percurso Ordenado (Infixo)

15
18
23
46
47
52
57
82
92
105
118

Infixo da subárvore esquerda

Infixo da subárvore direita



Percurso Ordenado

```
// Returns an iterator of the entries in the dictionary  
// which preserves the key order relation.  
public Iterator<Entry<K,V>> iterator( ){  
    return new BSTKeyOrderIterator<K,V>(root);  
}
```

Complexidades de Árvore Binária de Pesquisa (com n nós)

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

Interface Biblioteca

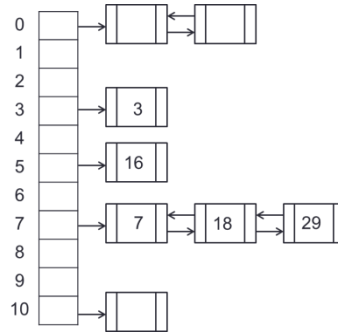
- Vamos adicionar ao TAD Biblioteca o seguinte método:
 - `Iterator<Book> listBooksByAuthor(String author) throws NonExistingAuthorException;`
- O método deve possibilitar a pesquisa no conjunto completo de autores da Biblioteca, de um autor, por nome do mesmo
- A pesquisa do nome do autor deverá devolver a lista, ordenada alfabeticamente, de todos os livros, escritos pelo mesmo autor



Classe LibraryClass

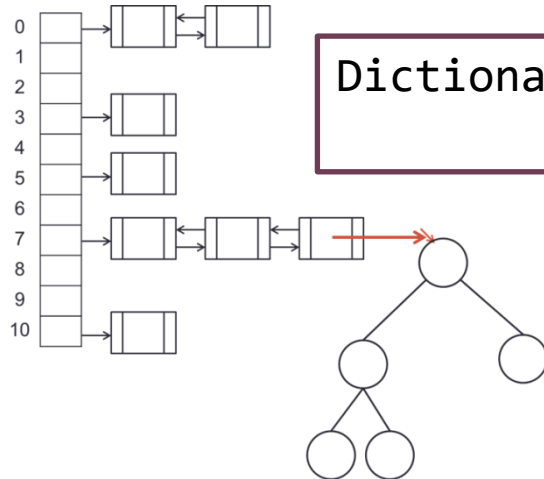
- A implementação deste método vai implicar:
 - Adicionar uma nova estrutura de dados à `LibraryClass`, para conter todos os autores de livros da biblioteca
 - Alterar a implementação de outros métodos da classe, dada a necessidade da atualização da nova estrutura de dados
- Requisitos:
 - Chave da Pesquisa: Nome do Autor (este deverá ser único)
 - Muitos autores
 - Associado a cada autor, todos os livros por ele escritos, ordenados alfabeticamente
 - Possibilidade de inserção e remoção de livros, que se vai refletir na nova estrutura
 - Depois da remoção de todos os livros de um autor, este deverá também ser removido

Classe LibraryClass – Que Entry <K,V> ?



Dictionary<String,Document> **documents**

Se o autor não tiver dados pessoais associados



Dictionary <String,
OrderedDictionary<String, Document>>**authors**

Se existir o TAD Author

Dictionary <String, Author> **authors**

LibraryClass – incompleta (1)

```
public class LibraryClass implements Library {  
  
    private Dictionary<String,Document> documents;  
    private CopyReturn returns;  
    private Dictionary<String, OrderedDictionary<String, Document>>  
        authors;  
  
    public LibraryClass(int docCapacity, int authorCapacity){  
        documents=  
            new SepChainHashTable<String,Document>(docCapacity);  
        authors =  
            new SepChainHashTable<String,  
                OrderedDictionary<String,Document>>(authorCapacity);  
        returns=new CopyReturnClass();  
    }  
}
```

Novo!

LibraryClass – incompleta (2)

```
public void addNewBook(String title, String subject,
    String documentCode, String publisher, String author,
    long ISBN) throws ExistingDocException{

    Document doc;
    if (documents.find(documentCode) != null)
        throw new ExistingDocException();
    else {
        doc = new BookClass(title, subject, documentCode,
                               publisher, author, ISBN);
        documents.insert(documentCode, doc);
        OrderedDictionary<String,Document> authorBST
            = authors.find(author);
        if (authorBST == null){
            authorBST = new BinarySearchTree<String,Document>();
            authors.insert(author, authorBST);
        }
        authorBST.insert(title, doc);
    }
}
```

LibraryClass – incompleta (3)

```
public Document removeDocument(String documentCode)
    throws NonExistingDocException{

    Document doc = documents.remove(documentCode);
    if (doc == null)
        throw new NonExistingDocException();
    else {
        if (doc instanceof Book){
            Book b = (Book)doc;
            OrderedDictionary<String,Document> authorBST
                = authors.find(b.getAuthor());
            authorBST.remove(b.getTitle());
            if (authorBST.isEmpty())
                authors.remove(b.getAuthor());
        }
        return doc;
    }
}
```

LibraryClass – incompleta (4)

```
public Iterator<Book> listBooksByAuthor(String author)
    throws NonExistingAuthorException {

    OrderedDictionary<String,Document> authorBST
        = authors.find(author);
    if (authorBST == null)
        throw new NonExistingAuthorException();
    else return new BookIterator(authorBST.iterator());
}
```

Interface Iterador de Elementos do Tipo E

```
package dataStructures;
public interface Iterator<E>{

    // Returns true iff the iteration has more elements.
    // In other words, returns true if a call to next()
    // would return an element instead of throwing an exception.
    boolean hasNext( );

    // Returns the next element in the iteration.
    E next( ) throws NoSuchElementException;

    // Restarts the iteration.
    // After rewind, if the iteration is not empty,
    // next() will return the first element in the iteration.
    void rewind( );
}
```

Interface Dicionário Ordenado (K,V)

```
package dataStructures;

public interface OrderedDictionary<K extends Comparable<K>, V>
    extends Dictionary<K,V>{

    // Returns the entry with the smallest key in the dictionary.
    Entry<K,V> minEntry( ) throws EmptyDictionaryException;

    // Returns the entry with the largest key in the dictionary.
    Entry<K,V> maxEntry( ) throws EmptyDictionaryException;

    // Returns an iterator of the entries in the dictionary
    // which preserves the key order relation.
    // Iterator<Entry<K,V>> iterator( );

}
```


BookIterator (1)

```
package library;

import dataStructures.*;

public class BookIterator implements Iterator<Book> {

    Iterator<Entry<String,Document>> itBST;

    public BookIterator(Iterator<Entry<String,Document>> it){
        itBST = it;
        rewind();
    }

    public void rewind() {
        itBST.rewind();
    }
}
```

BookIterator (2)

```
public boolean hasNext() {  
    return itBST.hasNext();  
}  
  
public Book next() throws NoSuchElementException {  
    Entry<String, Document> ent = itBST.next();  
    return (Book)ent.getValue();  
}  
}
```

Desafio

- Se pretendemos utilizar uma ordenação específica, numa árvore binária de pesquisa
 - Exemplo 1:
 - Uma árvore de entradas `Entry<Integer, Client>` onde a chave é o número de compras efetuadas pelo cliente num site de compras.
 - Pretendemos ordenar a árvore de forma descendente começando no cliente que efetuou mais compras e terminando naquele que efetuou menos compras (imaginando que não há empates)
 - Exemplo 2:
 - No exemplo anterior da árvore de livros escritos por um autor
 - Pretendemos que os livros, cuja chave é o título do livro, sejam ordenados de forma independente da utilização de maiúsculas ou minúsculas
 - `OrderedDictionary<Entry<String, Document> authorBST = new BinarySearchTree<String,Document>();`

BST with Comparator

- Estender a Árvore Binária de pesquisa com um construtor que recebe um comparador
- Interface Comparator<E>

```
public interface Comparator<E>
```

```
    // Compares its two arguments for order.  
    // Returns a negative integer, zero or a positive integer  
    // as the first argument is less than, equal to, or greater  
    // than the second.
```

```
    int compare( E element1, E element2 );
```

```
}
```

BST with Comparator

- Comparador para ordenar BST com chave inteira
- Com este comparador a ordenação gerada para a árvore dá prioridade a chaves maiores

```
public class InvertIntegerCompare implements Comparator<Integer> {  
    public int compare( Integer key1, Integer key2) {  
        return key2-key1;  
    }  
}
```

BST with Comparator

- Estender a Árvore Binária de pesquisa com um construtor que recebe um comparador

```
public class BSTWithComparator<K extends Comparable<K>, V> extends BinarySearchTree<K,V> {
```

```
    //Comparator to order BST
```

```
    protected Comparator<K> comparator;
```

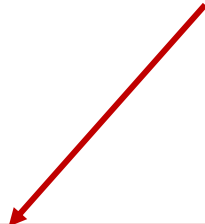
```
    public BSTWithComparator(Comparator<K> comparator) {
```

```
        super();
```

```
        this.comparator = comparator;
```

```
    }
```

Será necessário redefinir `findNode`, de forma a usar o comparador na comparação de chaves



```
    // Returns the node whose key is the specified key;
```

```
    // or null if no such node exists.
```

```
    // Moreover, stores the last step of the path in lastStep.
```

```
    protected BSTNode<K,V> findNode( K key, PathStep<K,V> lastStep ){
```

```
    }
```

```
}
```