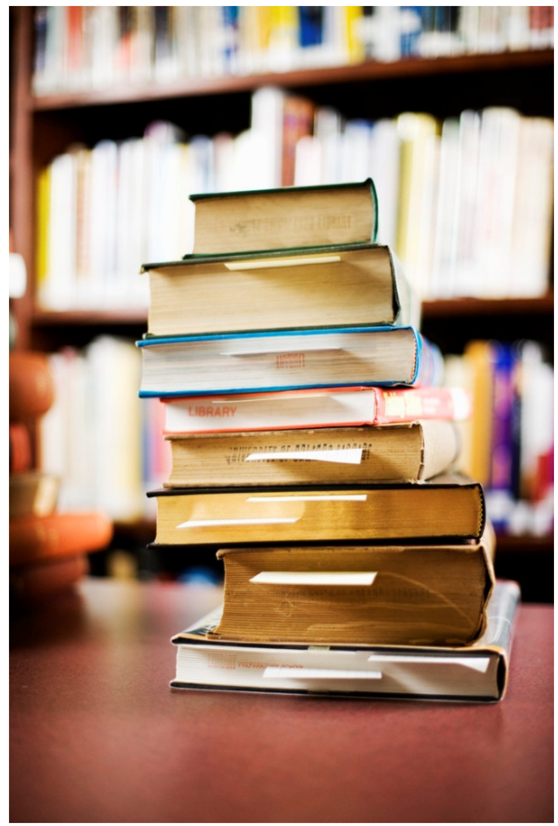
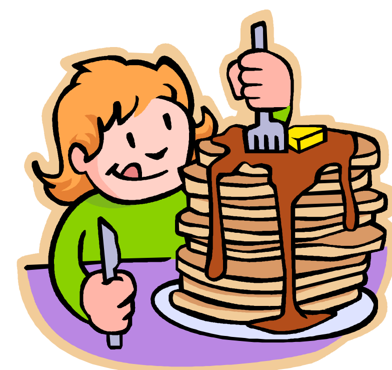
More Data Structures (Part 1)

Stacks

Stack

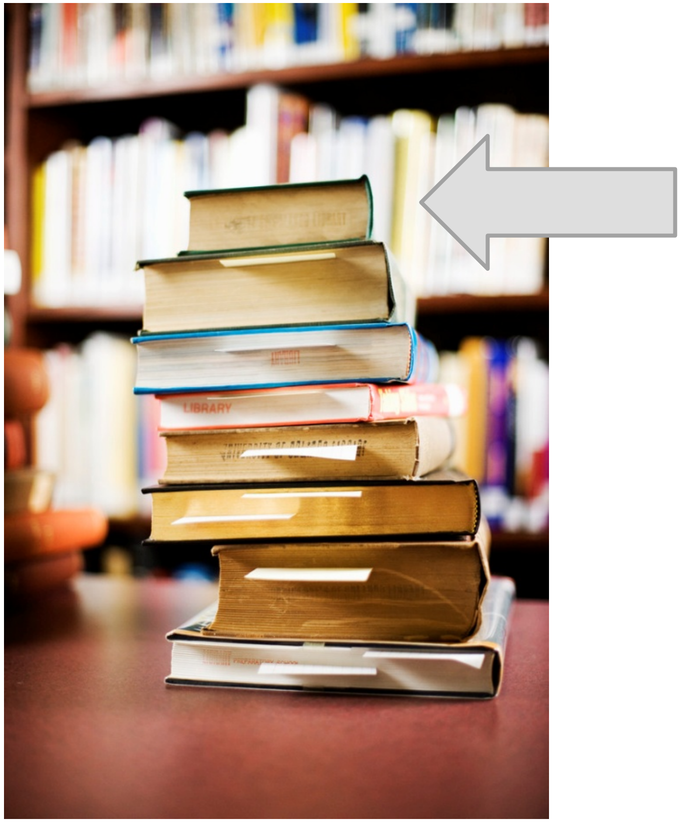
- examples of stacks





Top of Stack

- top of the stack



Stack Operations

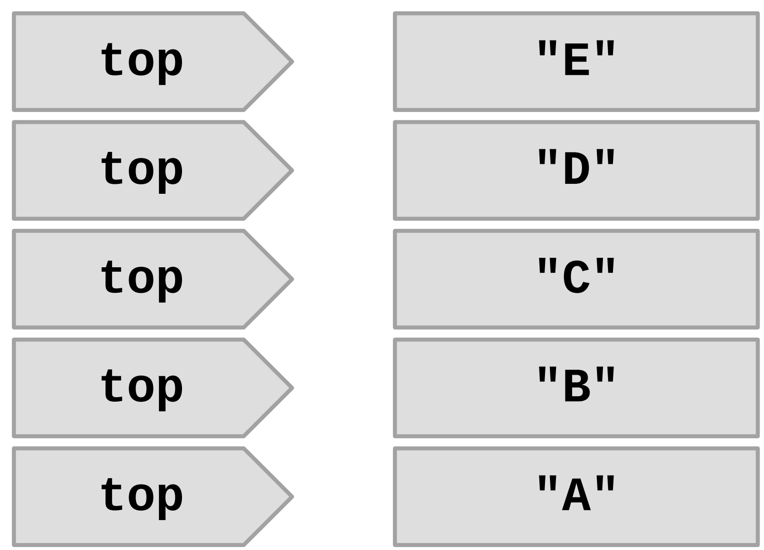
- classically, stacks only support two operations
  1. push
     - add to the top of the stack
  2. pop
     - remove from the top of the stack

Stack Optional Operations

- optional operations
  1. size
     - number of elements in the stack
  2. isEmpty
     - is the stack empty?
  3. peek
     - get the top element (without removing it)
  4. search
     - find the position of the element in the stack
  5. isFull
     - is the stack full? (for stacks with finite capacity)
  6. capacity
     - total number of elements the stack can hold (for stacks with finite capacity)

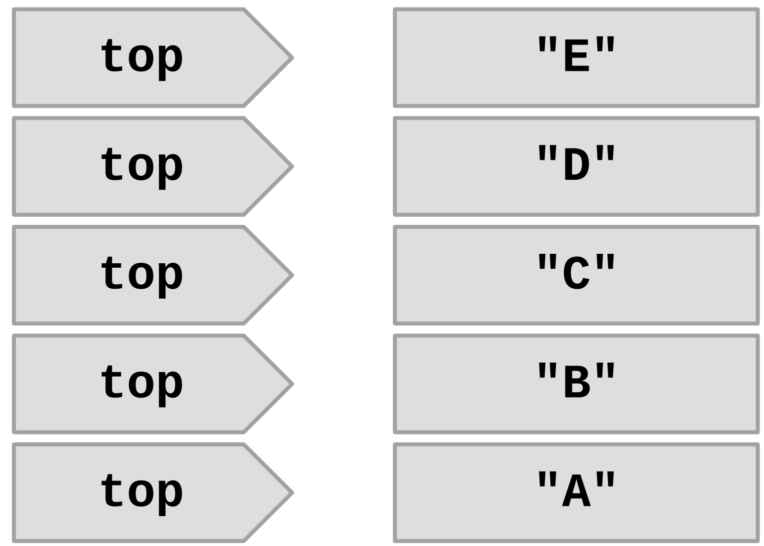
Push

1. `st.push("A")`
2. `st.push("B")`
3. `st.push("C")`
4. `st.push("D")`
5. `st.push("E")`



Pop

1. `String s = st.pop()`
2. `s = st.pop()`
3. `s = st.pop()`
4. `s = st.pop()`
5. `s = st.pop()`



LIFO

- stack is a Last-In-First-Out (LIFO) data structure
  - the last element pushed onto the stack is the first element that can be accessed from the stack

Implementation with LinkedList

- a linked list can be used to efficiently implement a stack
- the head of the list becomes the top of the stack
  - adding (push) and removing (pop) from the head of a linked list requires O(1) time

```
public class Stack<E> {
  private LinkedList<E> stack;

  public Stack() {
    this.stack = new LinkedList<E>();
  }

  public push(E element) {
    this.stack.addFirst(element);
  }

  public E pop() {
    return this.stack.removeFirst();
  }
}
```

Implementation with ArrayList

- **`ArrayList`** can be used to efficiently implement a stack
- the end of the list becomes the top of the stack
  - adding and removing to the end of an **`ArrayList`** usually can be performed in O(1) time

```java
public class Stack<E> {
  private ArrayList<E> stack;

  public Stack() {
    this.stack = new ArrayList<E>();
  }

  public push(E element) {
    this.stack.add(element);
  }

  public E pop() {
    return this.stack.remove(this.stack.size() - 1);
  }
}
```

Implementation with ArrayDeque

- a deque is a double ended queue
  - a linear collection that supports element insertion and removal from both ends
- an **`ArrayDeque`** can be used to efficiently implement a stack
- the head of the deque becomes the top of the stack
  - adding (push) and removing (pop) from the head of a deque requires O(1) time

```java
public class Stack<E> {
  private ArrayDeque<E> stack;

  public Stack() {
    this.stack = new ArrayDeque<E>();
  }

  public push(E element) {
    this.stack.addFirst(element);
  }

  public E pop() {
    return this.stack.removeFirst();
  }
}
```

Implementations in java.util

- java.util.Stack provides a stack class
- could also use any class that implements java.util.Deque directly
  - java.util.ArrayDeque
  - java.util.LinkedList

Applications

- stacks are used widely in computer science and computer engineering
  - a call stack is used to store information about the active methods in a Java program
  - undo/redo
  - widely used in parsing

Example: Reversing a sequence

- a silly and usually inefficient way to reverse a sequence is to use a stack

Don't do this

```
public static <E> List<E> reverse(List<E> t) {
  List<E> result = new ArrayList<E>();
  Stack<E> st = new Stack<E>();
  for (E e : t) {
    st.push(e);
  }
  while (!st.isEmpty()) {
    result.add(st.pop());
  }
  return result;
}
```

Example: eCheck11B

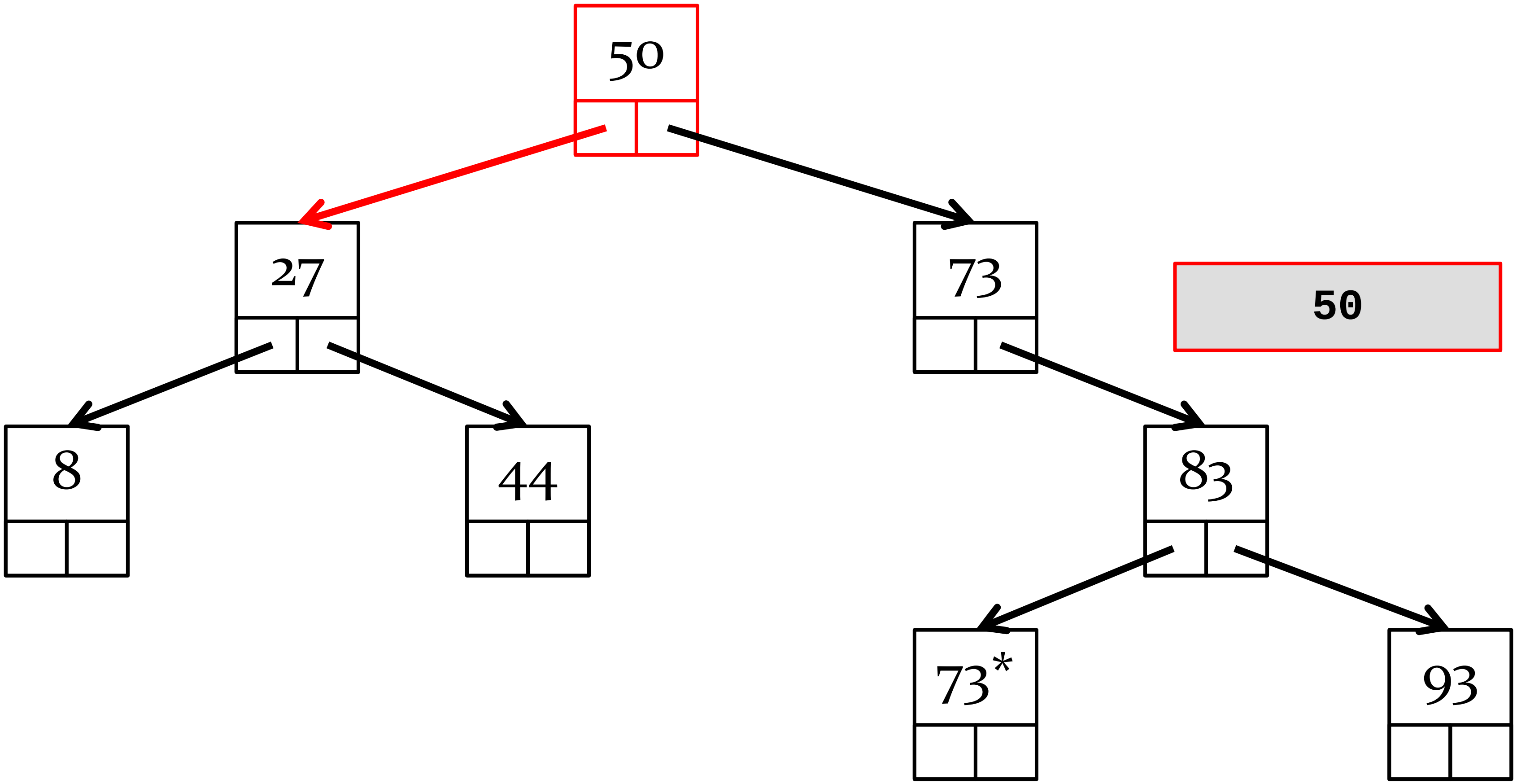
- see http://www.cse.yorku.ca/course_archive/2010-11/F/1020/sectionE/day35.html#%282%29

Example: Tree traversal

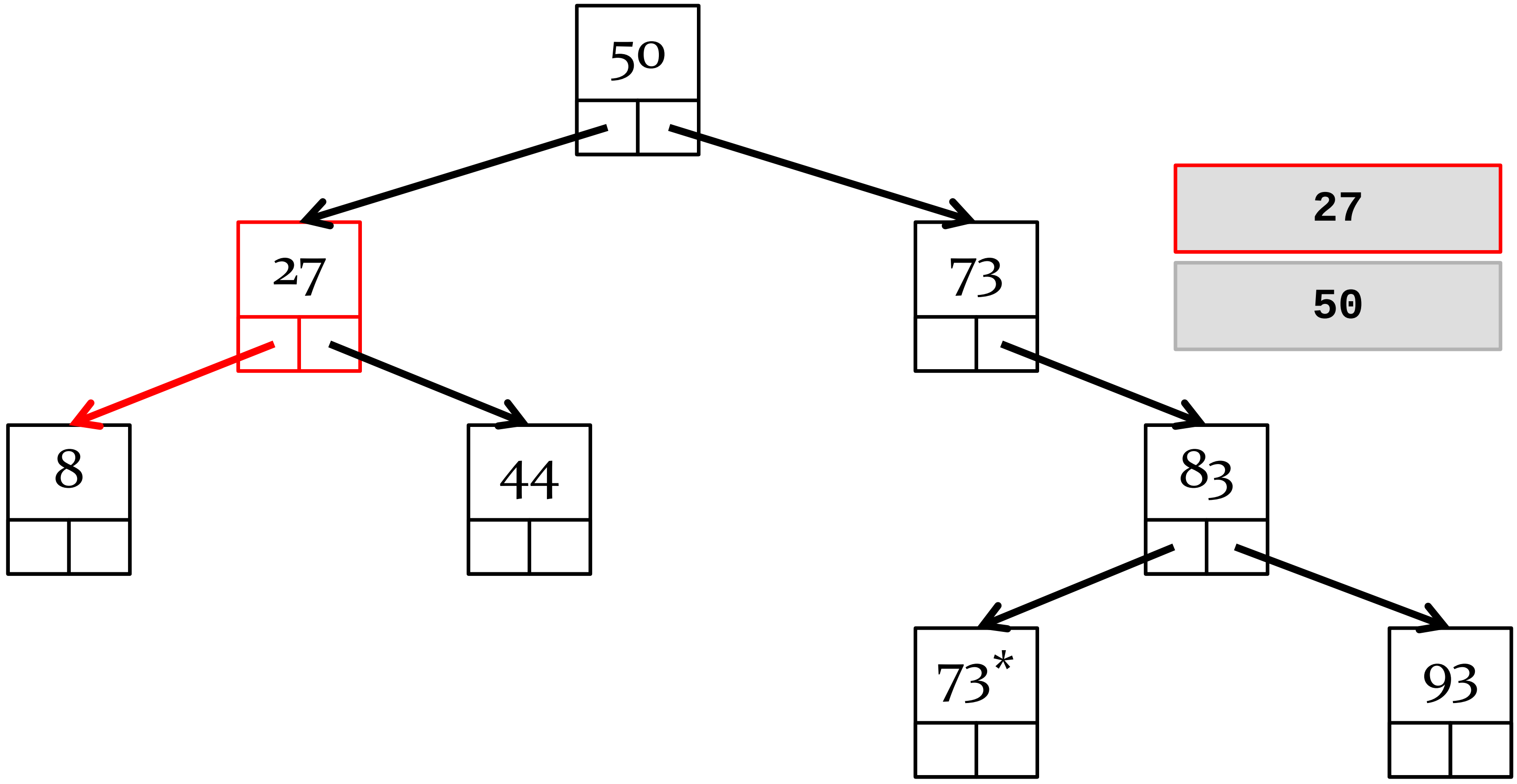
- a stack can be used in place of recursion for visiting all of the nodes of a tree
  - basic idea is to push nodes onto the stack as you traverse the tree
  - pushing the node onto the stack allows you to remember that you have to visit the other branch of the tree rooted at the node

Recursive inorder traversal

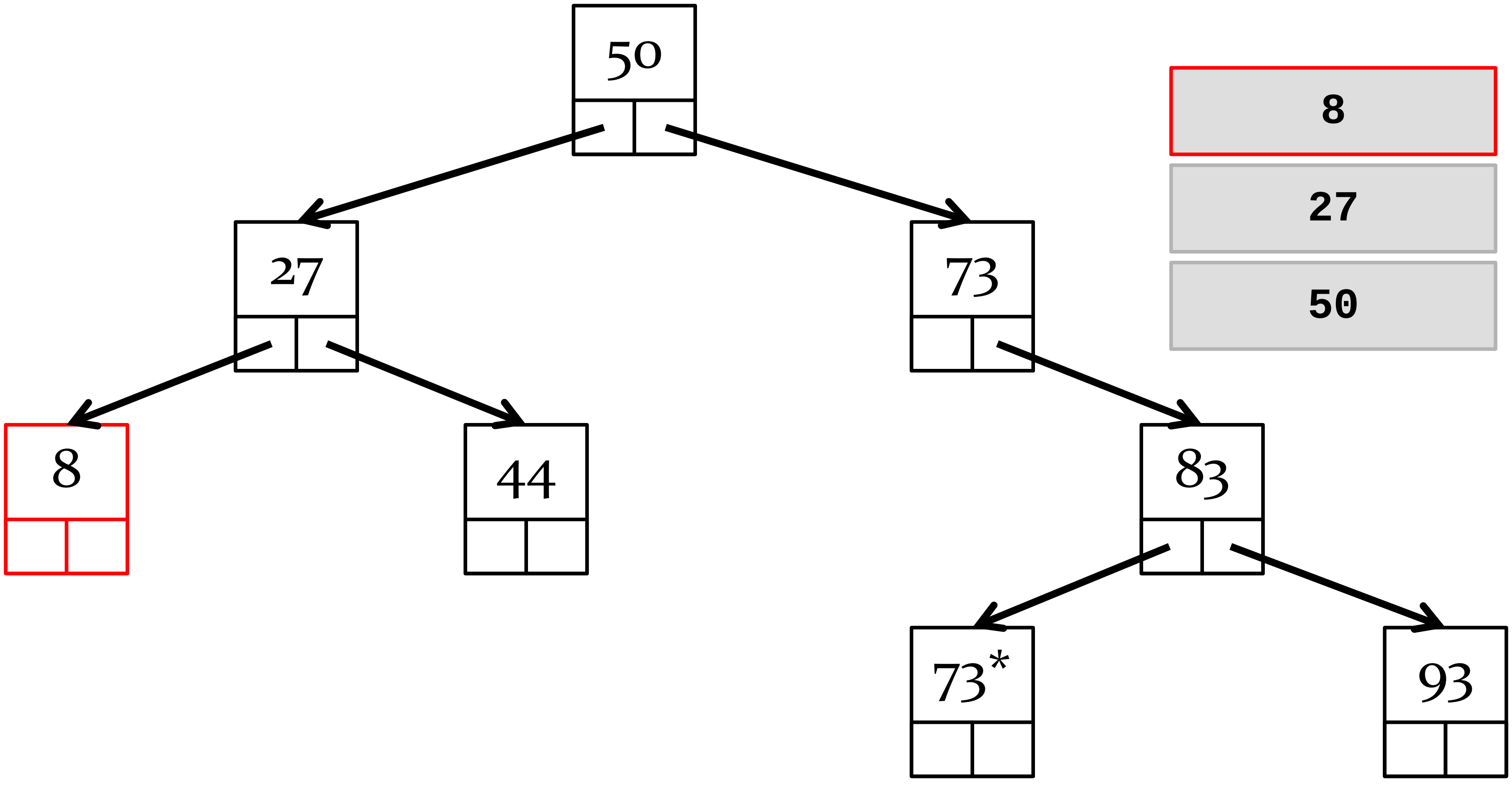
```
public String toString() {
  return "{" + toString(this.root) + "}";
}

private static <E extends Comparable<? super E>>
  String toString(Node<E> subtreeRoot) {
  if (subtreeRoot == null) {
    return "";
  }
  String left = toString(subtreeRoot.left);
  String right = toString(subtreeRoot.right);
  return left + subtreeRoot.data + right;
}
```



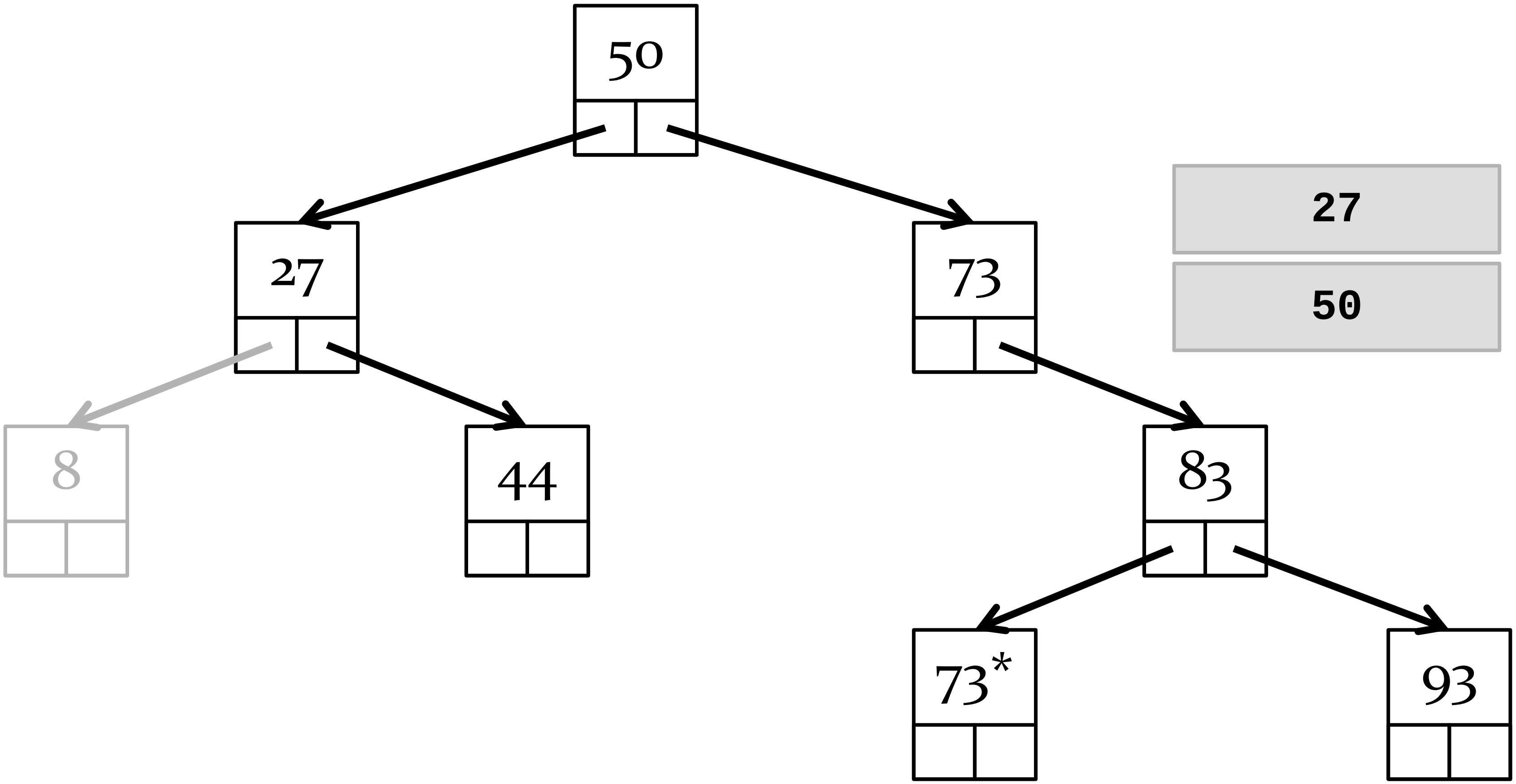
inorder: 8, 27, 44, 50, 73, 73*, 83, 93



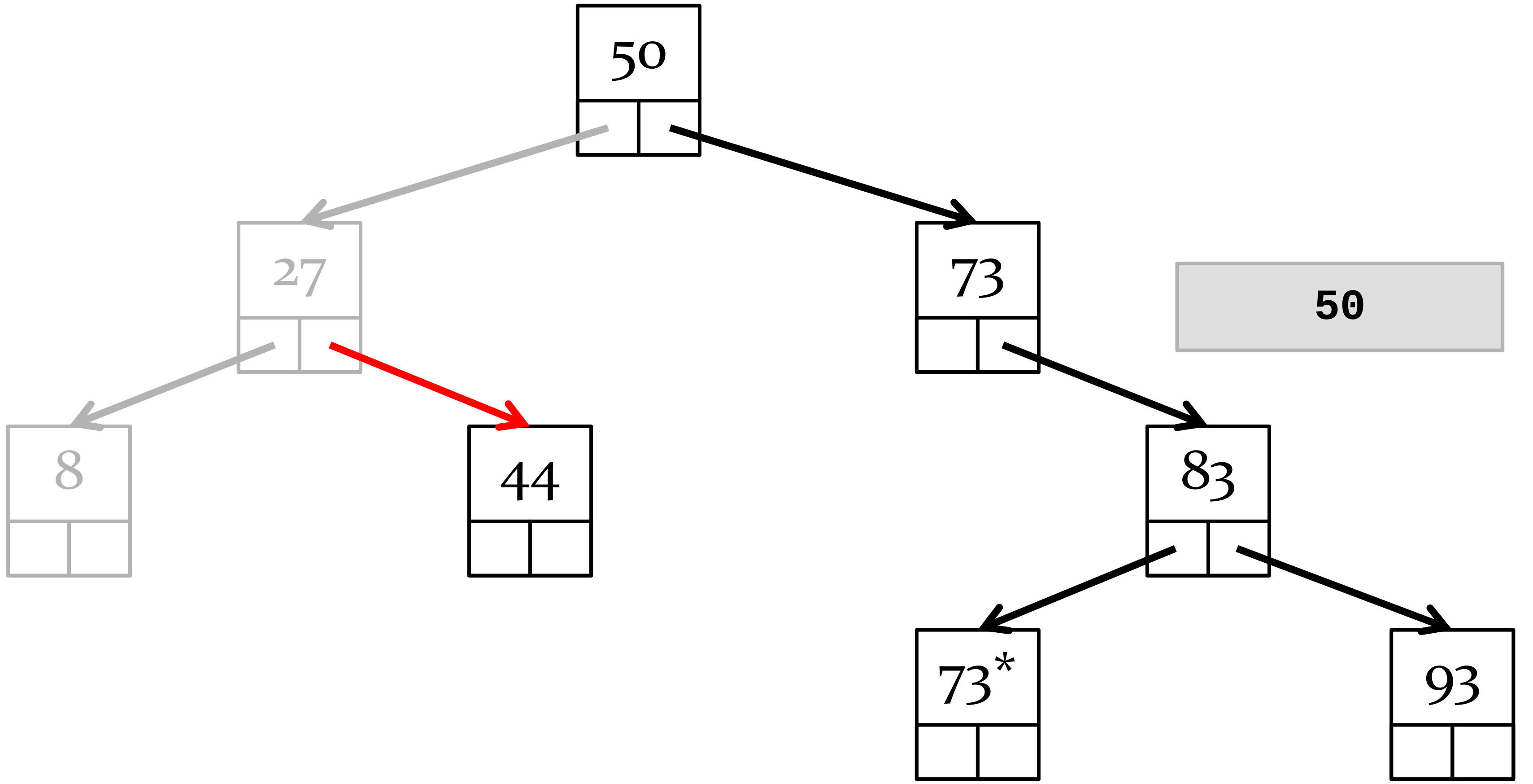
inorder: 8, 27, 44, 50, 73, 73*, 83, 93



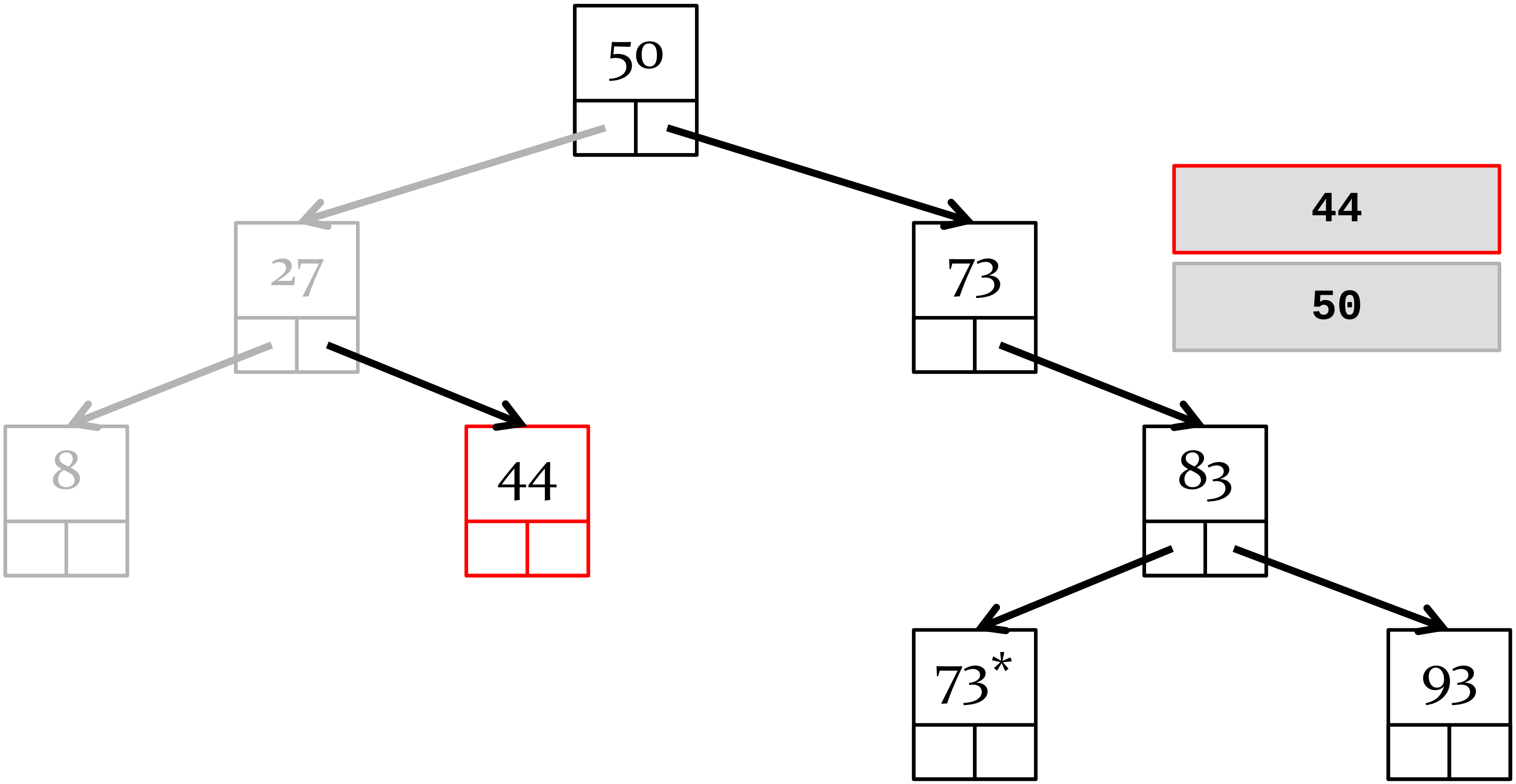
inorder: 8, 27, 44, 50, 73, 73*, 83, 93



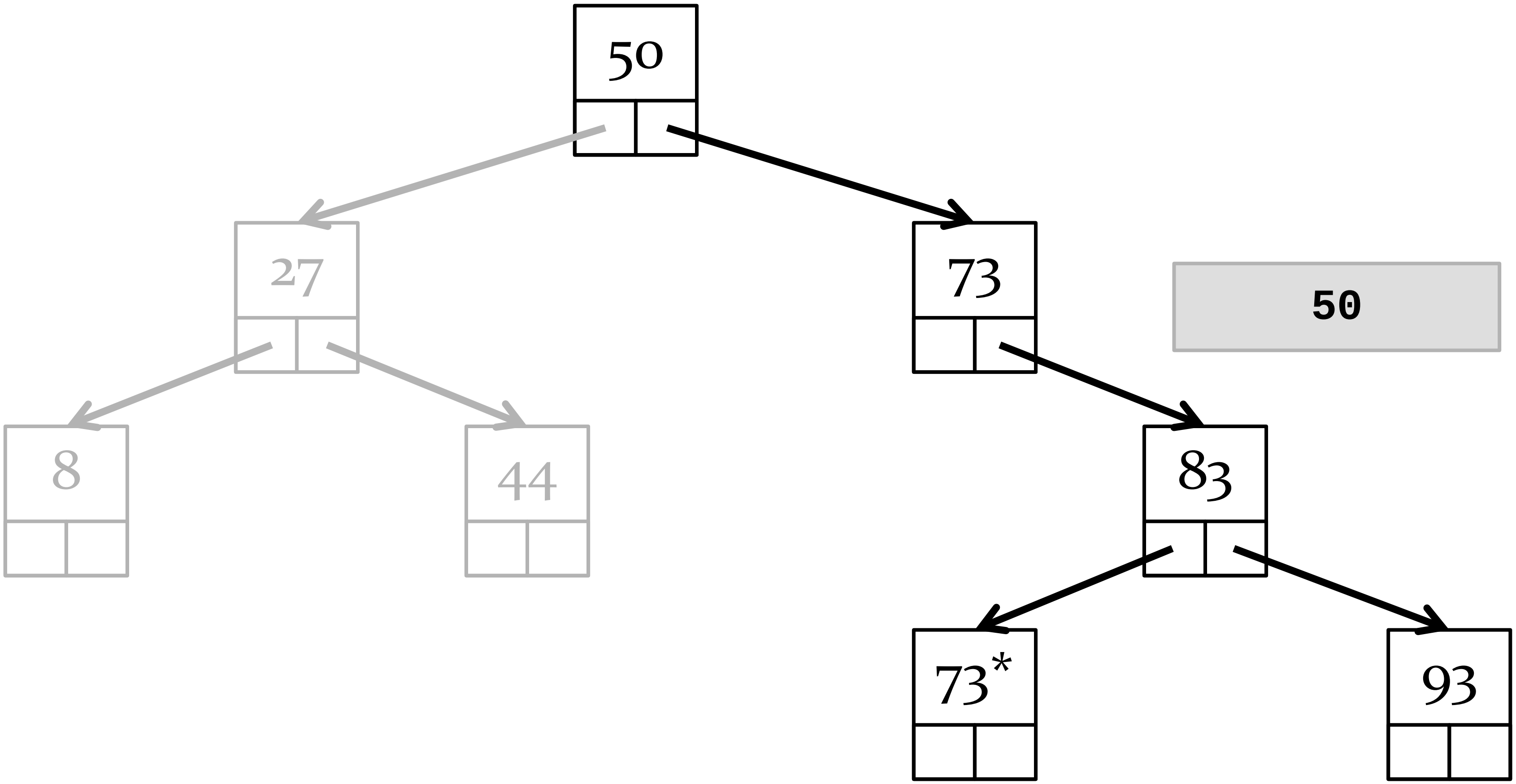
inorder: 8, 27, 44, 50, 73, 73*, 83, 93



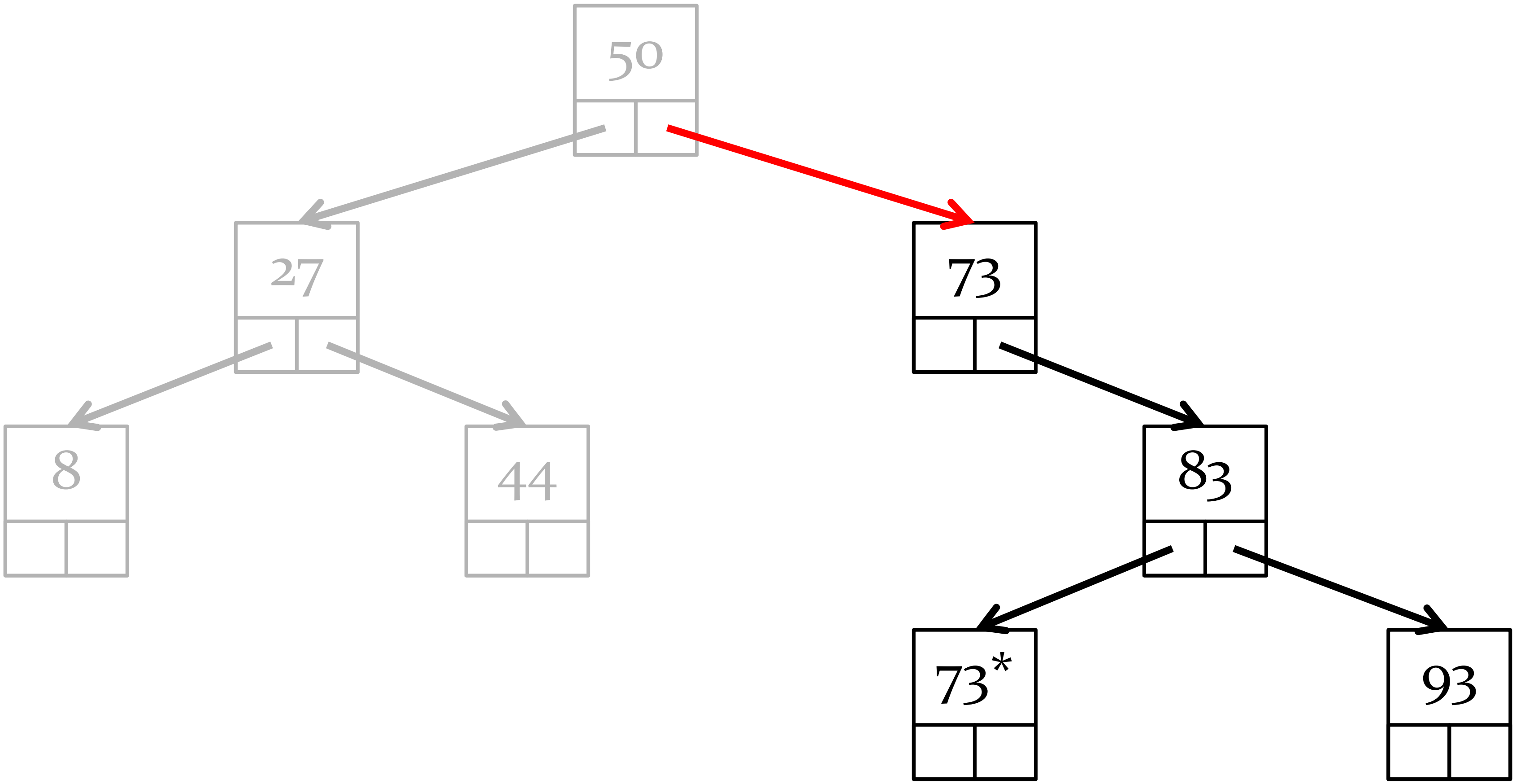
inorder: 8, 27, 44, 50, 73, 73*, 83, 93



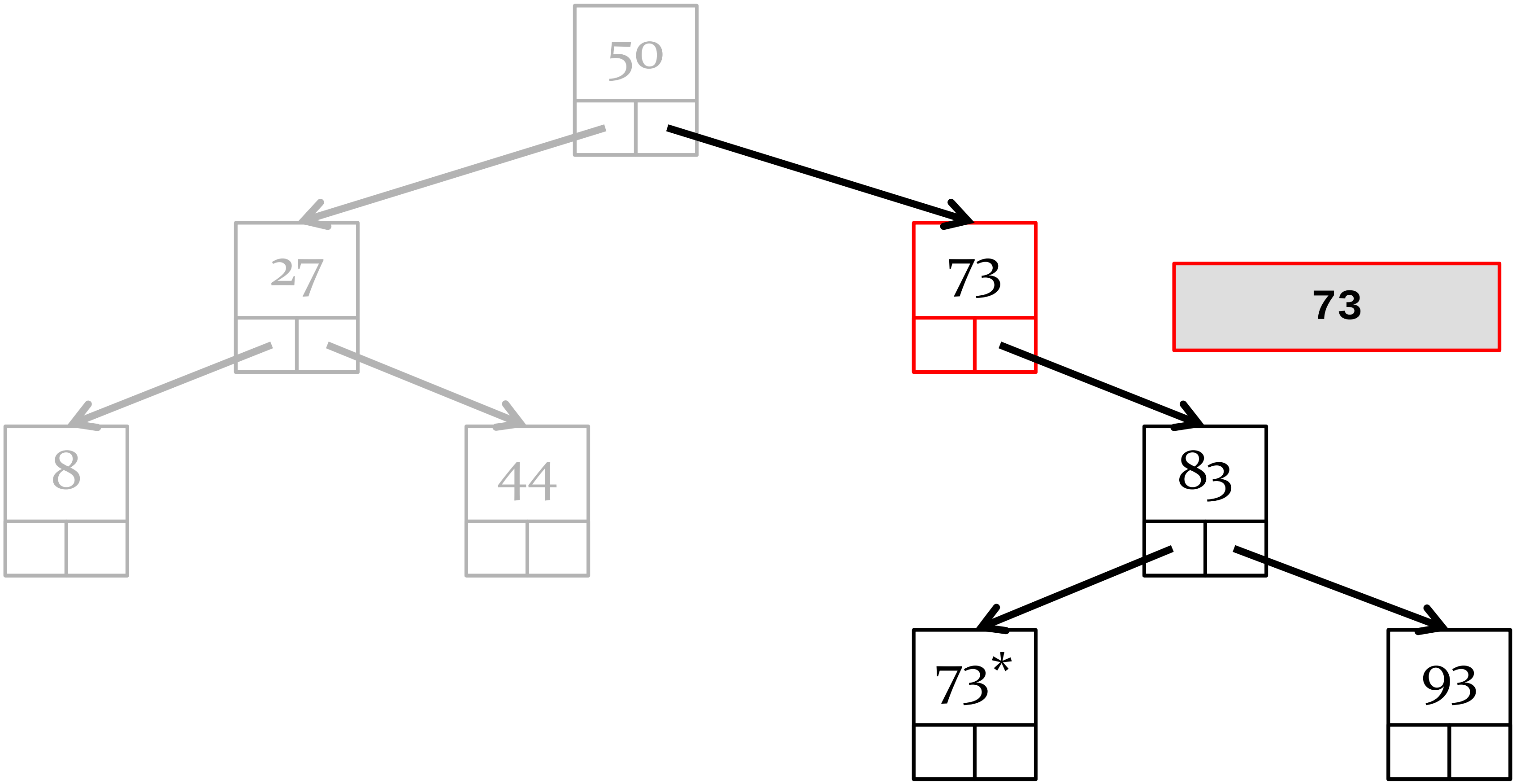
inorder: 8, 27, 44, 50, 73, 73*, 83, 93



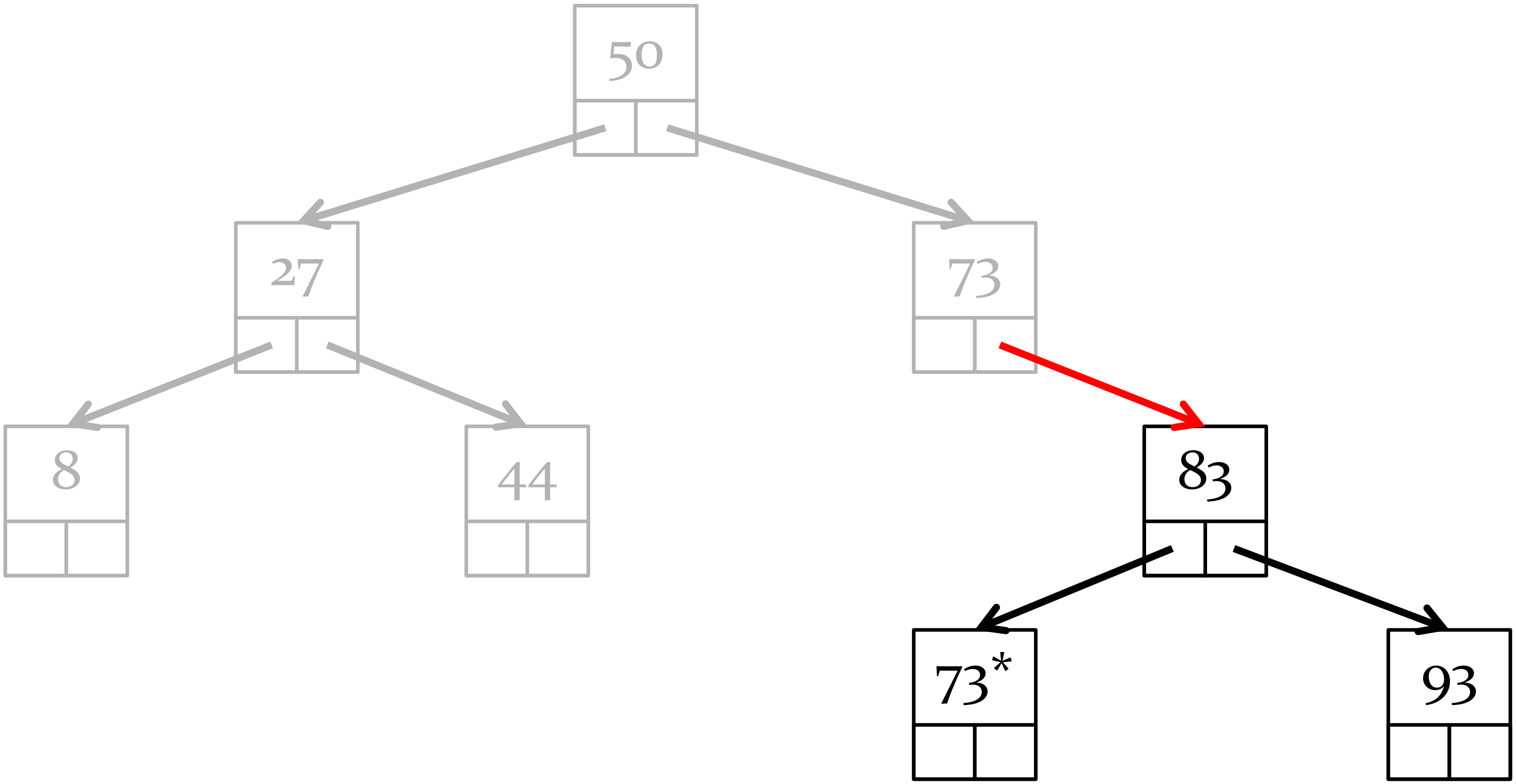
inorder: 8, 27, 44, 50, 73, 73*, 83, 93



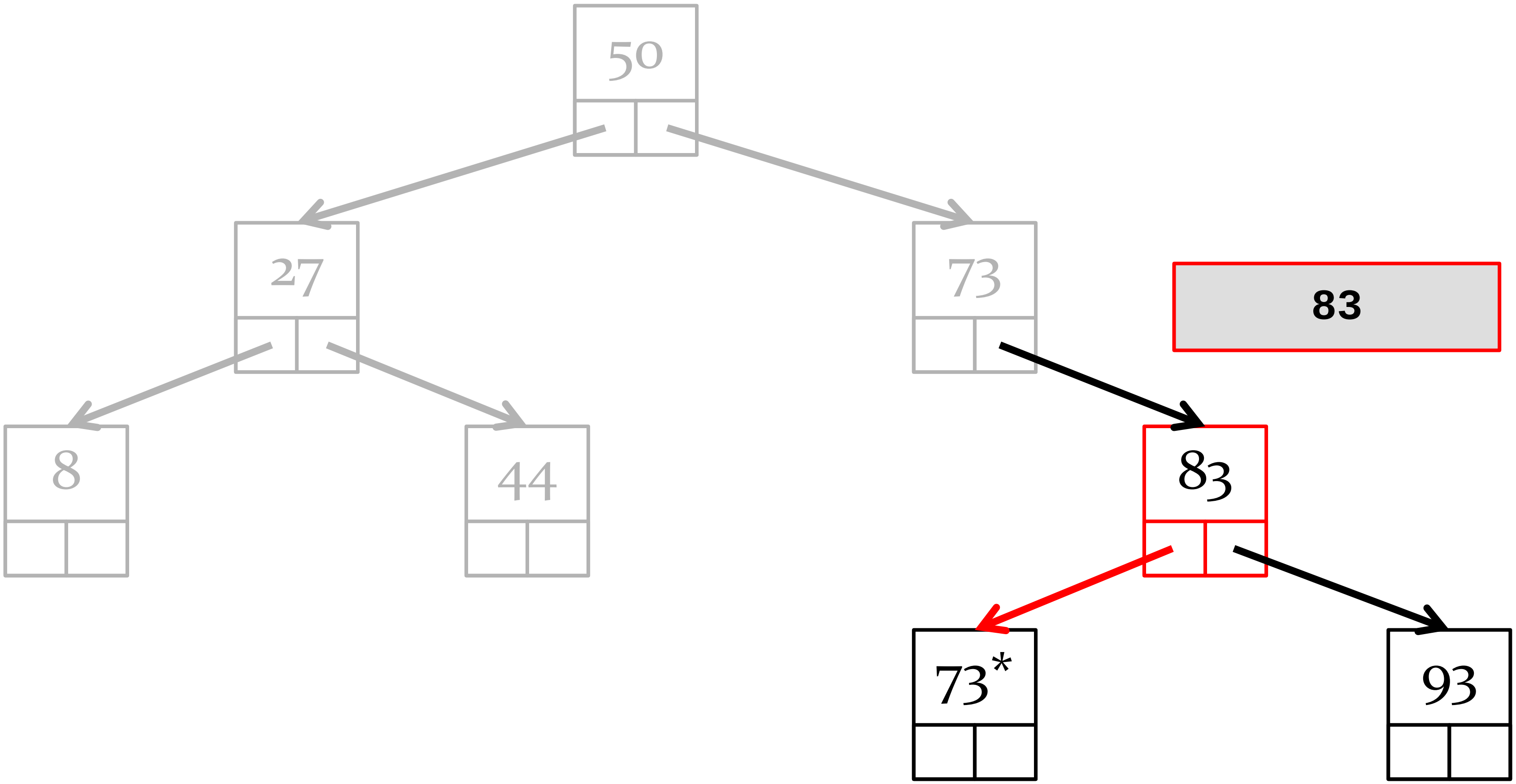
inorder: 8, 27, 44, 50, 73, 73*, 83, 93



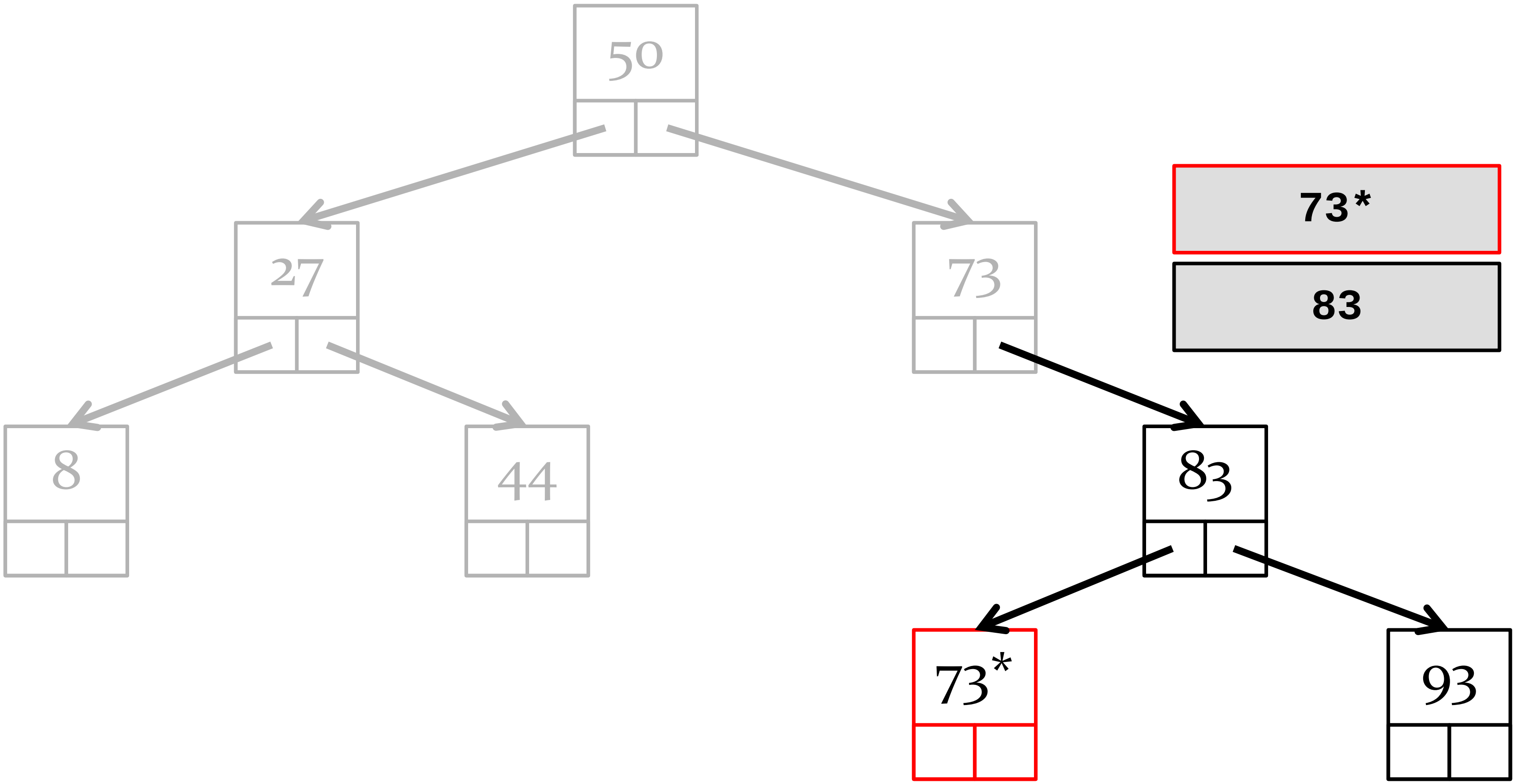
inorder: 8, 27, 44, 50, 73, 73*, 83, 93



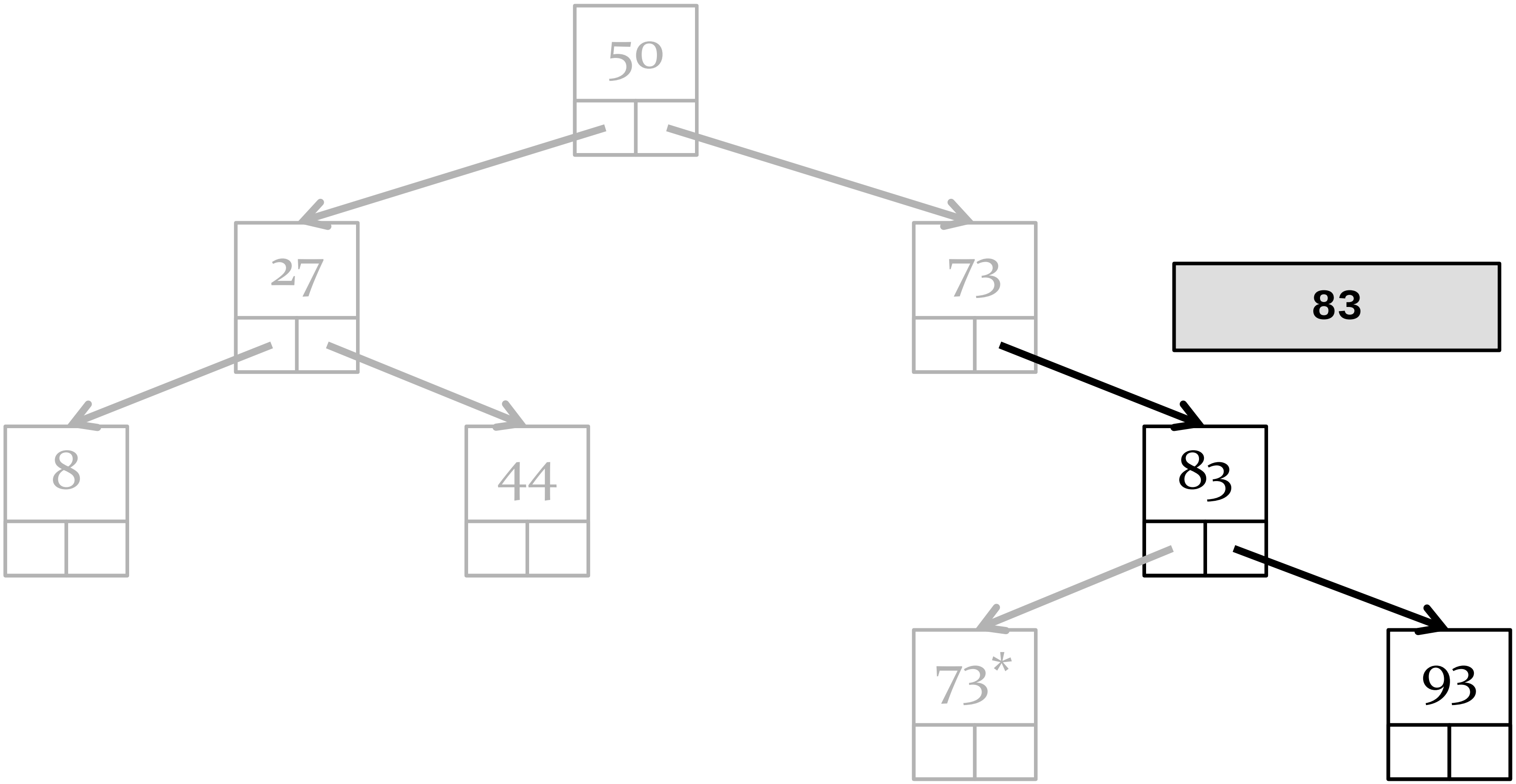
inorder: 8, 27, 44, 50, 73, 73*, 83, 93



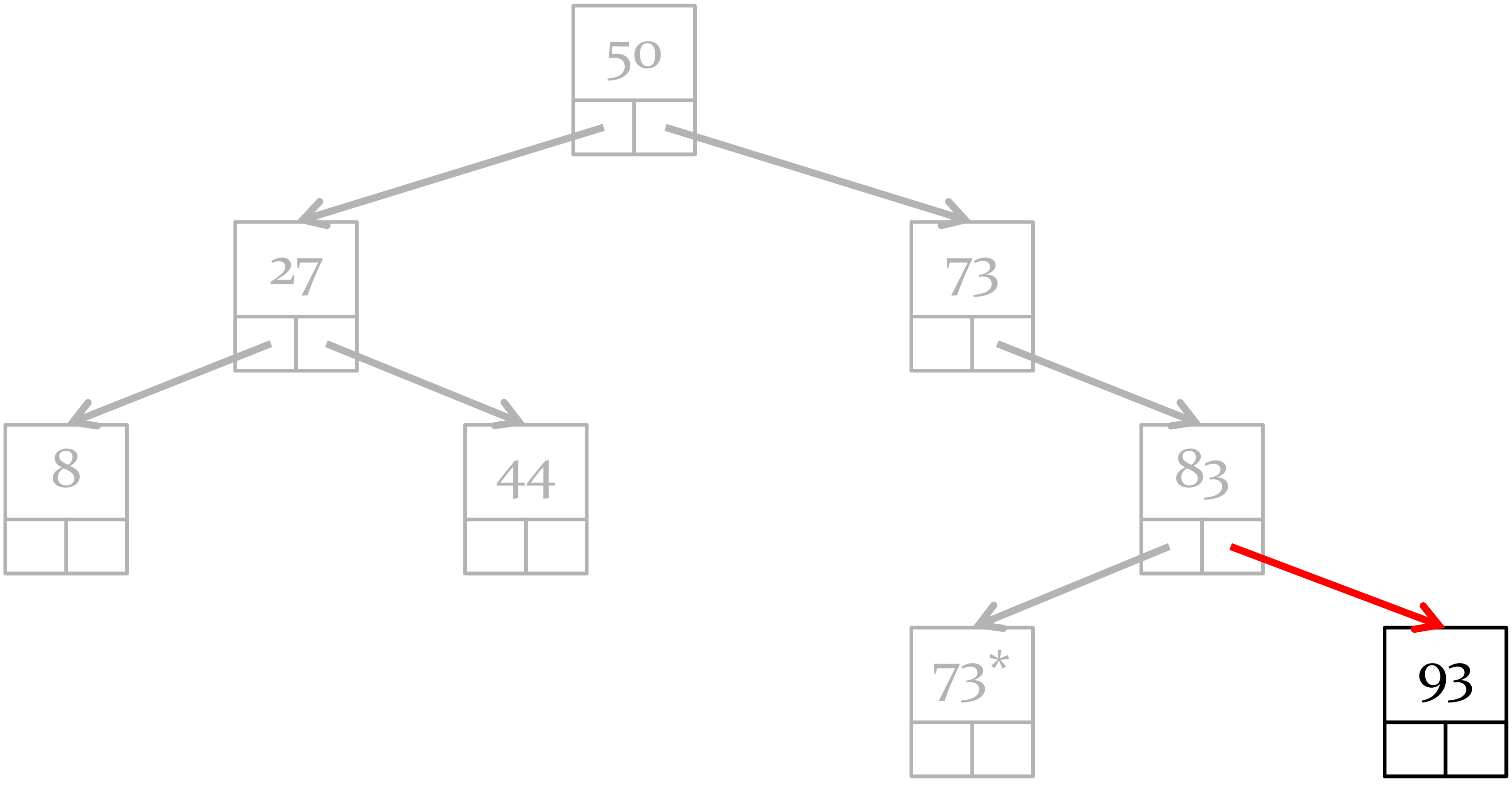
inorder: 8, 27, 44, 50, 73, 73*, 83, 93



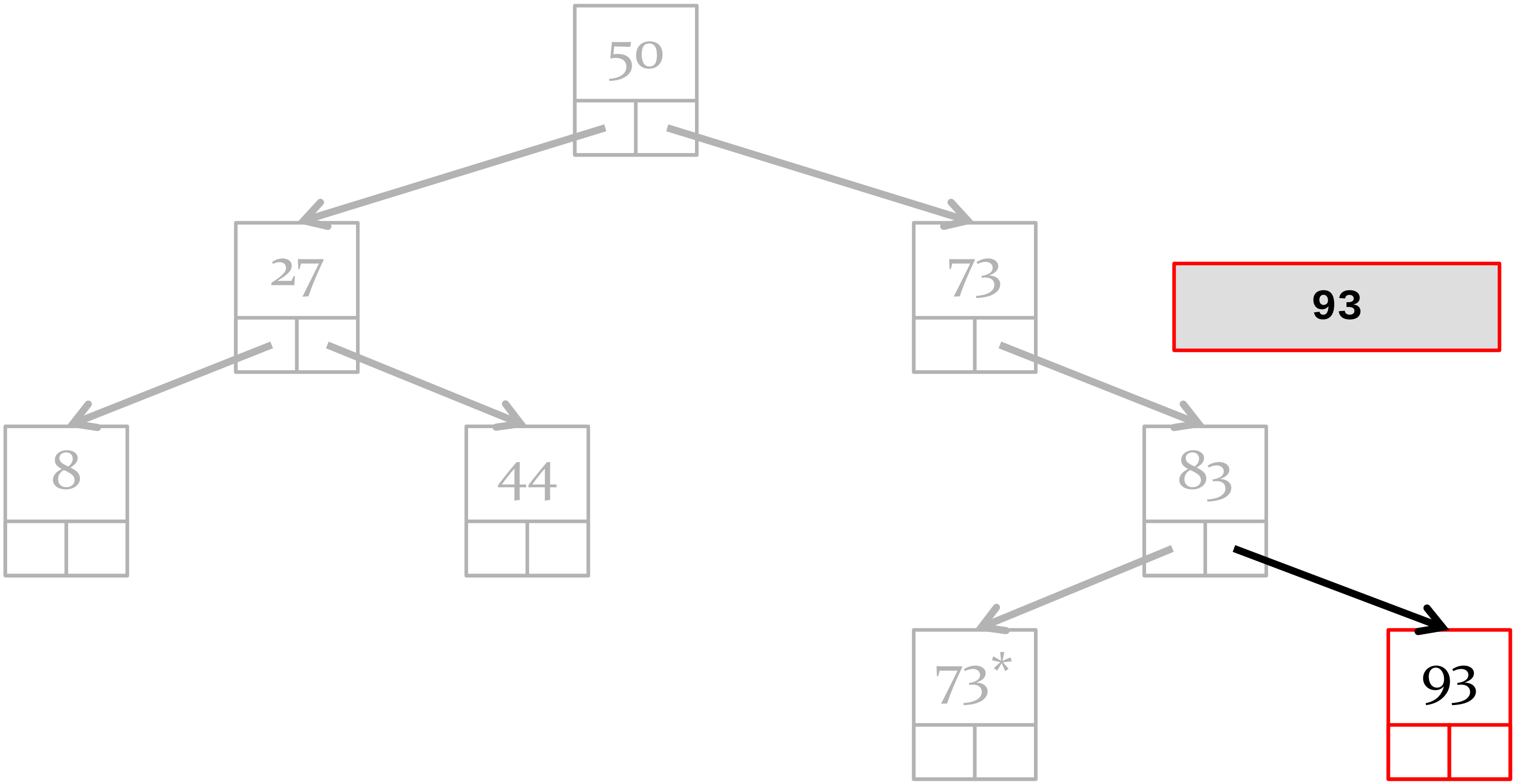
inorder: 8, 27, 44, 50, 73, 73*, 83, 93



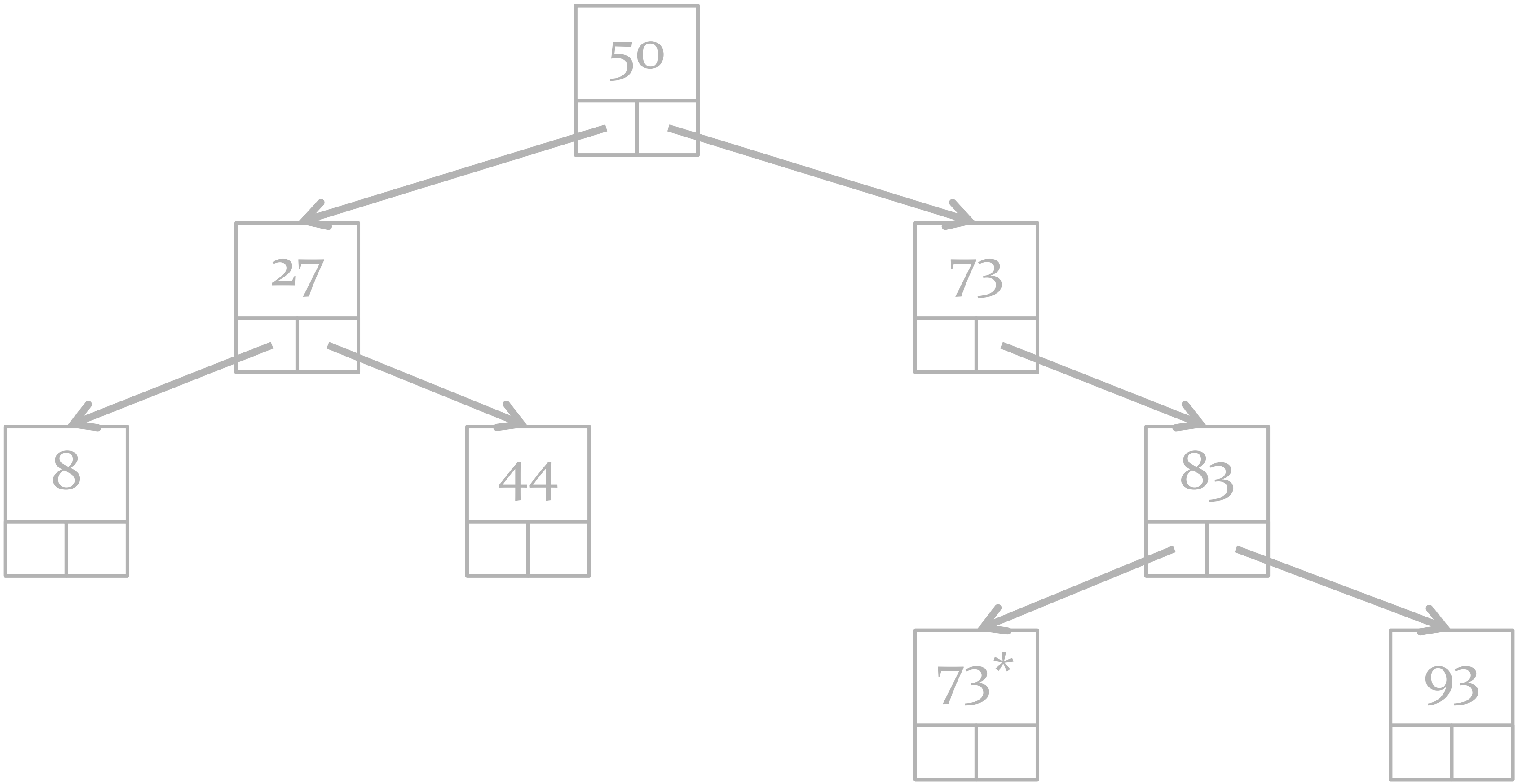
inorder: 8, 27, 44, 50, 73, 73*, 83, 93



inorder: 8, 27, 44, 50, 73, 73*, 83, 93



inorder: 8, 27, 44, 50, 73, 73*, 83, 93



inorder: 8, 27, 44, 50, 73, 73*, 83, 93

Implementation for BST

```
public String inorder() {
  StringBuilder b = new StringBuilder();
  Stack<Node<E>> st = new Stack<Node<E>();
  Node<E> n = this.root;
  while (!st.isEmpty() || n != null) {
    if (n != null) {
      st.push(n);
      n = n.left;
    }
    else {
      n = st.pop();
      b.append(n.data);
      n = n.right;
    }
  }

  return b.toString();
}
```