Iterators and Binary Search Trees

Iterators

- Remember that an **iterator** is an object that allows you to access the elements of a collection, one at a time
- Java's built-in **Iterator** interface specifies that any iterator implementation must support three methods:
  - **hasNext()**: returns whether the iterator has more elements to visit
  - **next()**: returns the iterator's next element, and advances the iterator to the subsequent element
  - **remove()**: removes the last element returned from the iterator (optional operation – may throw an **UnsupportedOperationException** if not implemented)

Iterators and linked lists

(Using **java.util.LinkedList**, **java.util.Iterator**)

```
LinkedList<String> stuff = new LinkedList<>();
// add elements to the list here...

// create an iterator object over the list
Iterator<String> it = stuff.iterator();

// use the iterator to access the list elements, one
//  at a time
while (it.hasNext())
   System.out.println(it.next());
```

Iterators and BSTs

One of the nice things about iterators is that they provide a consistent way of accessing the elements in a collection:

```
BinarySearchTree<String> stuff = new BinarySearchTree<>();
// add elements to the BST here...

// create an iterator object over the BST
Iterator<String> it = stuff.iterator();

// use the iterator to access the BST elements, one
//  at a time
while (it.hasNext())
   System.out.println(it.next());
```

Implementing a BST iterator

- We can make a nested class within **BinarySearchTree** that implements the **Iterator** interface (i.e., defines **hasNext()**, **next()**, and **remove()**)
  - For simplicity let's just make **remove()** throw an **UnsupportedOperationException**
- We'll focus on an **in-order iterator**, which means the nodes are visited in the same order as an in-order traversal
  - You could write pre-order/post-order iterators as well!

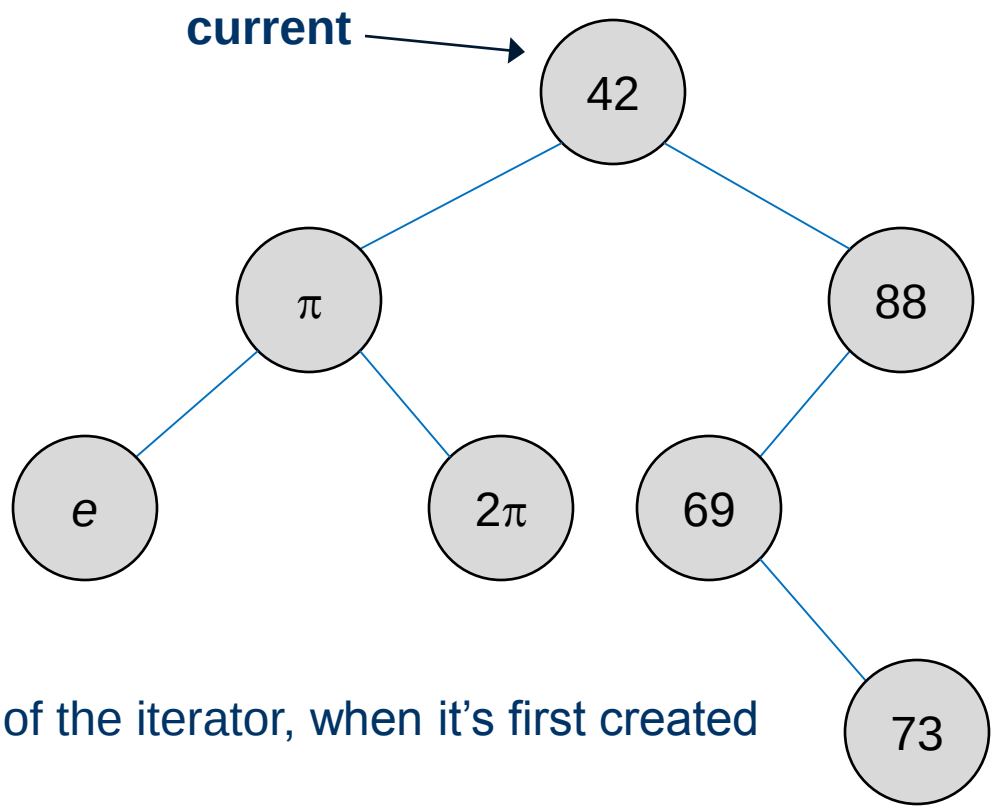
In-order BST iterator

- Maintain the following info:
  - The current **Node** where the iterator is located (this is initialized to the root of the tree)
  - A stack to track which **Node** to visit next
- Each time **next()** is called:
  - Push the current node onto the stack
  - Advance the current node to its left child
  - Repeat the previous two steps until the current node is **null**
  - Pop the top element from the stack
  - Set the current node to the popped node's right child
  - Return the popped element

In-order BST iterator

- Each time **hasNext()** is called, check for two things:
  - Is the current node **null**?
  - Is the stack empty?
- If both are true, then we've gone through the entire tree, and **hasNext()** can return **false**
  - Otherwise there are still more elements left, and **hasNext()** should return **true**

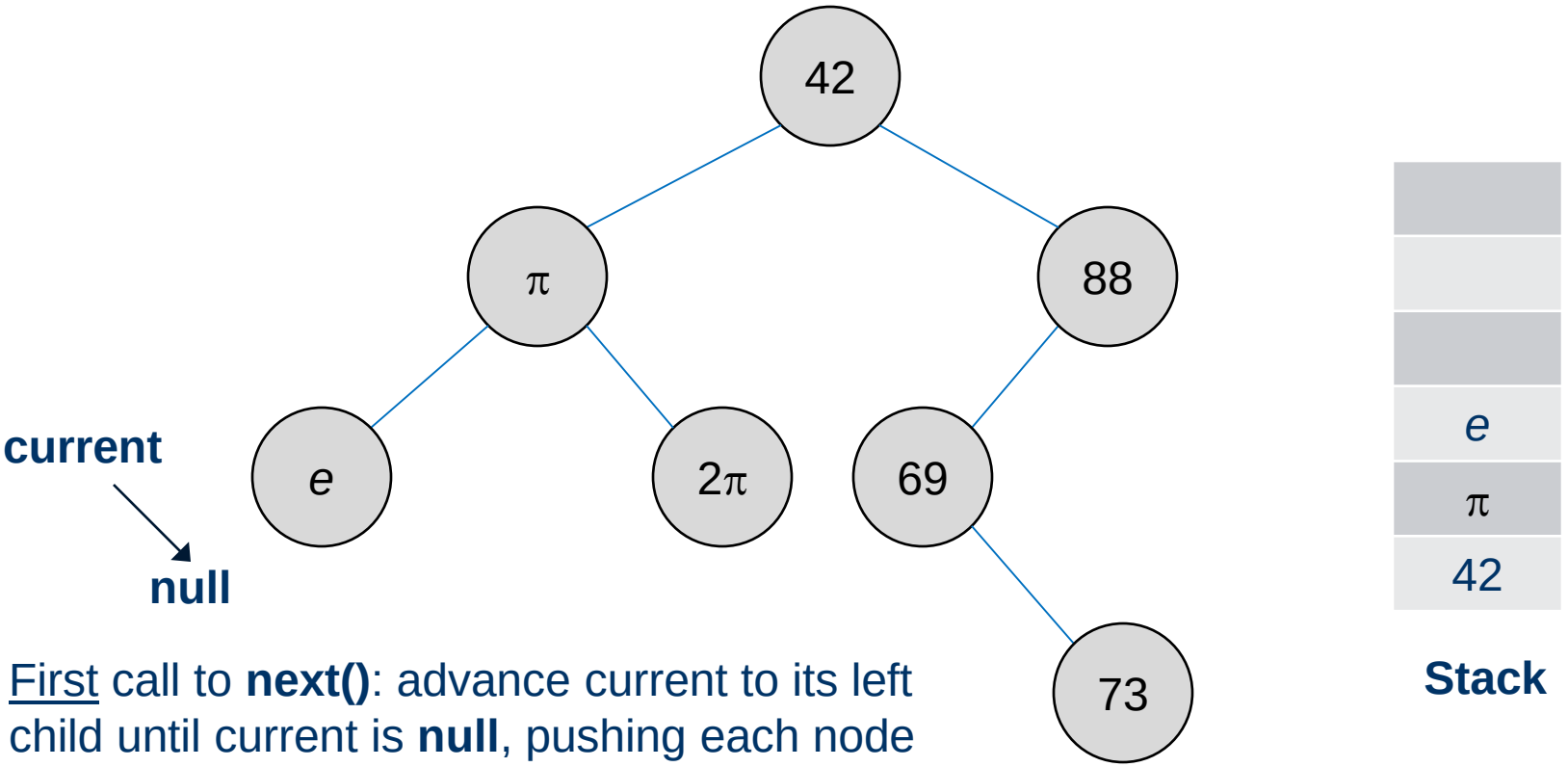
In-order BST iterator: example



Stack
(empty)

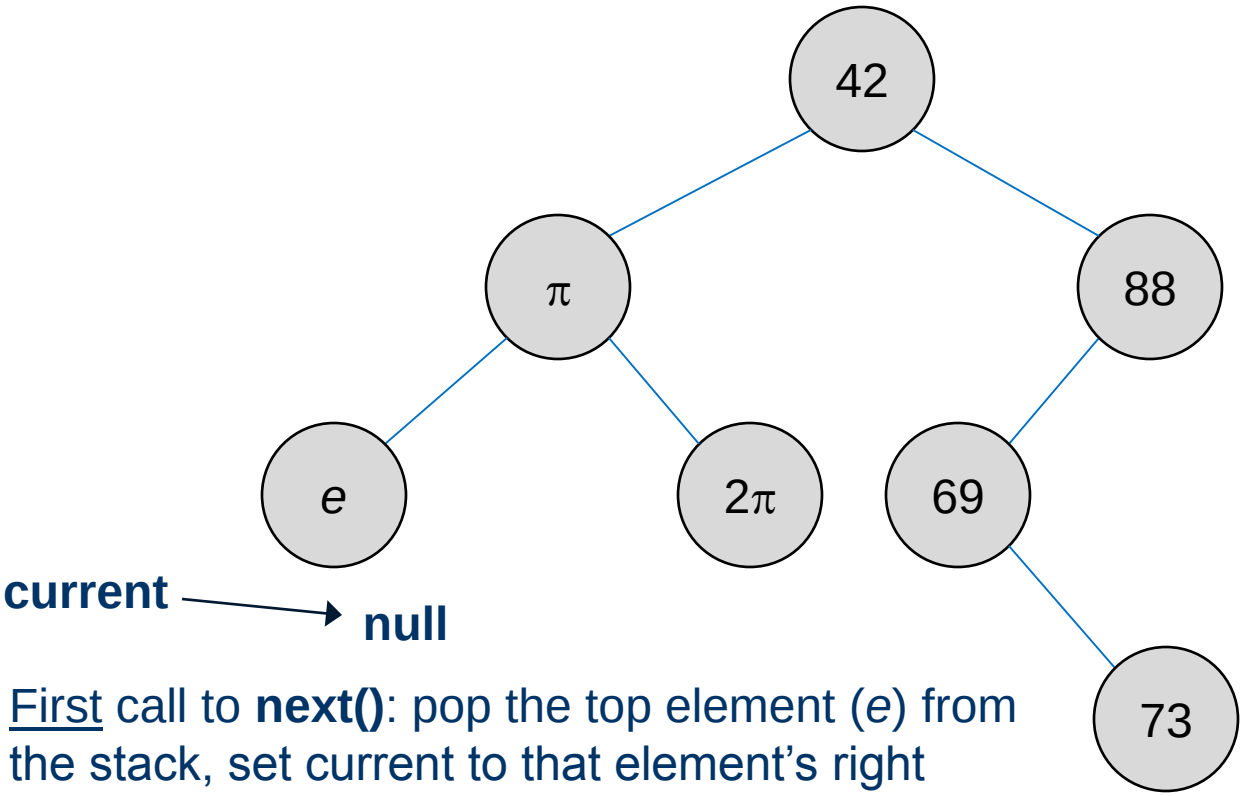
Initial state of the iterator, when it's first created

In-order BST iterator: example

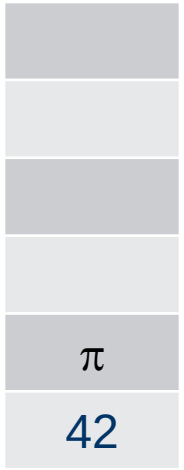


First call to **next()**: advance current to its left child until current is **null**, pushing each node onto the stack as we go

In-order BST iterator: example



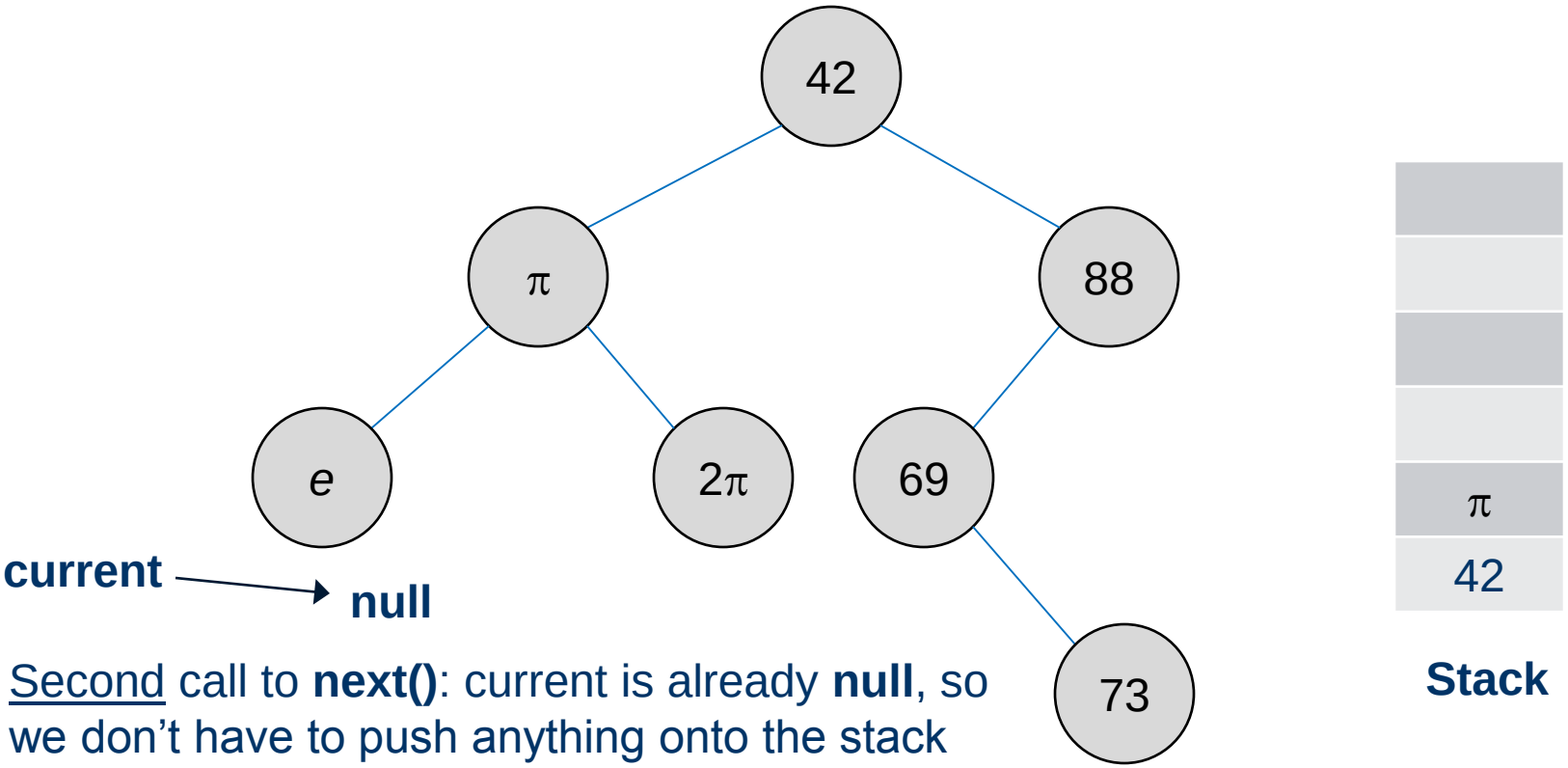
Return: *e*



Stack

First call to **next()**: pop the top element (*e*) from the stack, set current to that element's right child, and return the popped element

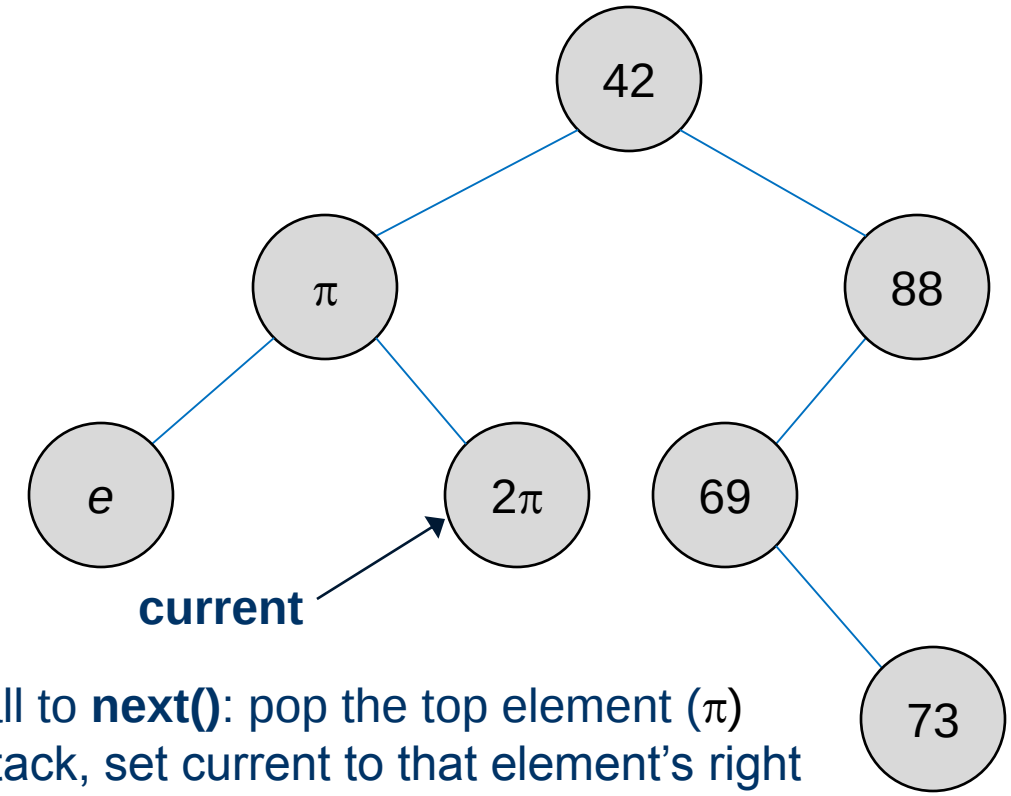
In-order BST iterator: example



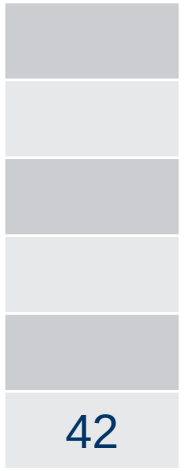
Stack

Second call to **next()**: current is already **null**, so we don't have to push anything onto the stack

In-order BST iterator: example



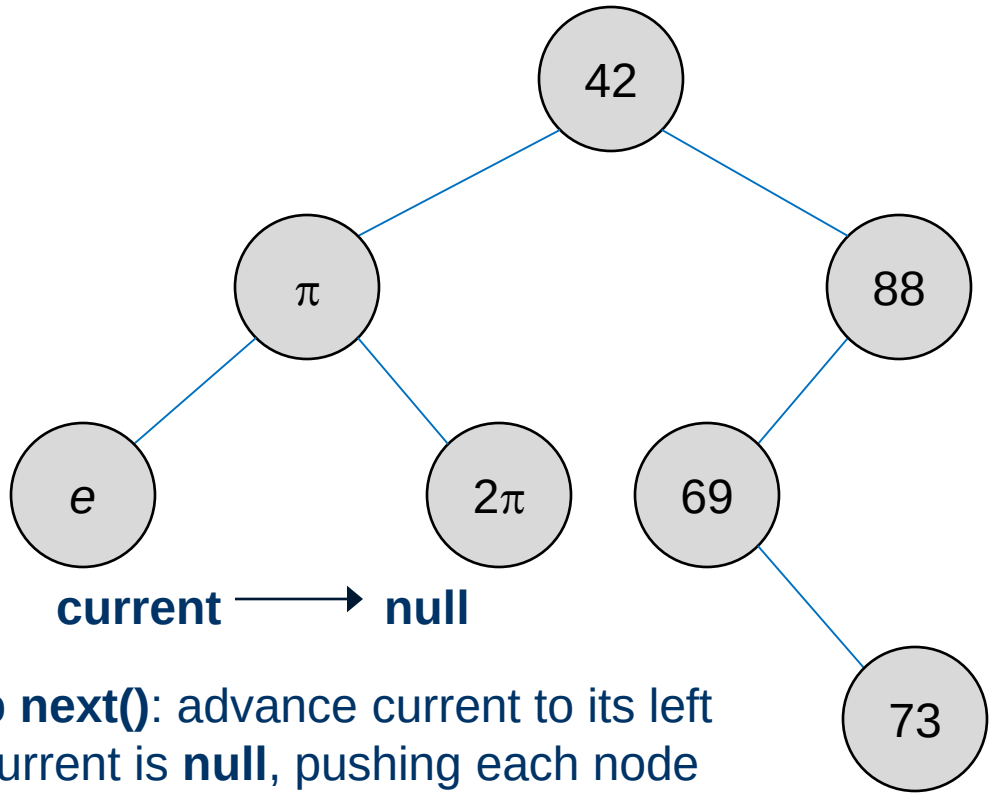
Return: π

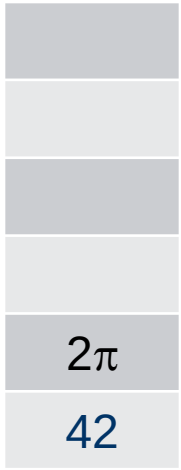


Stack

Second call to **next()**: pop the top element (π) from the stack, set current to that element's right child, and return the popped element

In-order BST iterator: example

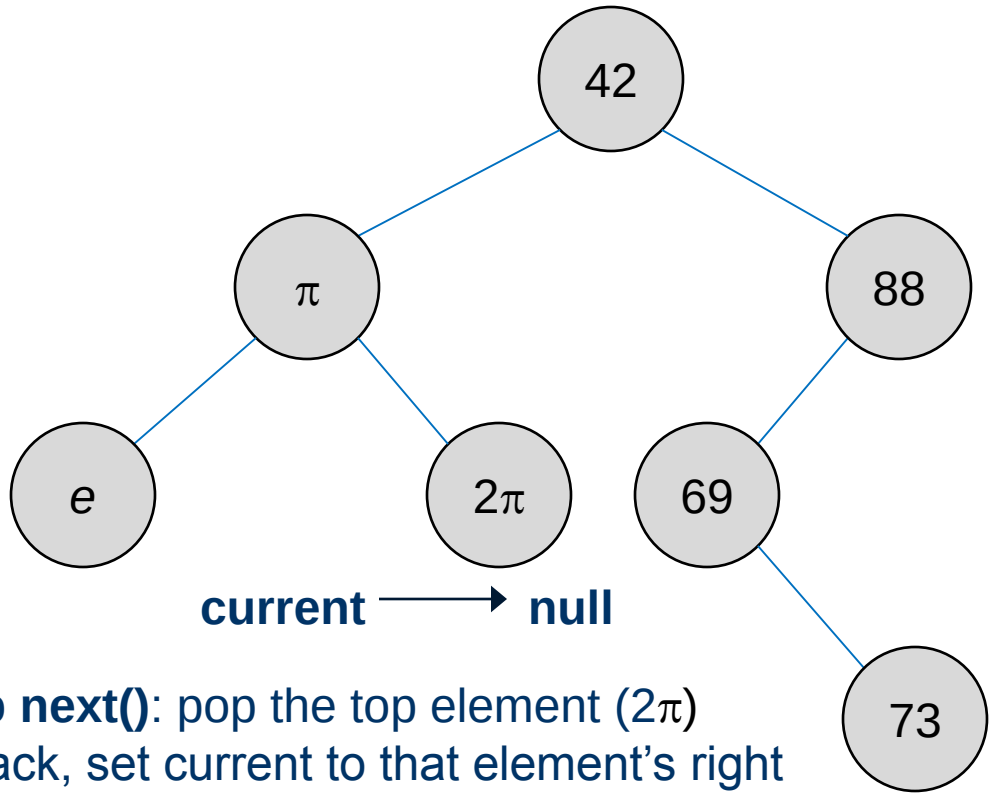




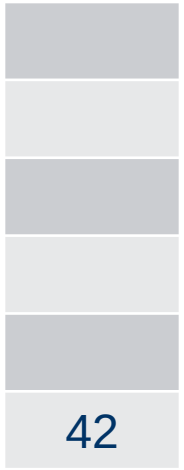
Stack

Third call to **next()**: advance current to its left child until current is **null**, pushing each node onto the stack as we go

In-order BST iterator: example



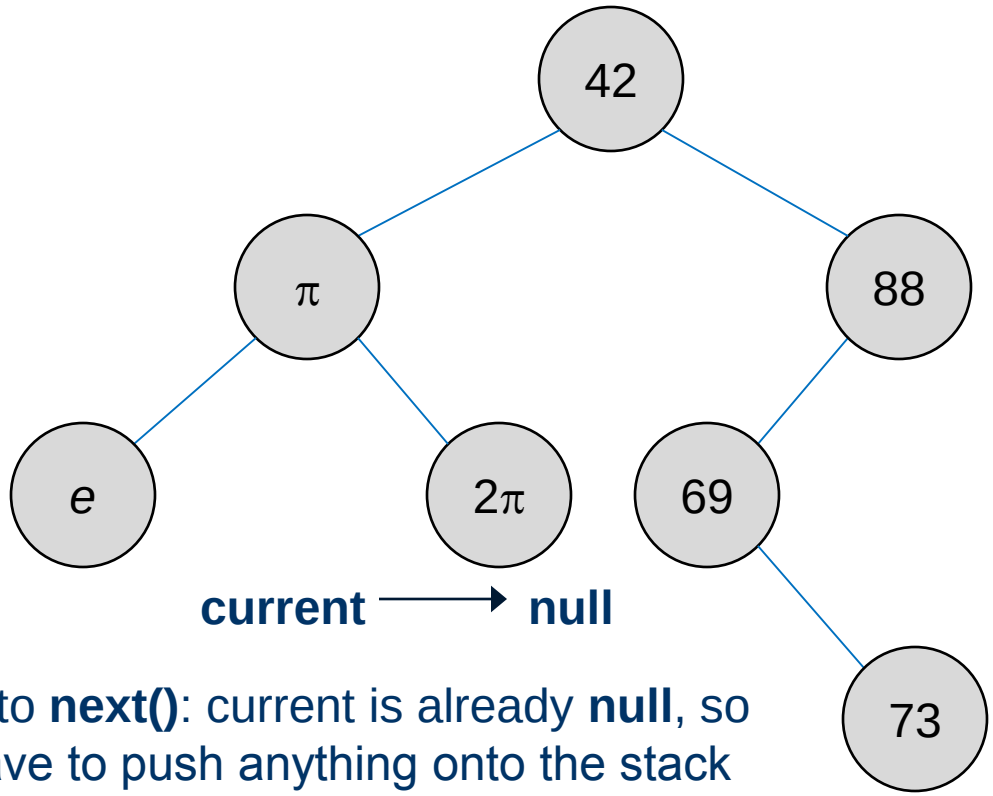
Return: 2π

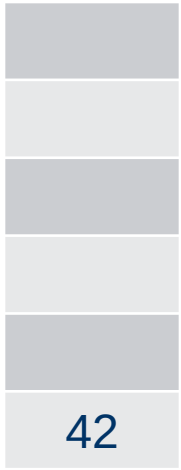


Stack

Third call to **next()**: pop the top element (2π) from the stack, set current to that element's right child, and return the popped element

In-order BST iterator: example

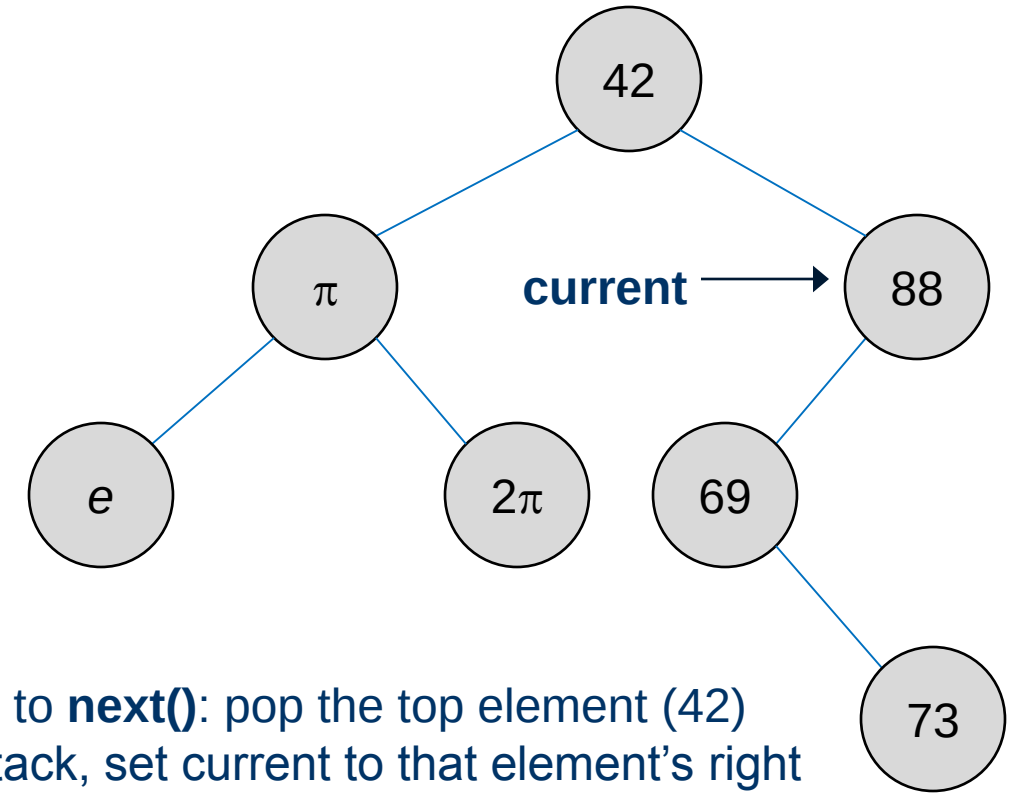




Stack

Fourth call to **next()**: current is already **null**, so we don't have to push anything onto the stack

In-order BST iterator: example

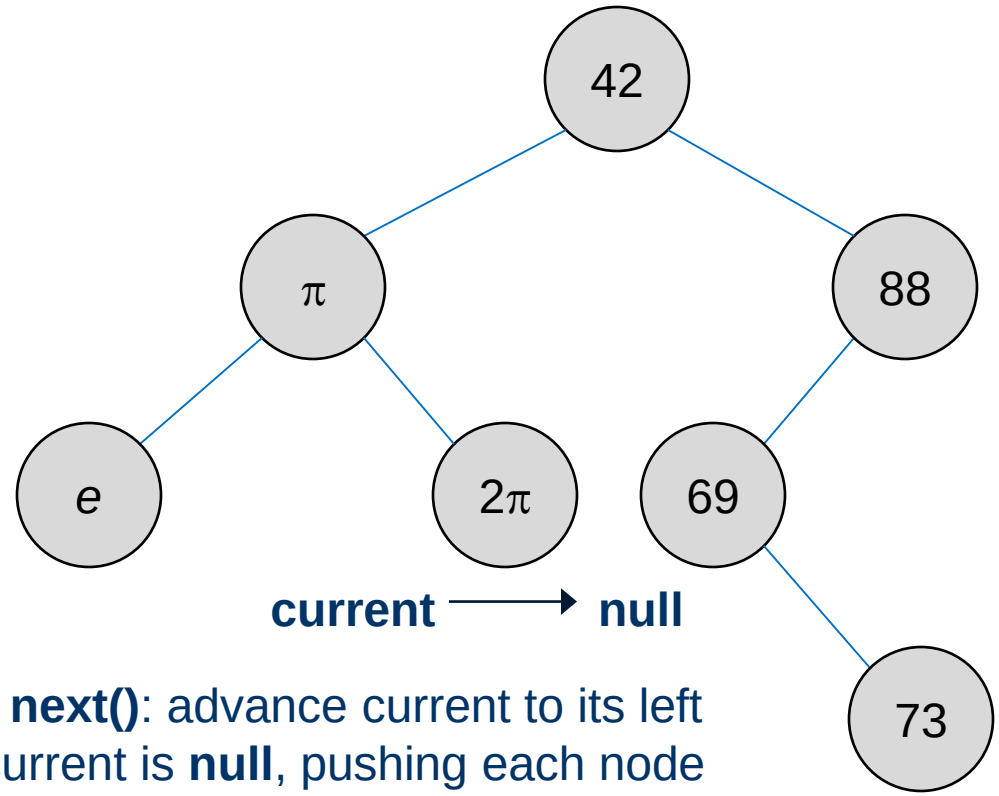


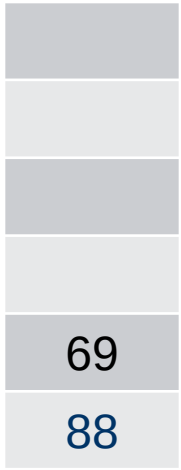
Return: 42

Stack
(empty)

Fourth call to **next()**: pop the top element (42) from the stack, set current to that element's right child, and return the popped element

In-order BST iterator: example

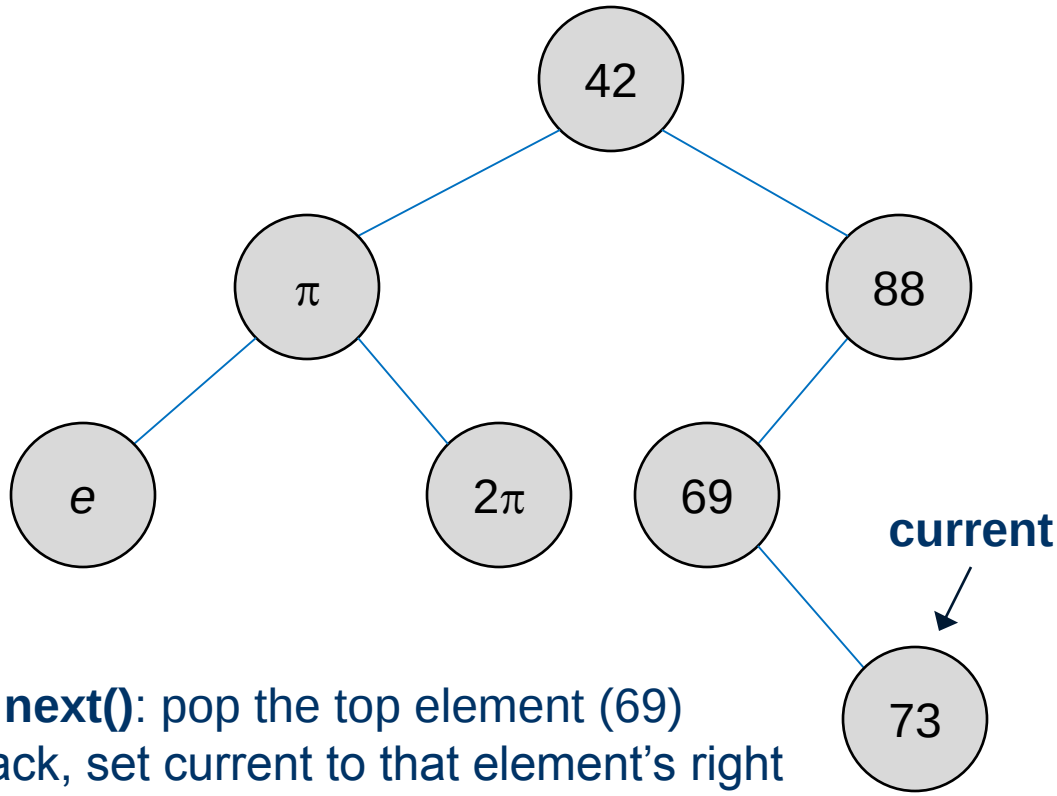




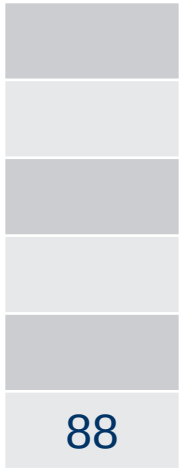
Stack

Fifth call to **next()**: advance current to its left child until current is **null**, pushing each node onto the stack as we go

In-order BST iterator: example



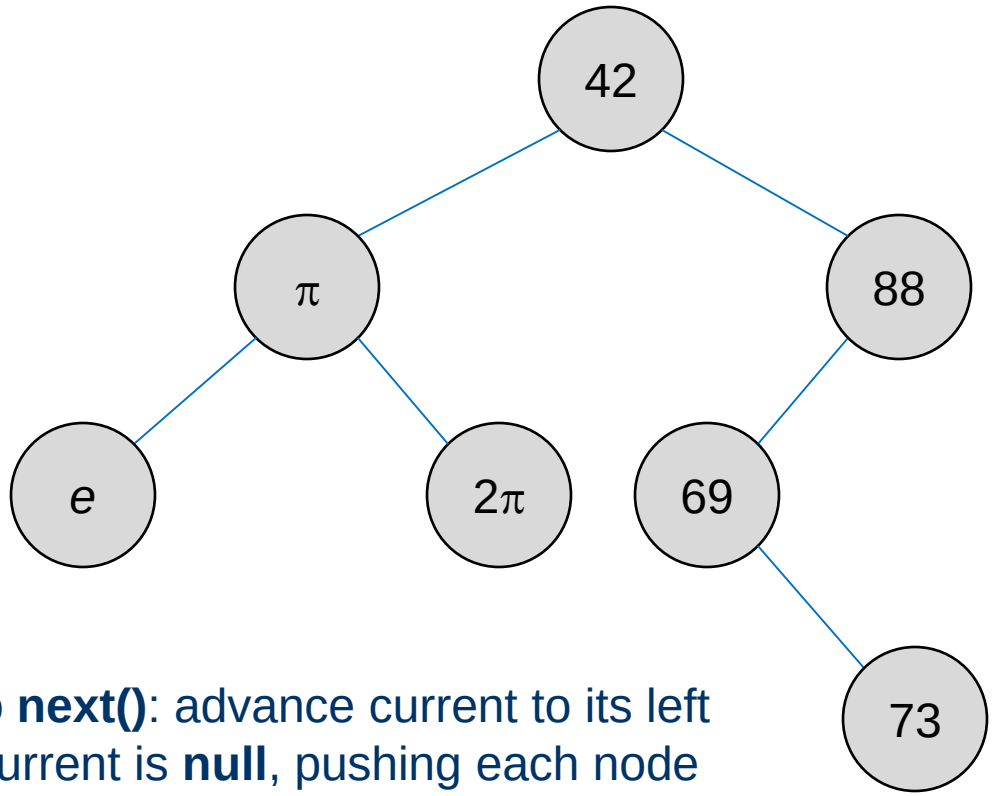
Return: 69

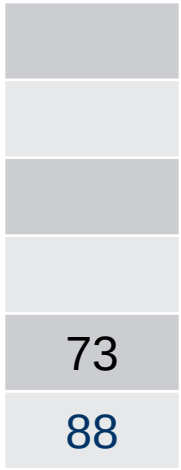


Stack

Fifth call to **next()**: pop the top element (69) from the stack, set current to that element's right child, and return the popped element

In-order BST iterator: example



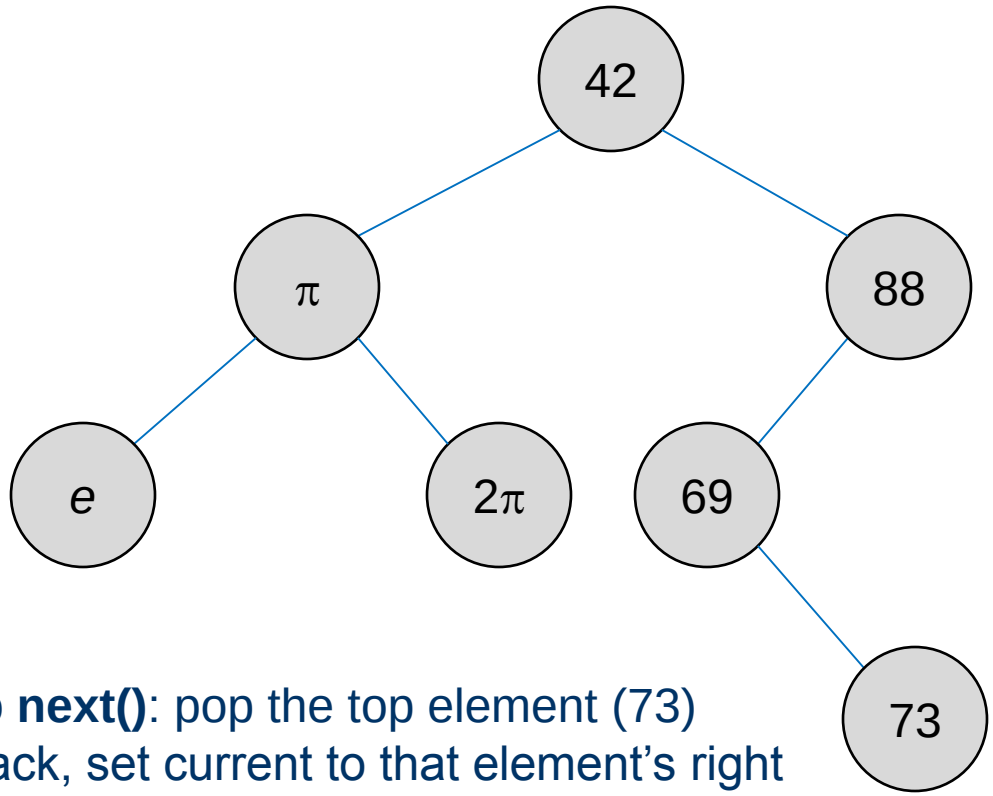


Stack

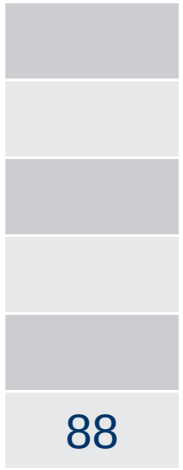
Sixth call to **next()**: advance current to its left child until current is **null**, pushing each node onto the stack as we go

current → **null**

In-order BST iterator: example



Return: 73

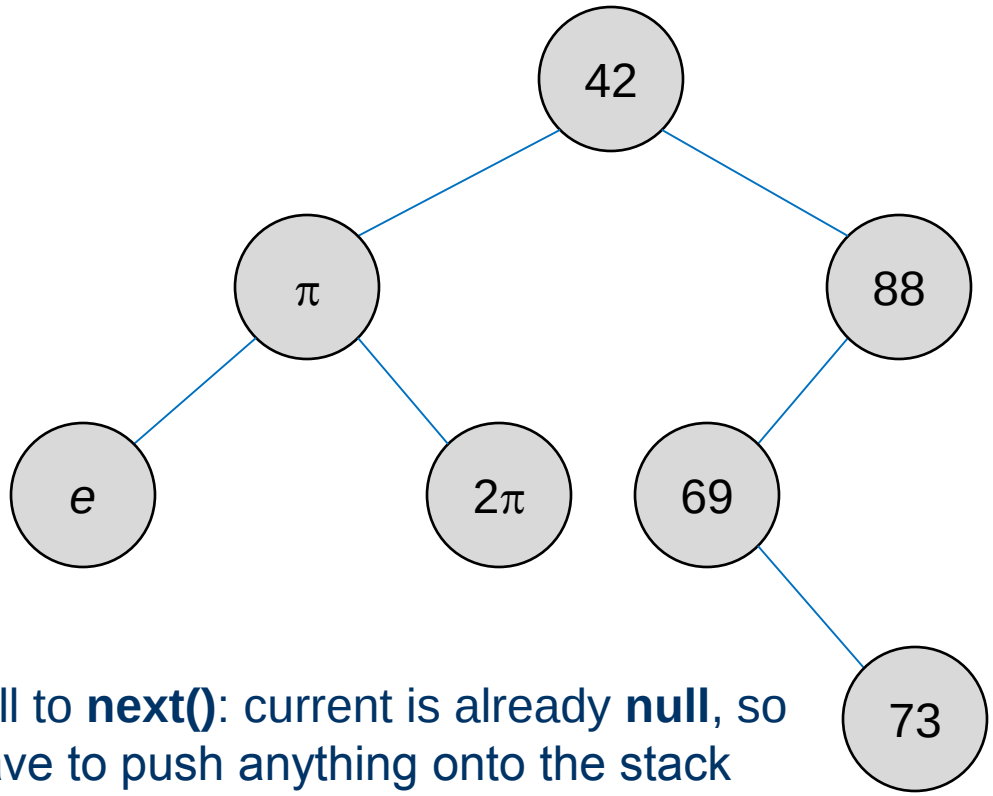


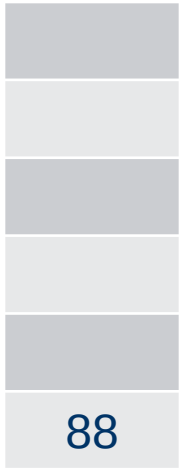
Stack

Sixth call to **next()**: pop the top element (73) from the stack, set current to that element's right child, and return the popped element

current → **null**

In-order BST iterator: example



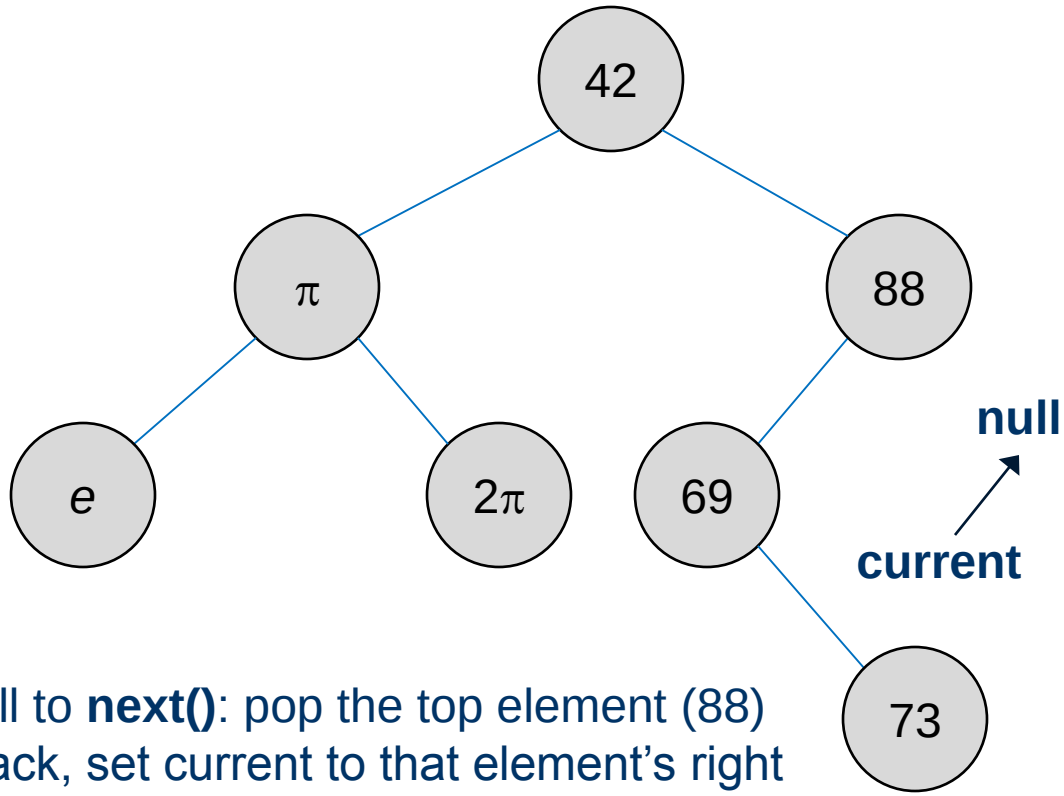


Stack

Seventh call to **next()**: current is already **null**, so we don't have to push anything onto the stack

current → **null**

In-order BST iterator: example

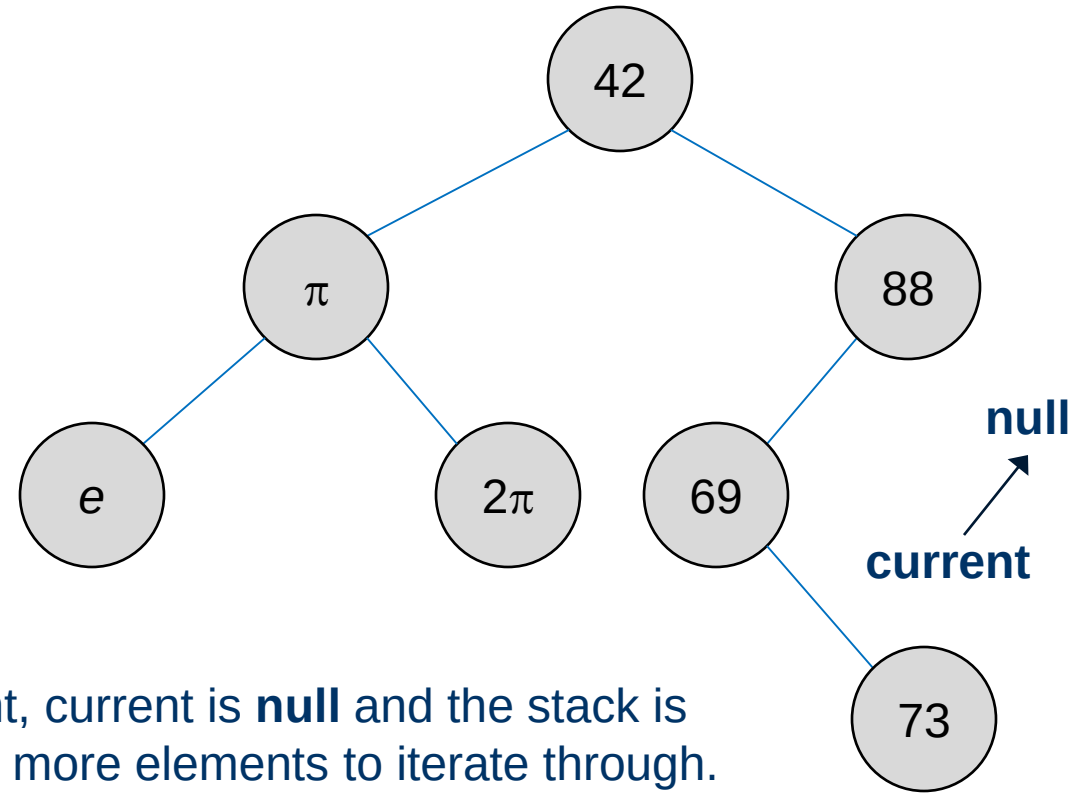


Return: 88

Stack
(empty)

Seventh call to **next()**: pop the top element (88) from the stack, set current to that element's right child, and return the popped element

In-order BST iterator: example



Stack
(empty)

At this point, current is **null** and the stack is empty! No more elements to iterate through.