Exam information

- in lab Tue, 18 Apr 2017, 9:00-noon
- programming part
  - from inheritance onwards but no GUI programming
    - expect to see an inheritance question, recursion questions, data structure questions
- written part
  - short questions (facts, definitions, etc.) covering everything in the course
  - longer questions covering from inheritance onwards
    - expect to see a proof of correctness and termination of recursive a method

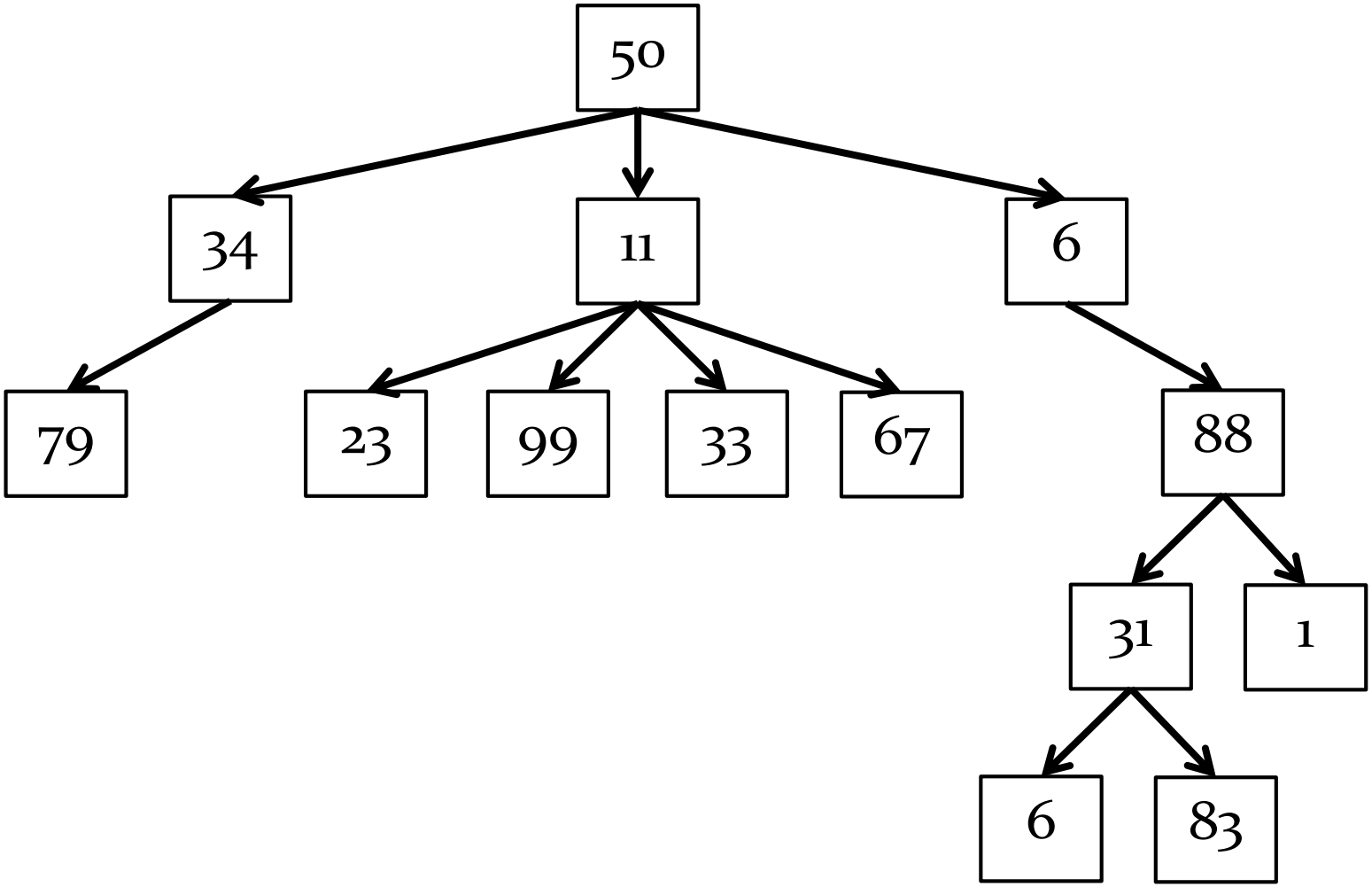
Trees

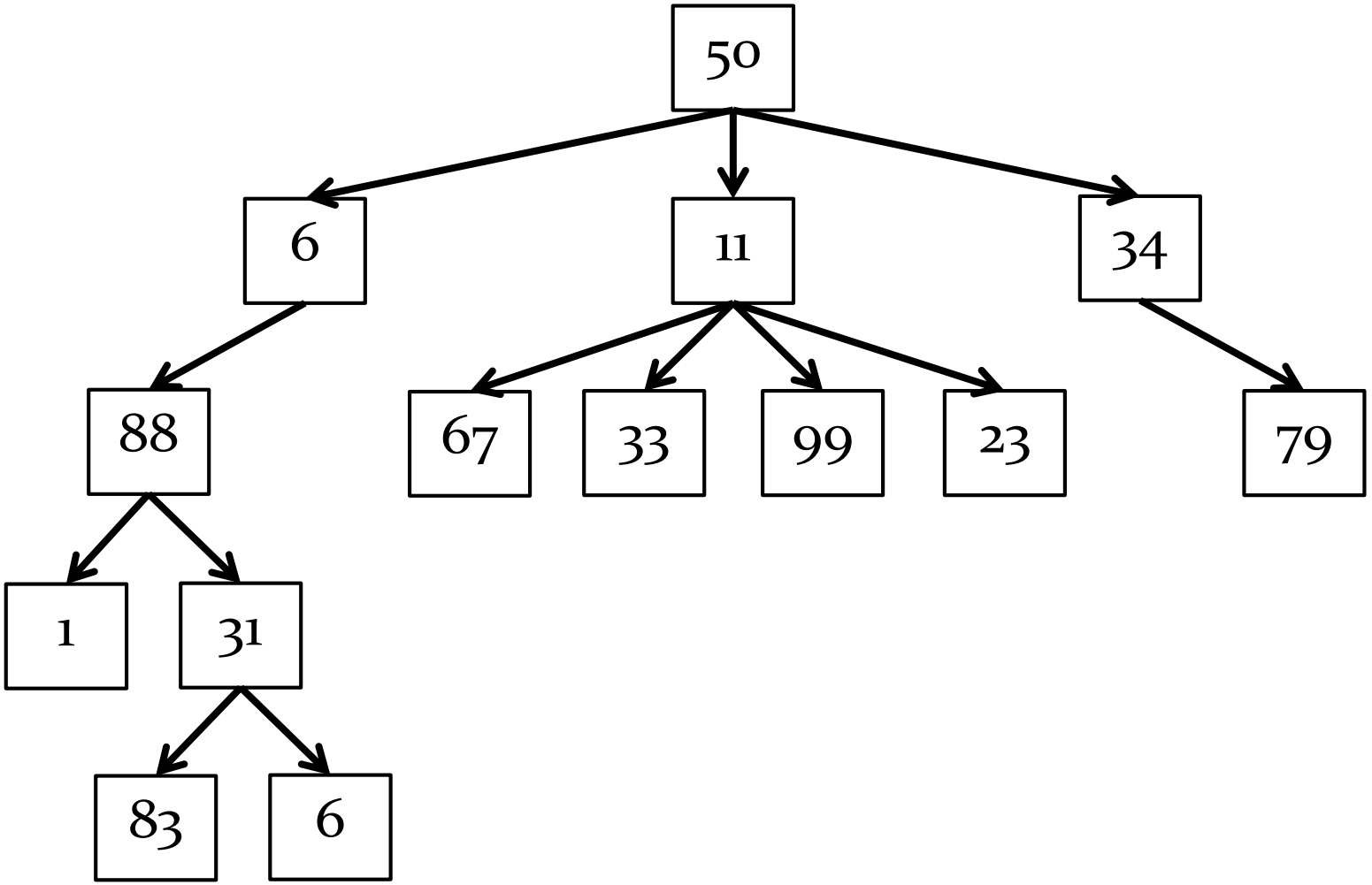
Graphs

- a graph is a data structure made up of nodes
  - each node stores data
  - each node has links to zero or more nodes
    - in graph theory the links are normally called *edges*
- graphs occur frequently in a wide variety of real-world problems
  - social network analysis
    - e.g., six-degrees-of-Kevin-Bacon, Facebook Friend Wheel
  - transportation networks
    - e.g., http://ac.fltmaps.com/en
  - many other examples
    - http://www.visualcomplexity.com/vc/

Trees

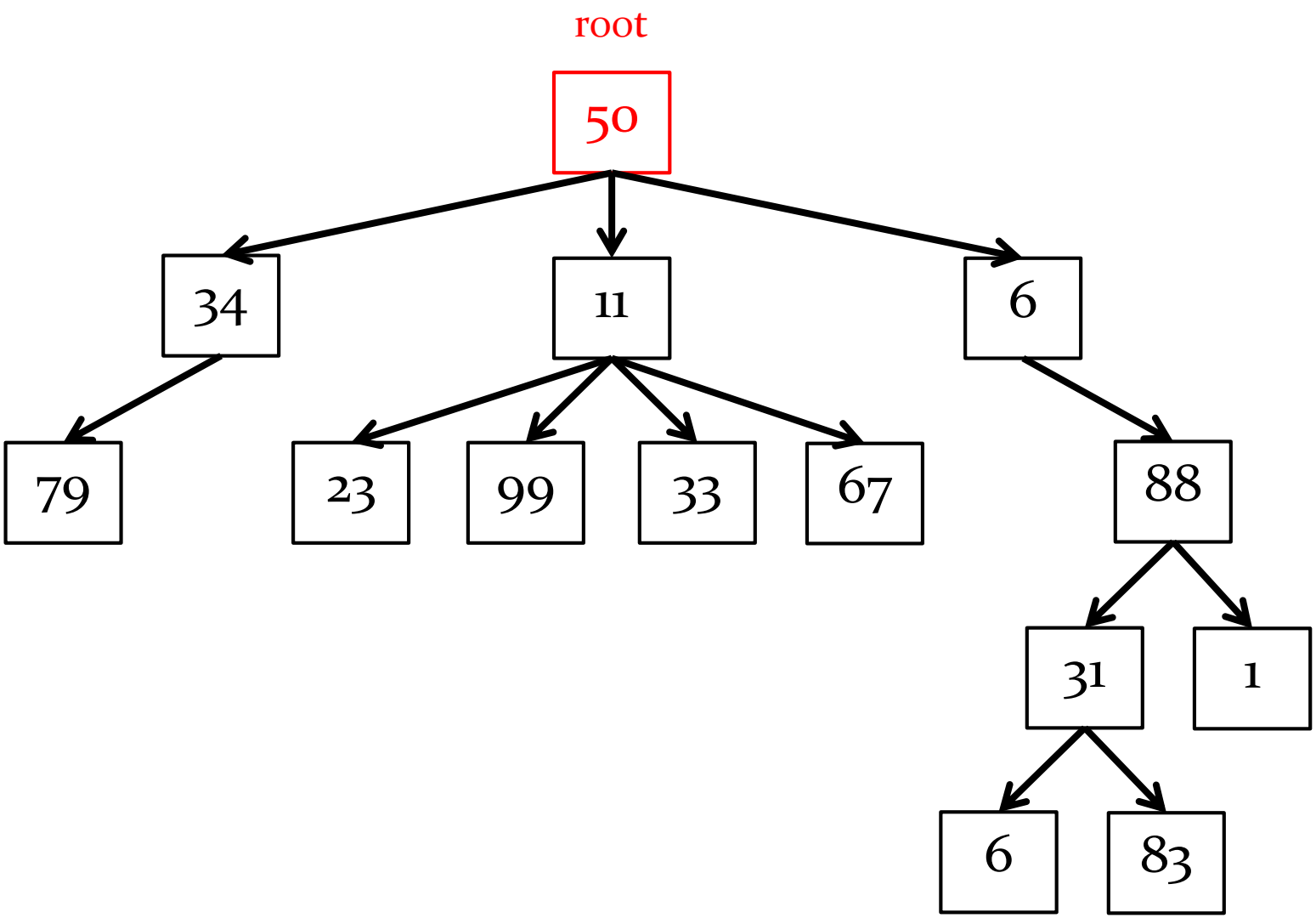
- trees are special cases of graphs
- a tree is a data structure made up of nodes
  - each node stores data
  - each node has links to zero or more nodes in the next level of the tree
    - children of the node
  - each node has exactly one parent node
    - except for the root node


50
34
11
6
79
23
99
33
67
88
31
1
6
83


50
6
11
34
88
67
33
99
23
79
1
31
83
6

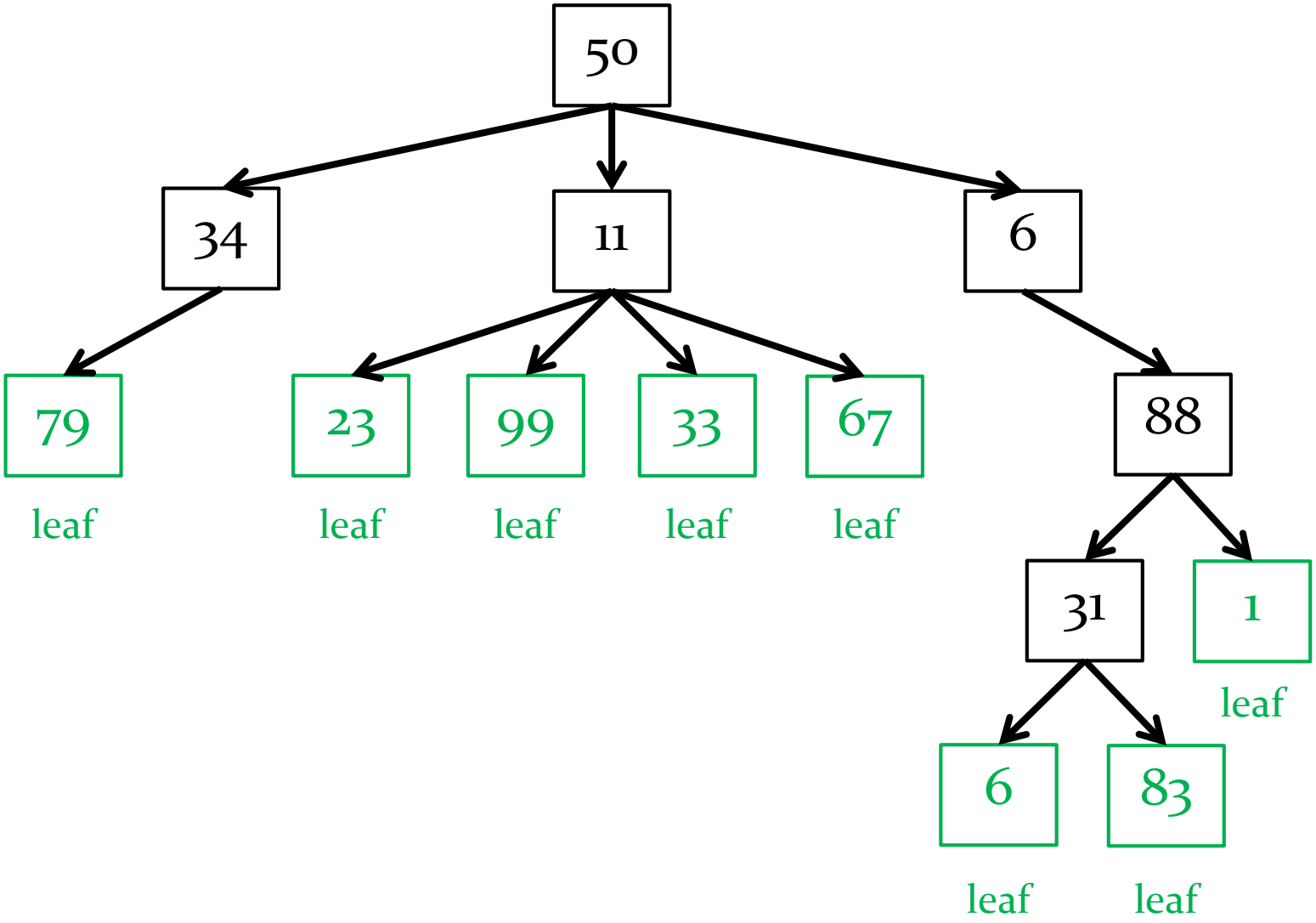
Trees

- the root of the tree is the node that has no parent node
- all algorithms start at the root


root
50
34
11
6
79
23
99
33
67
88
31
1
6
83

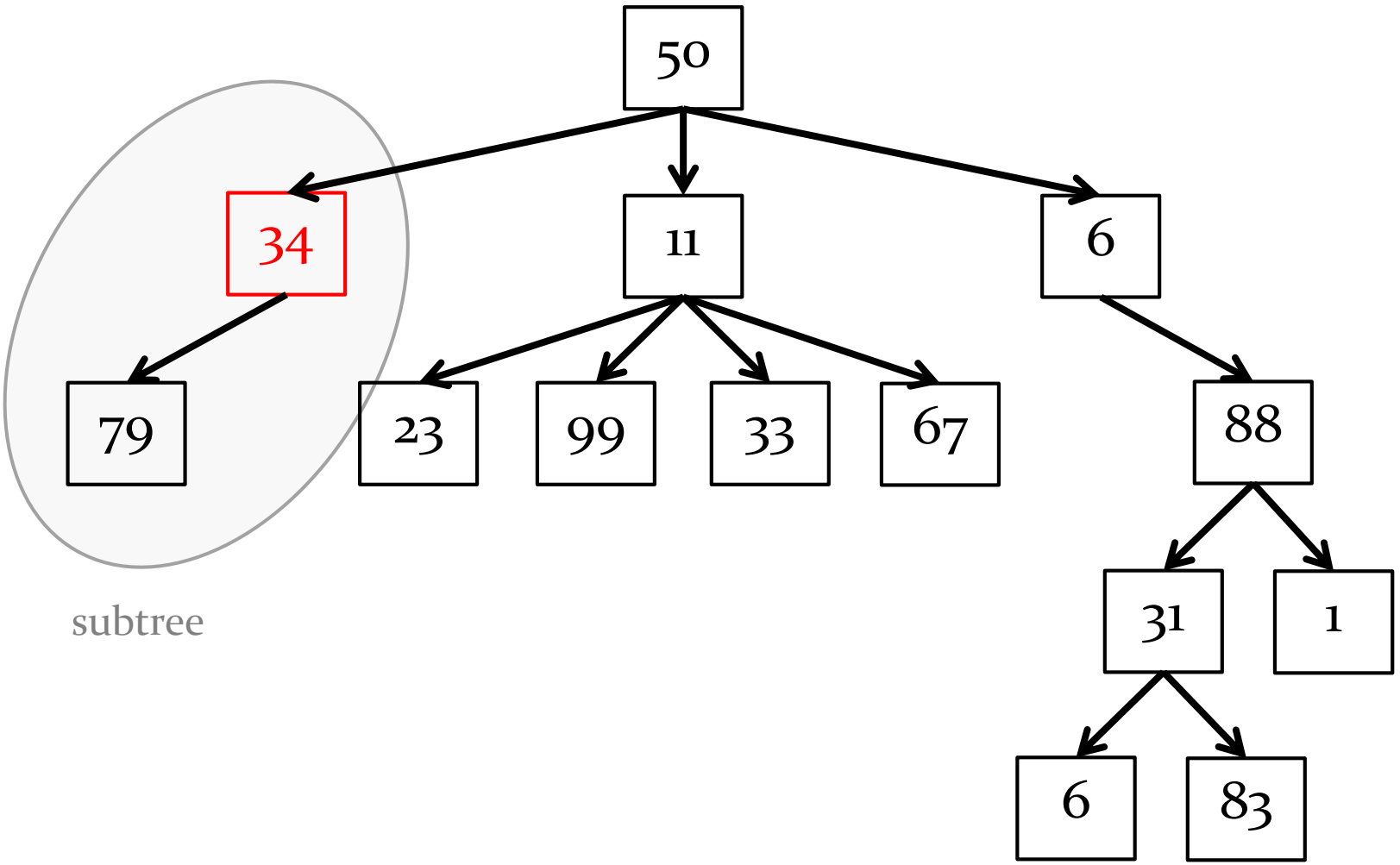
Trees

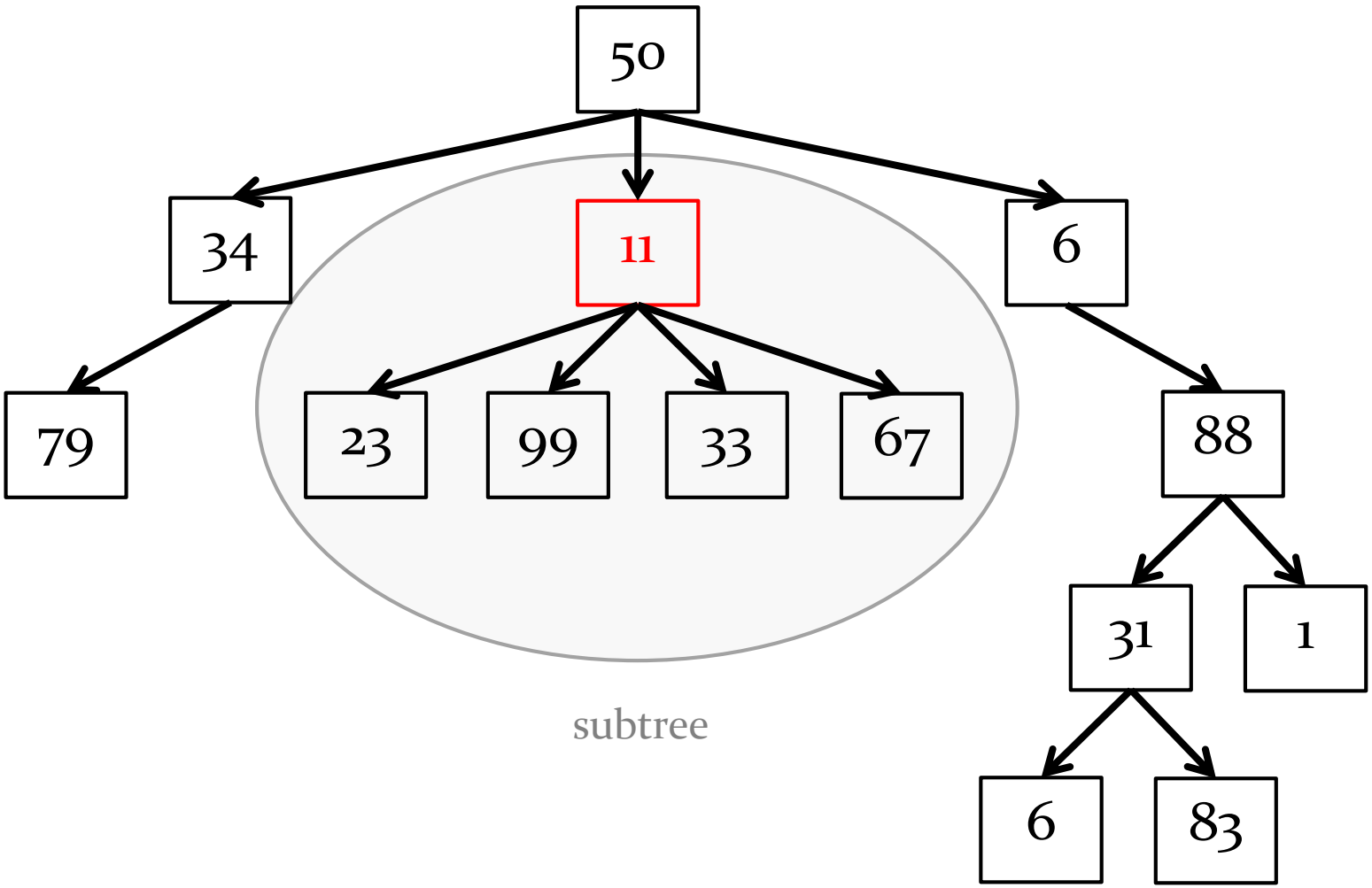
- a node without any children is called a leaf

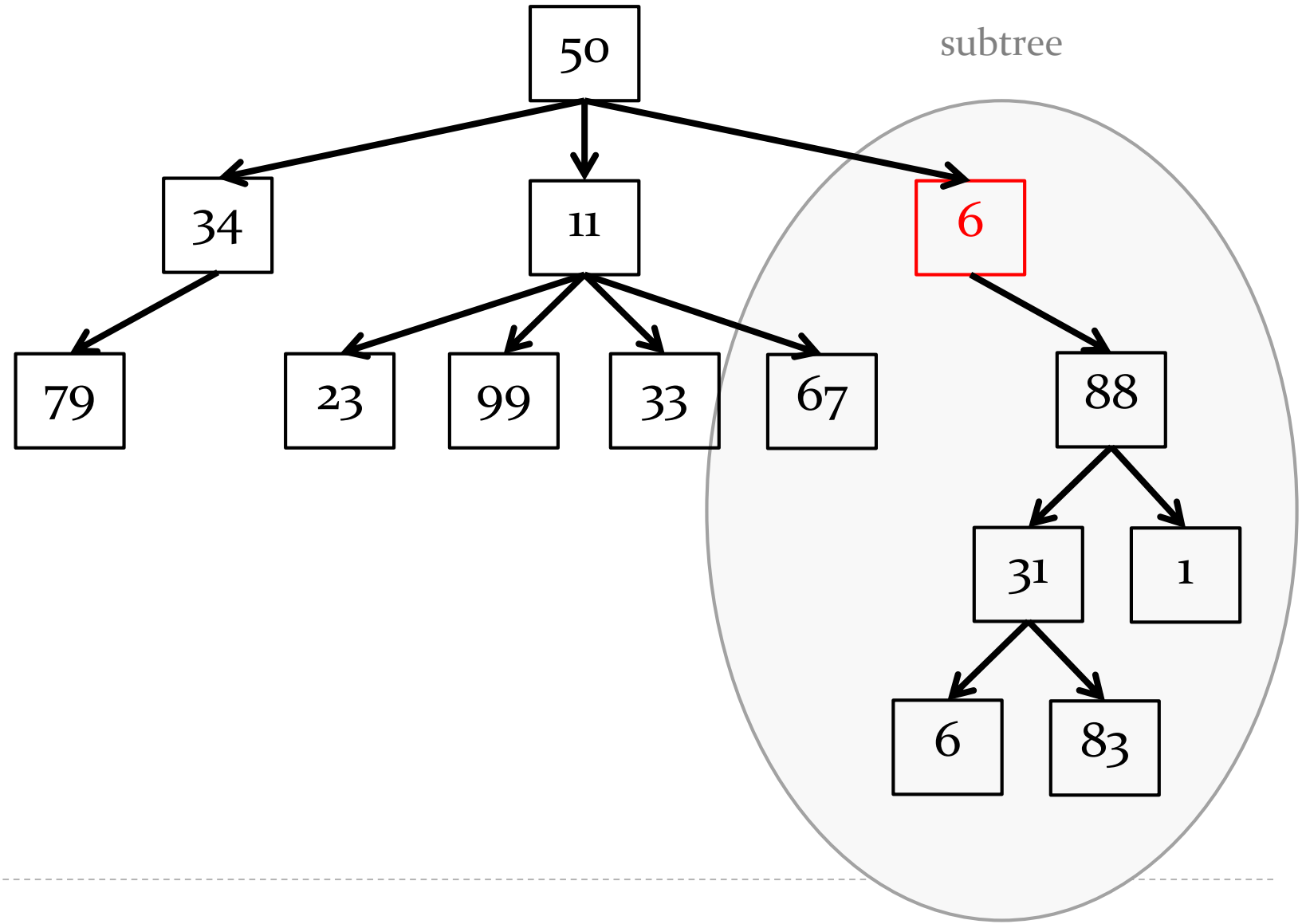

50
34
11
6
79
23
99
33
67
88
leaf
leaf
leaf
leaf
leaf
31
1
leaf
6
83
leaf
leaf

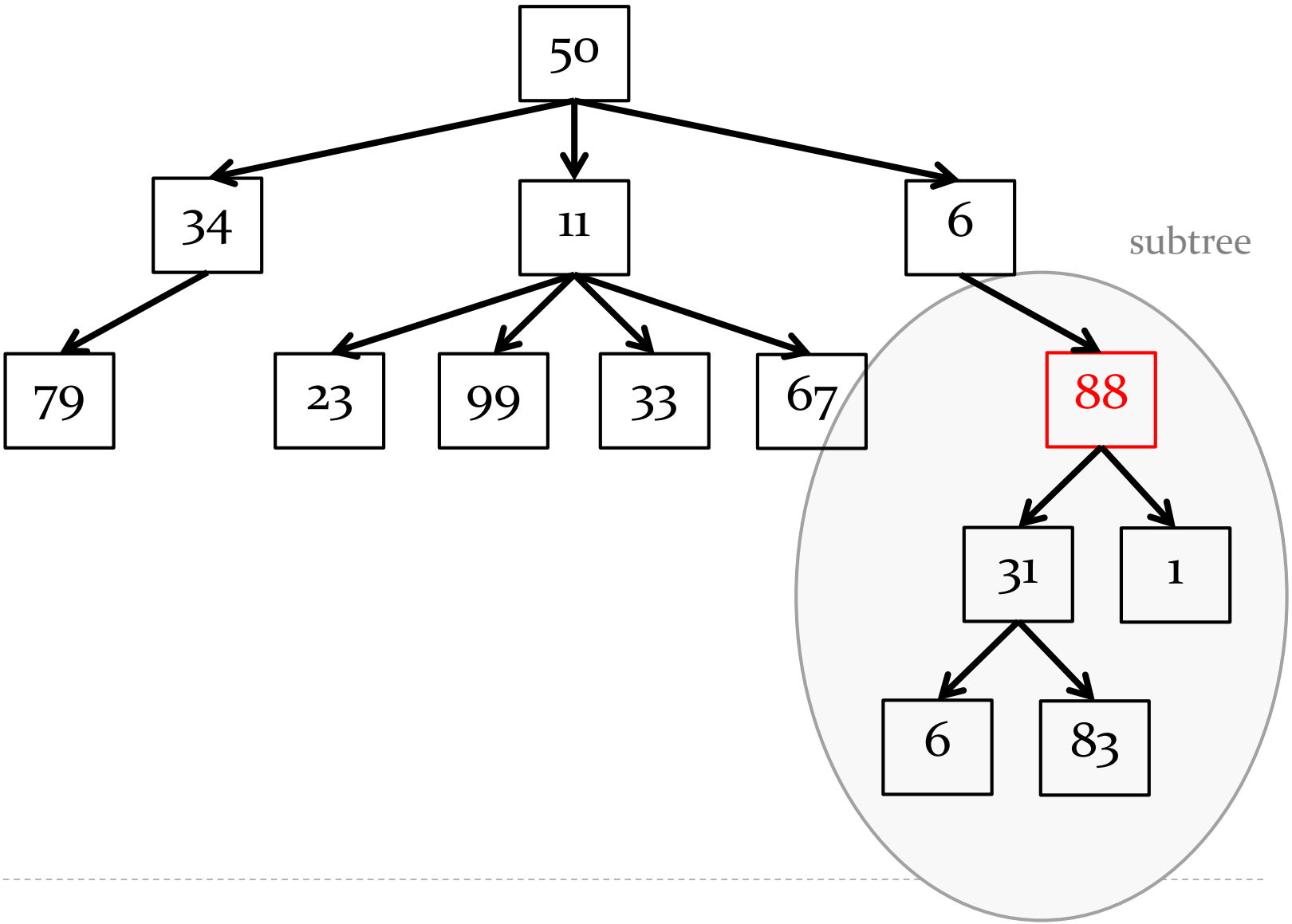
Trees

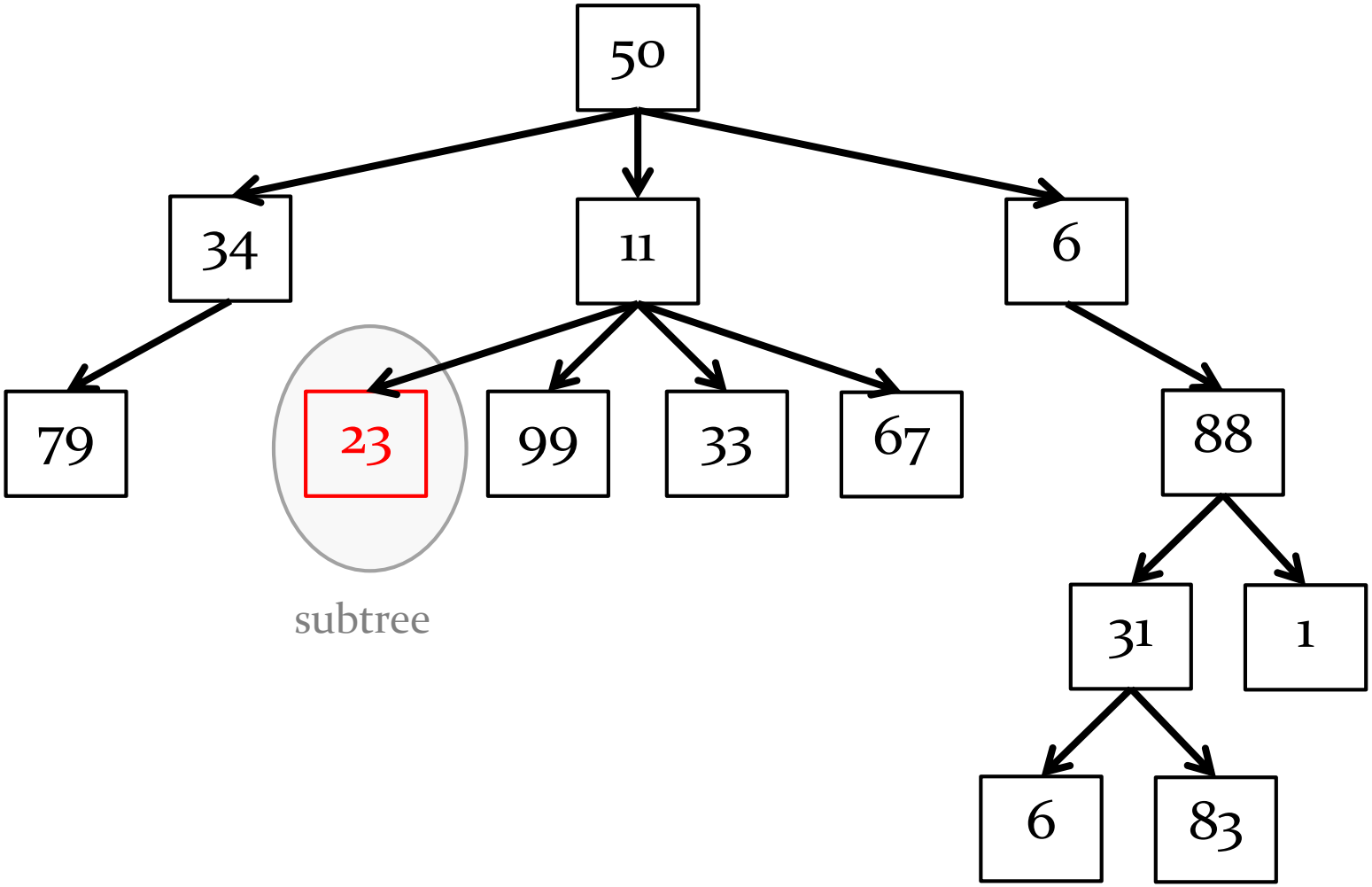
- the recursive structure of a tree means that every node is the root of a tree


50
34
11
6
79
23
99
33
67
88
31
1
6
83
subtree


50
34
11
6
79
23
99
33
67
88
31
1
6
83
subtree

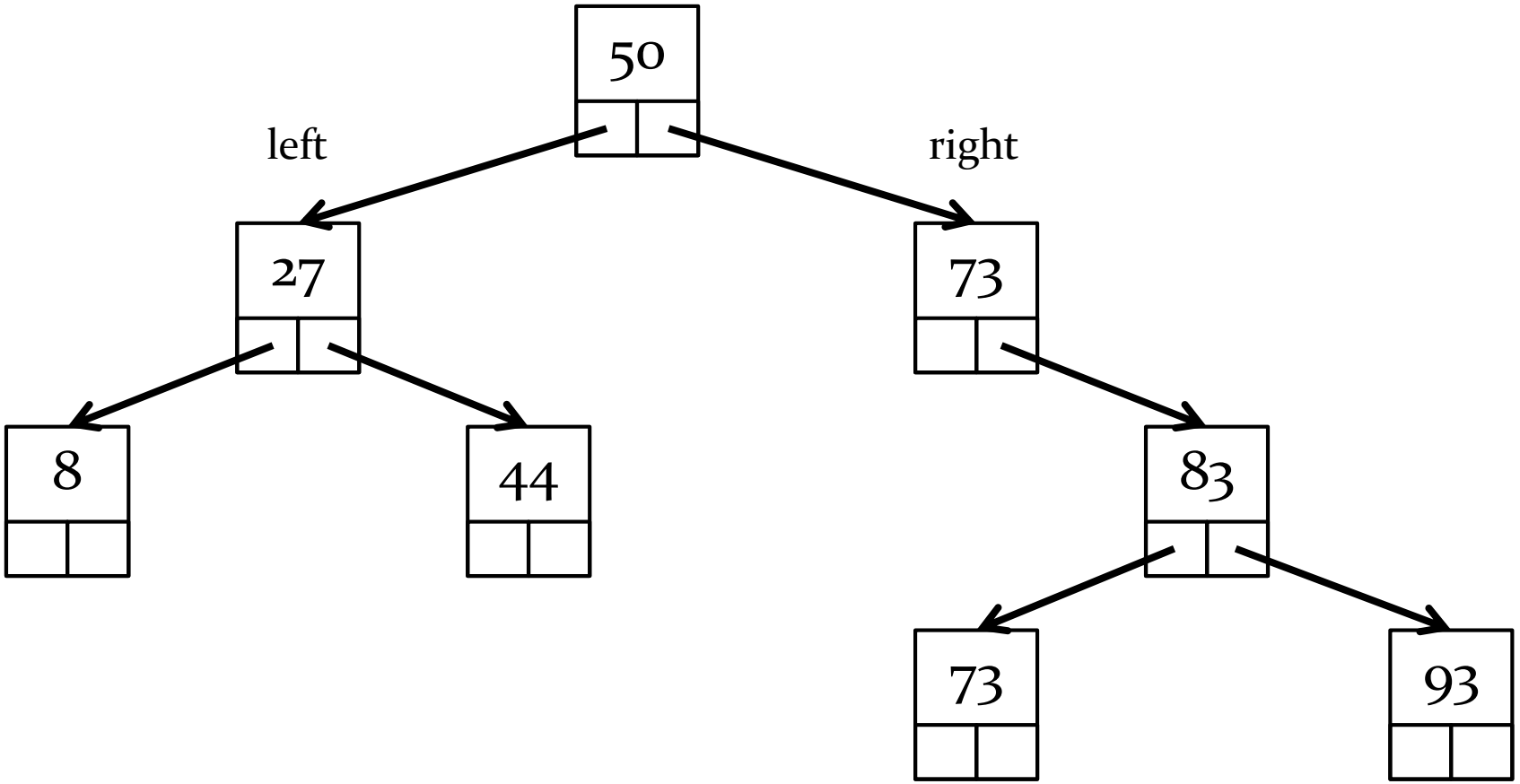

50
subtree
34
11
6
79
23
99
33
67
88
31
1
6
83

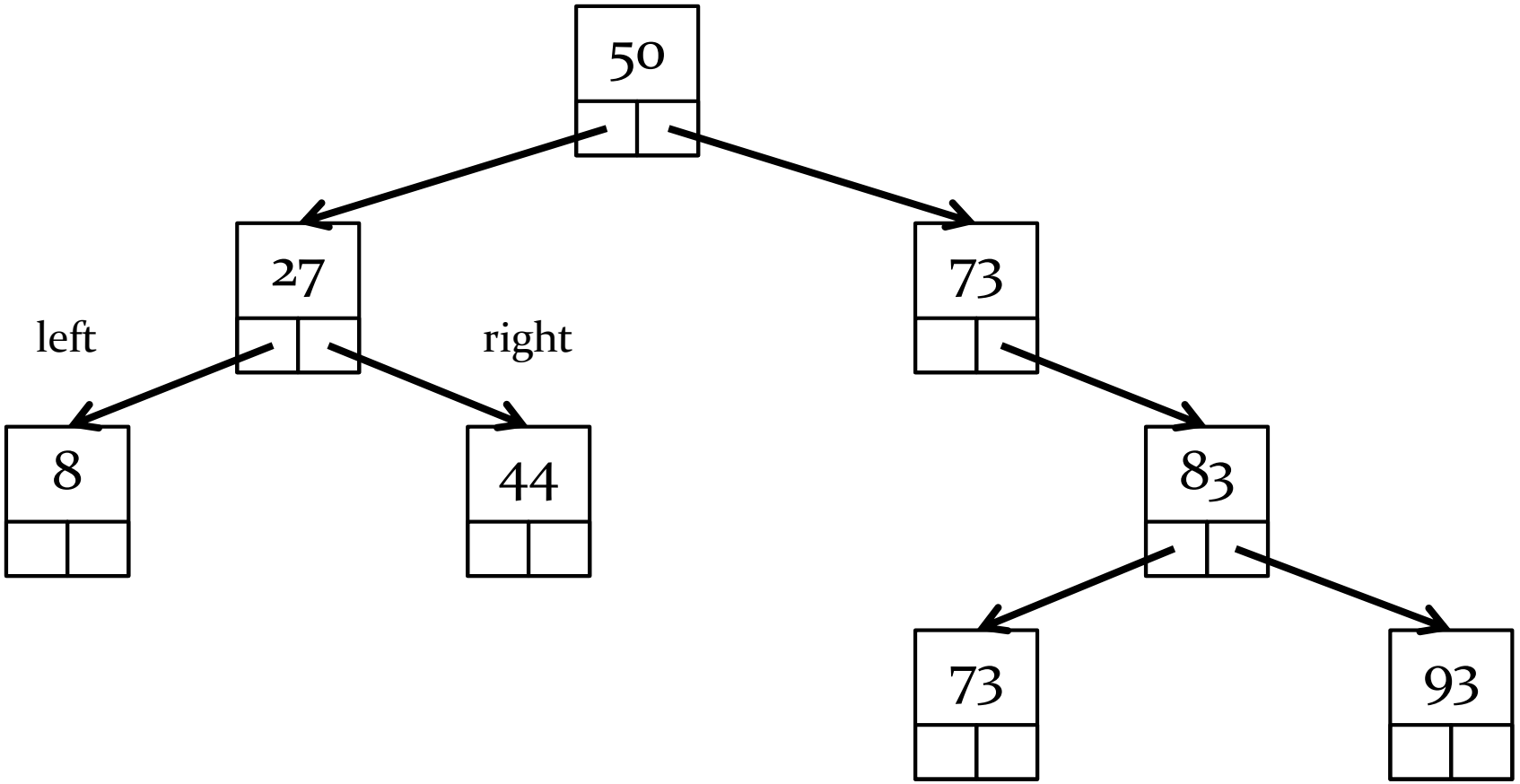

50
34
11
6
subtree
79
23
99
33
67
88
31
1
6
83

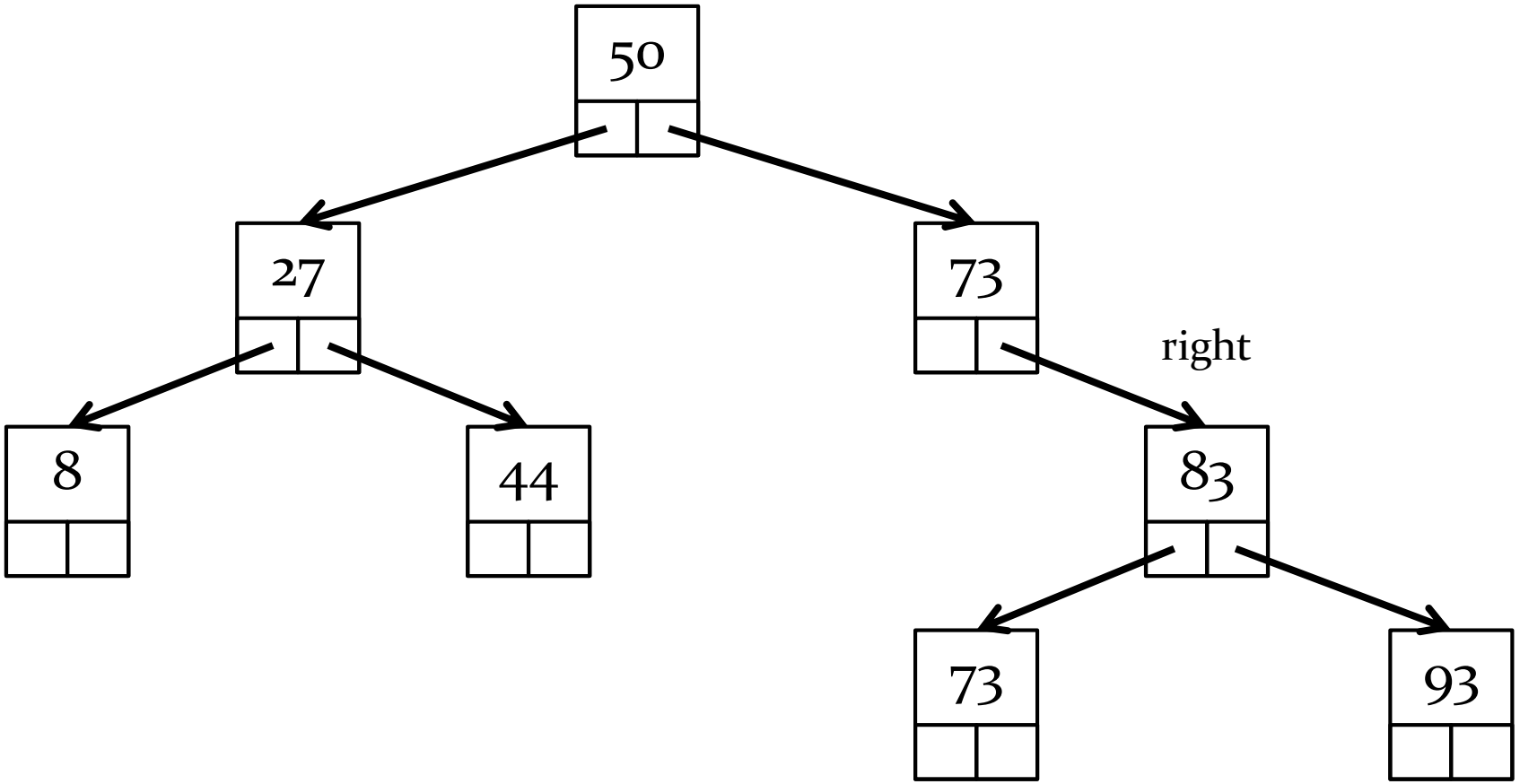

50
34
11
6
79
23
99
33
67
88
subtree
31
1
6
83

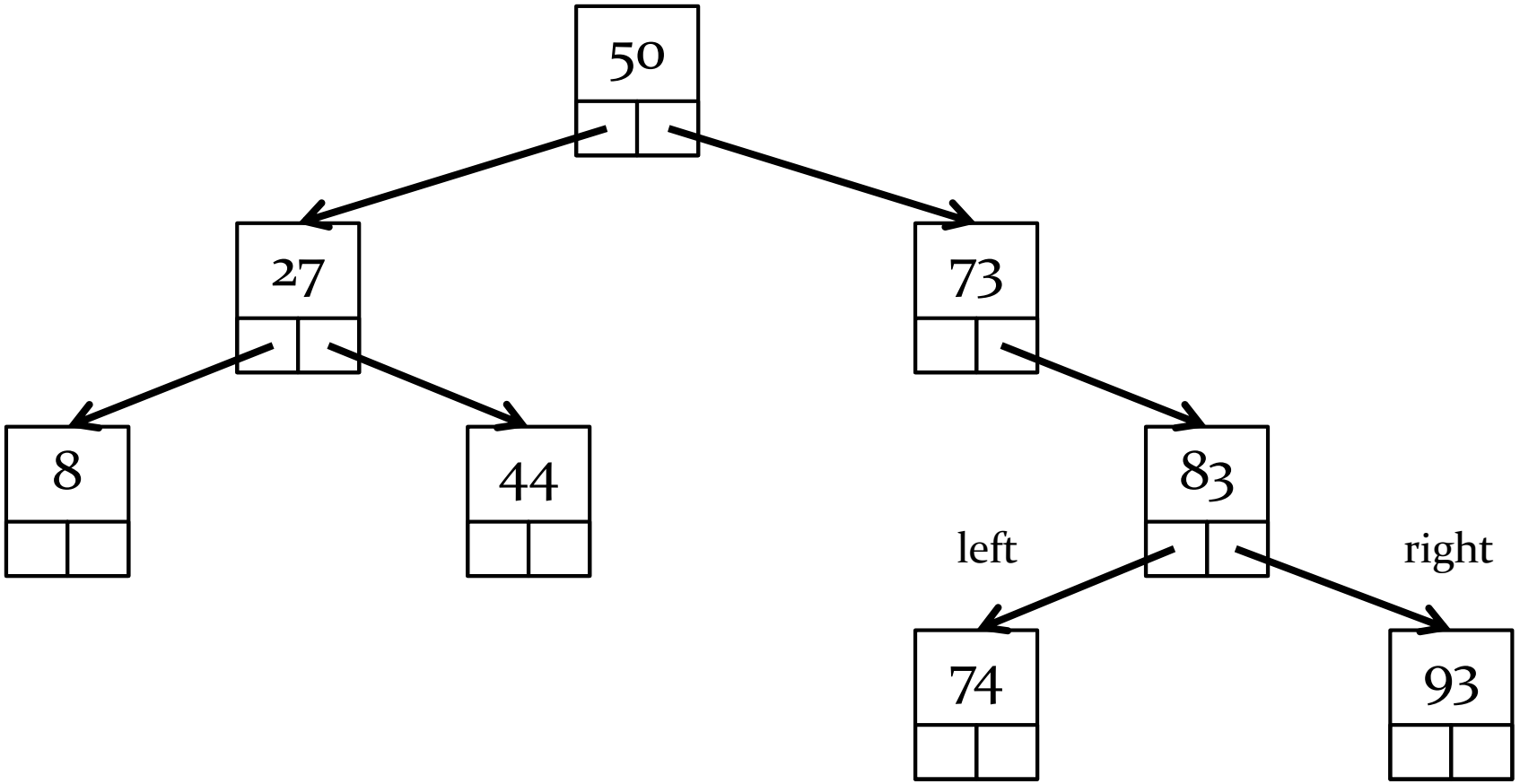
Binary Tree

- a binary tree is a tree where each node has at most two children
  - very common in computer science
  - many variations
- traditionally, the children nodes are called the left node and the right node


50
left
right
27
73
8
44
83
73
93


50
27
73
left
right
8
44
83
73
93


50
27
73
right
8
44
83
73
93


50
27
73
8
44
83
left
right
74
93

Binary Tree Algorithms

- the recursive structure of trees leads naturally to recursive algorithms that operate on trees
- for example, suppose that you want to search a binary tree for a particular element

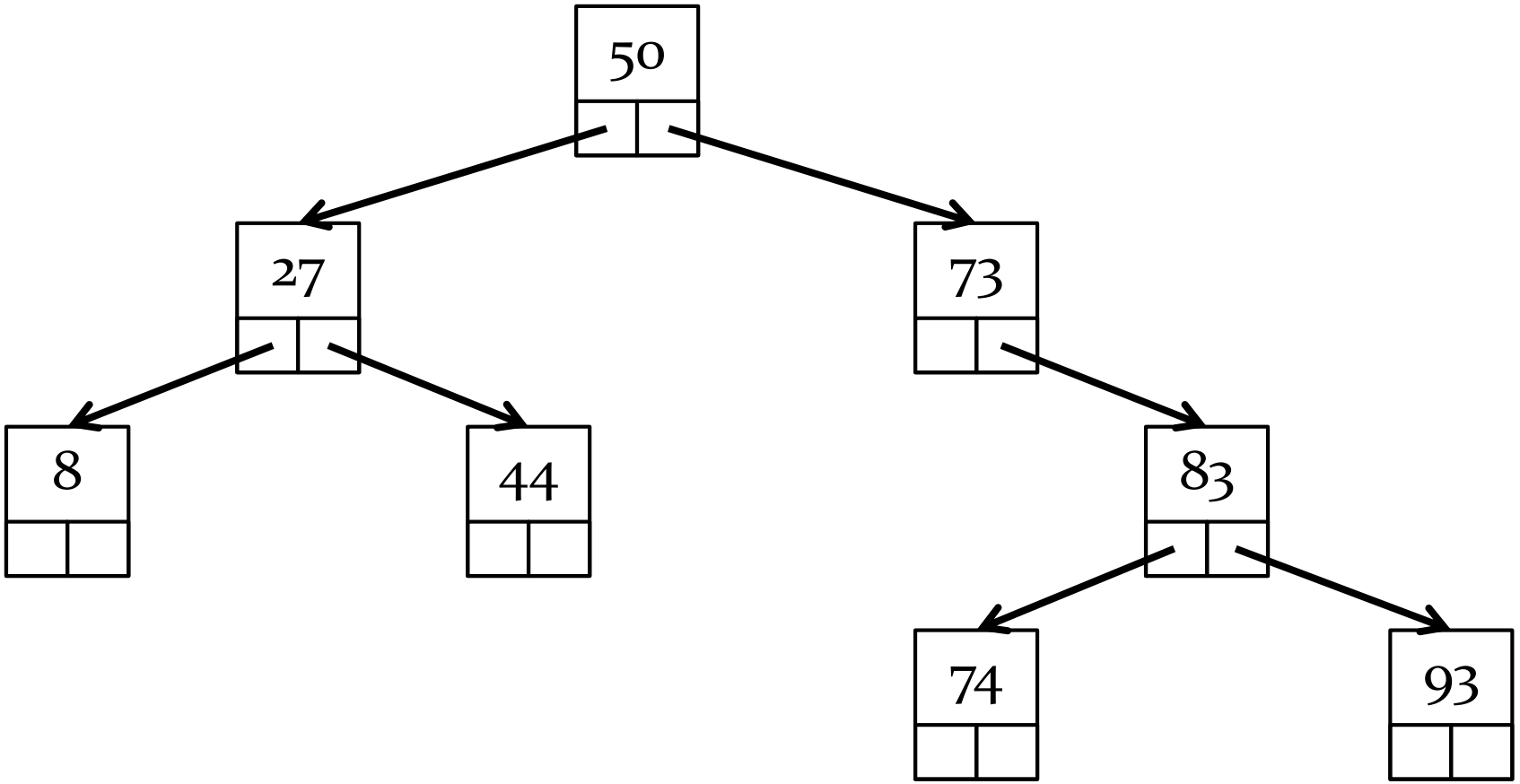
```
public static <E> boolean contains(E element, Node<E> node) {
  if (node == null) {
    return false;
  }
  if (element.equals(node.data)) {
    return true;
  }
  boolean inLeftTree = contains(element, node.left);
  if (inLeftTree) {
    return true;
  }
  boolean inRightTree = contains(element, node.right);
  return inRightTree;
}
```

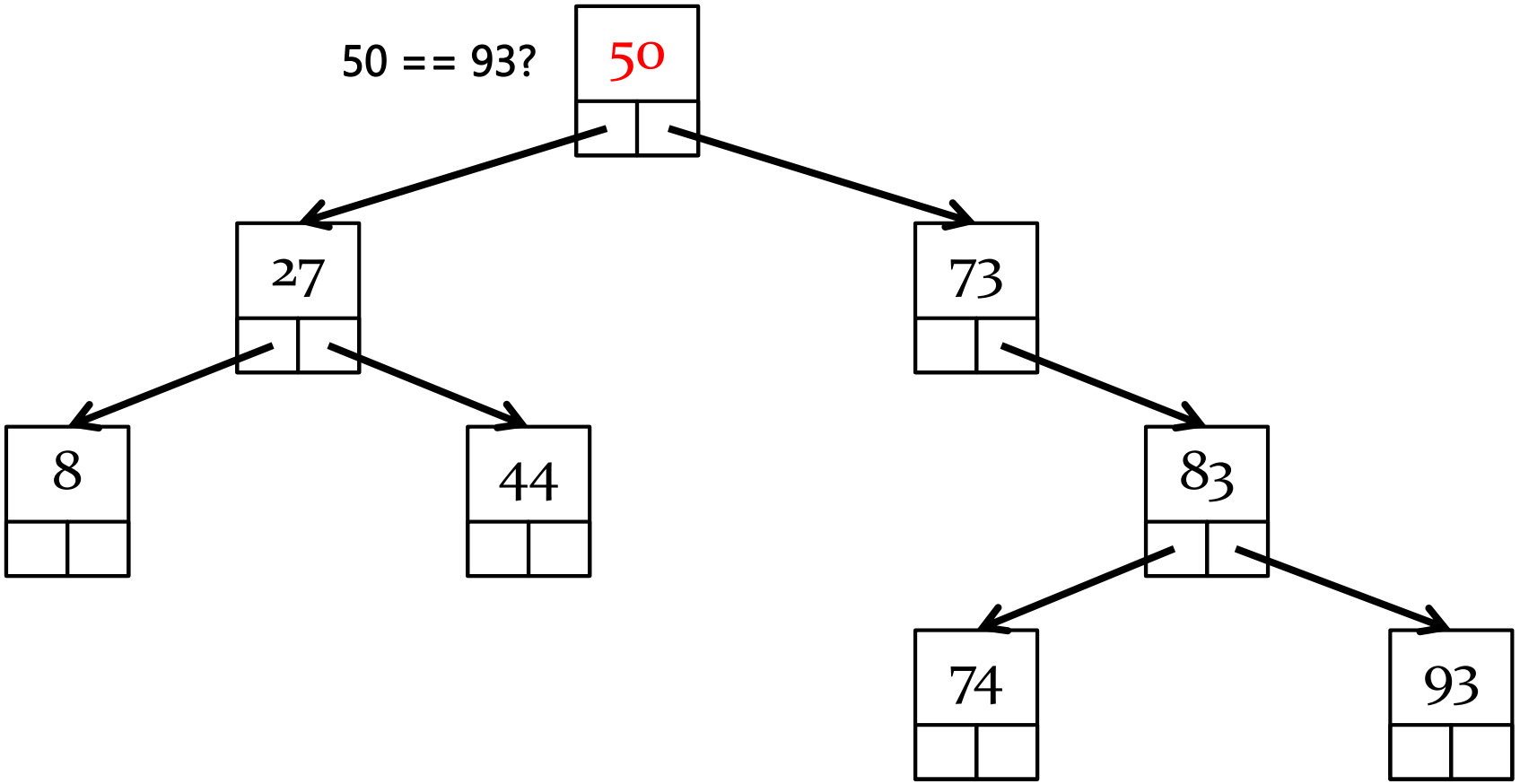
examine root

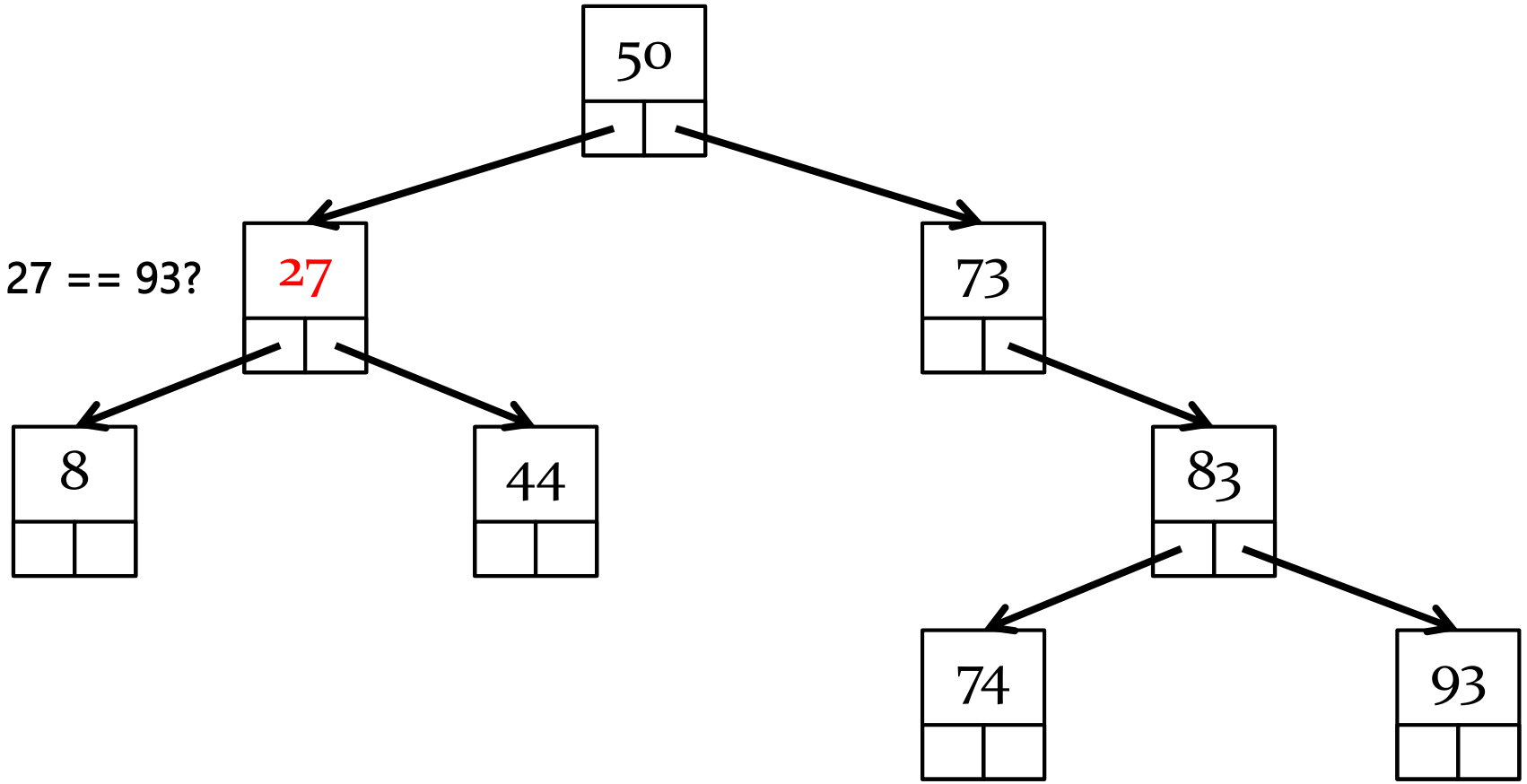
examine left subtree

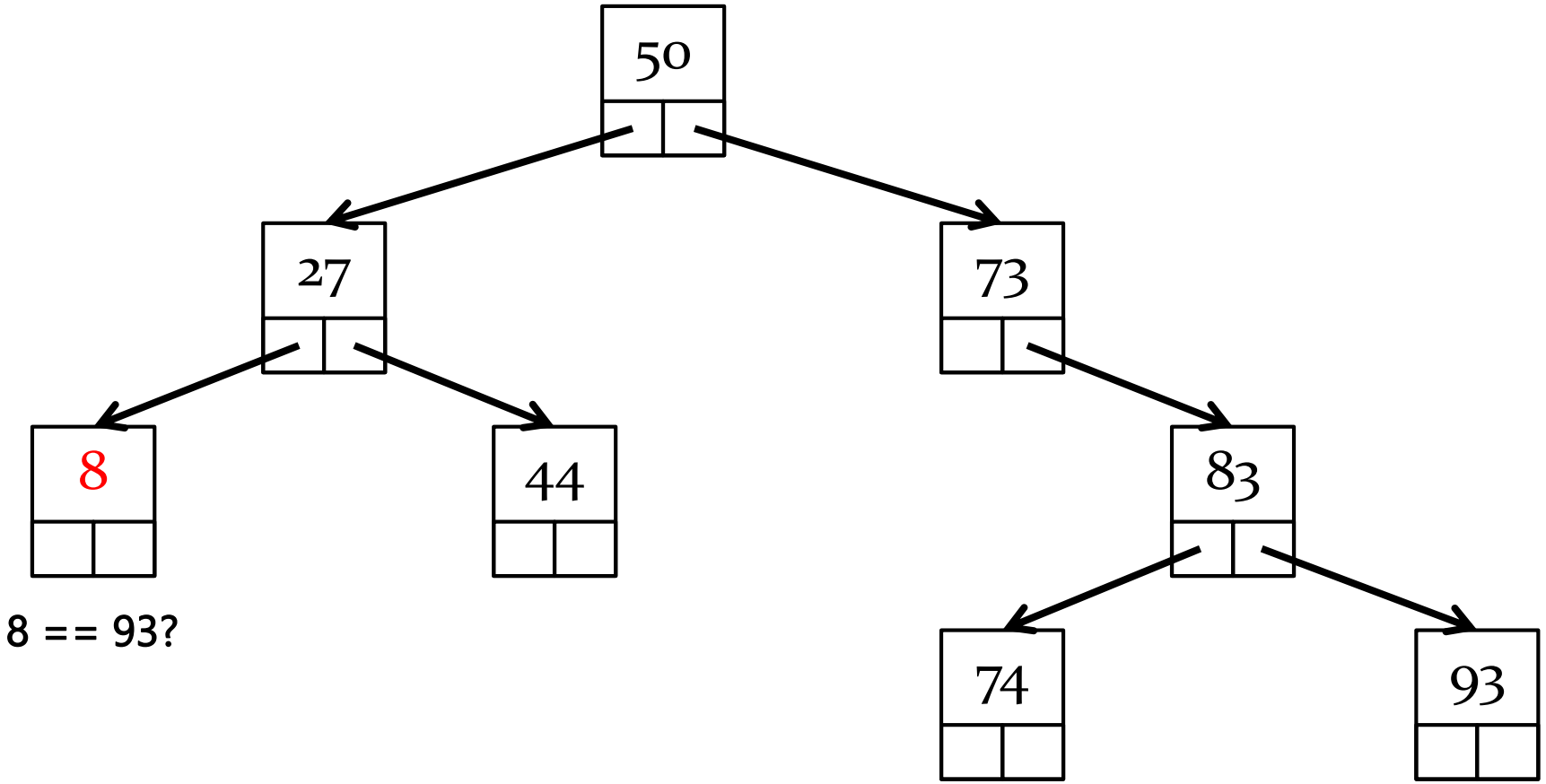
examine right subtree

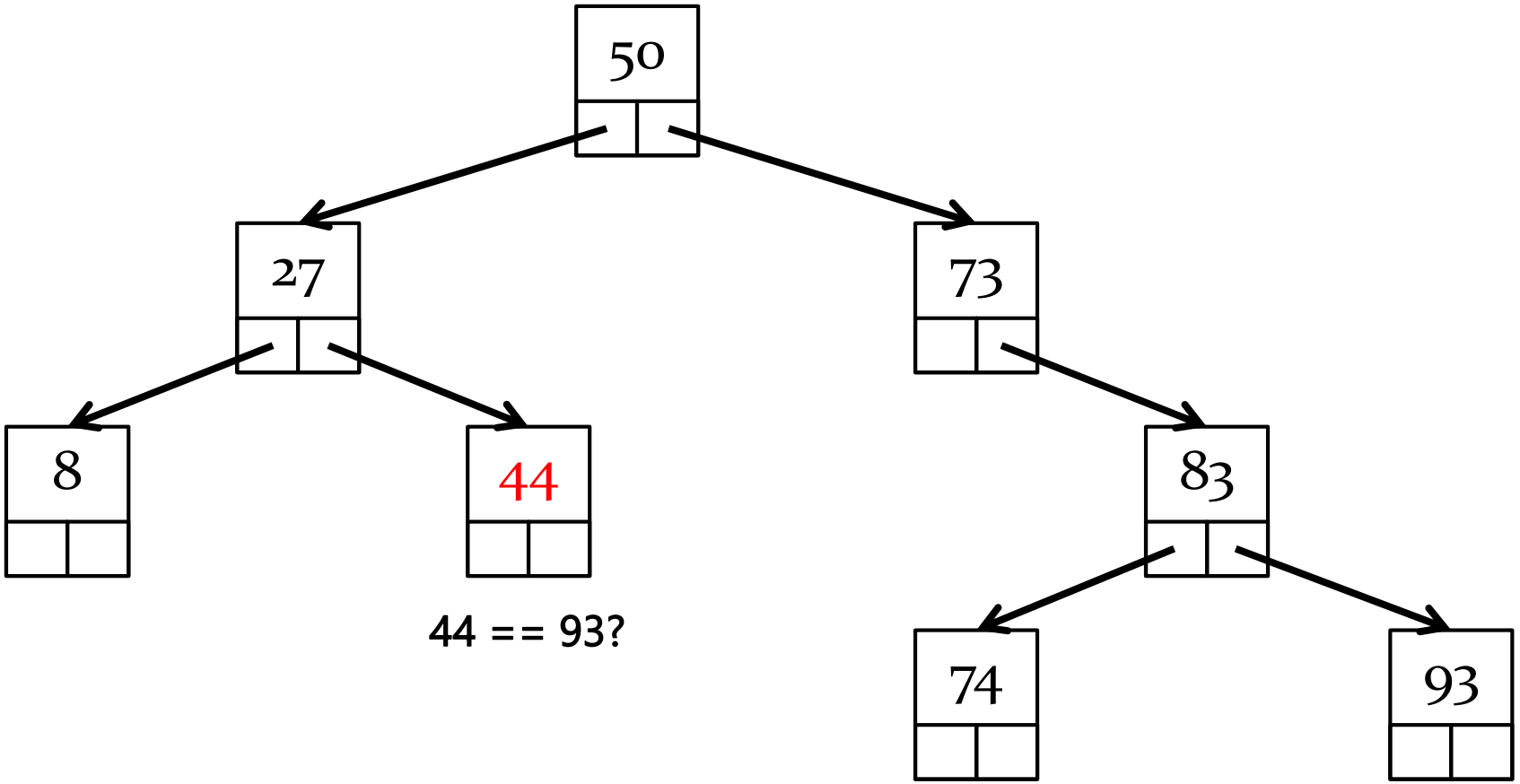
t.contains(93)

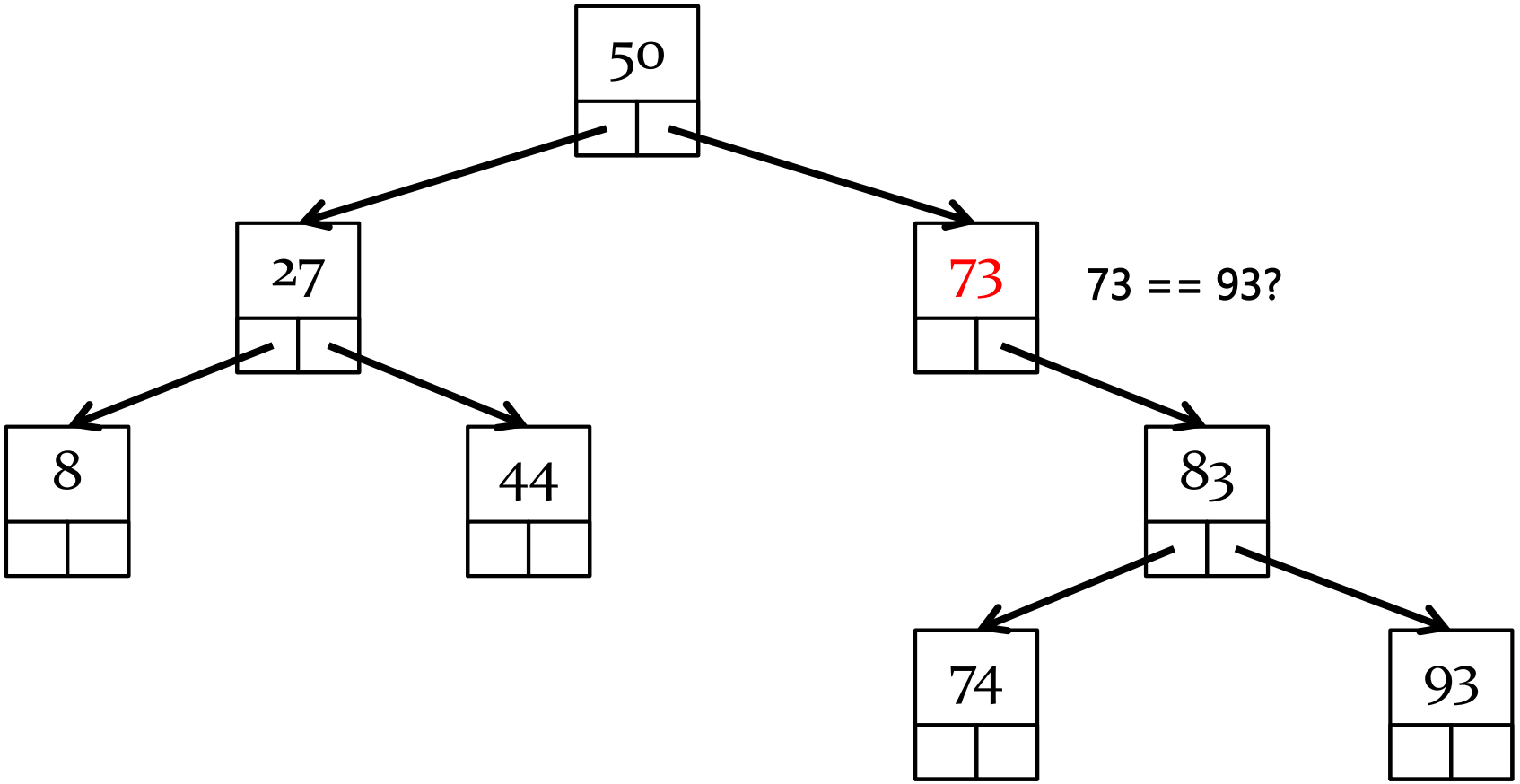


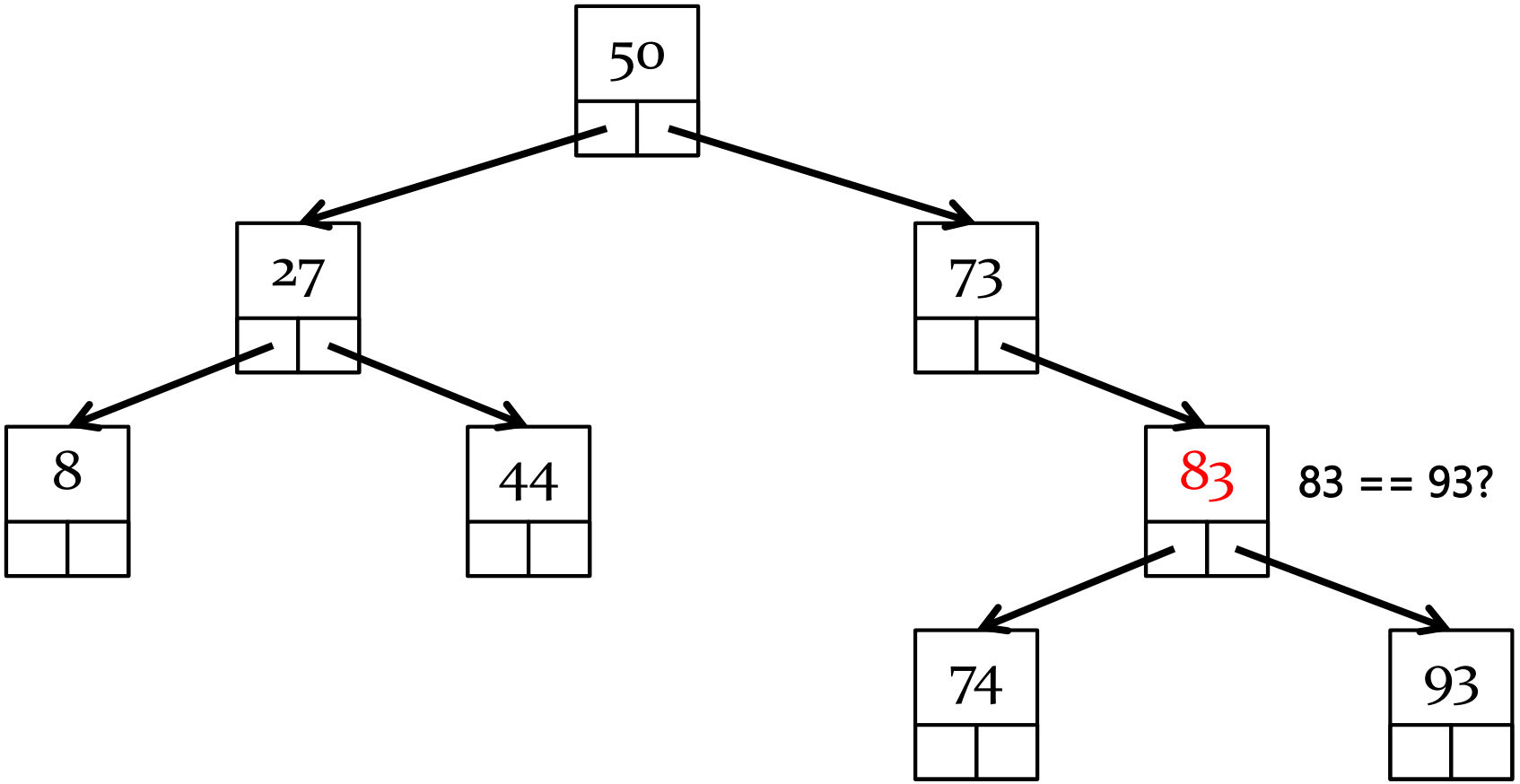

50 == 93?
50
27
73
8
44
83
74
93

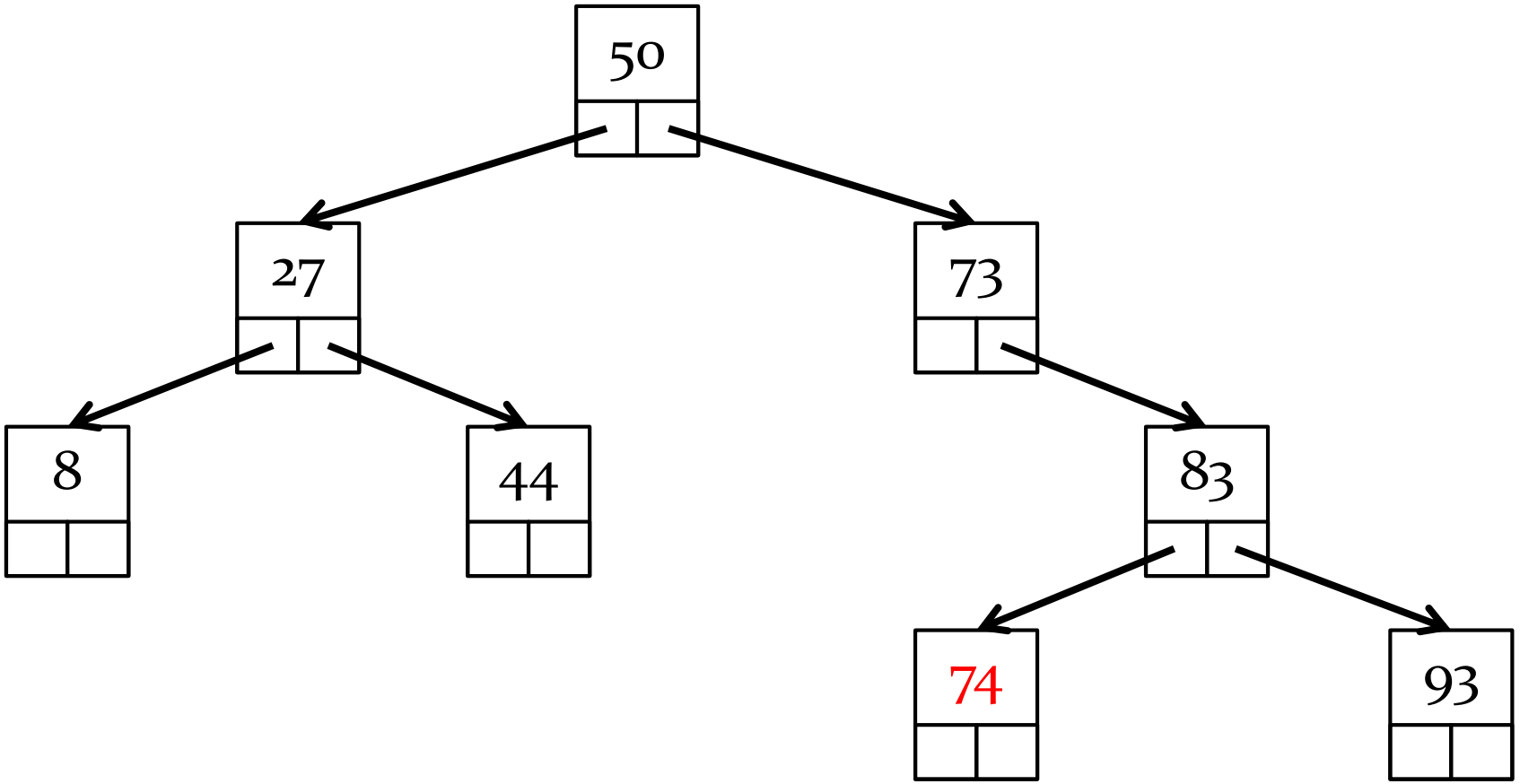

50
27 == 93?
27
73
8
44
83
74
93


50
27
73
8
44
83
74
93
8 == 93?

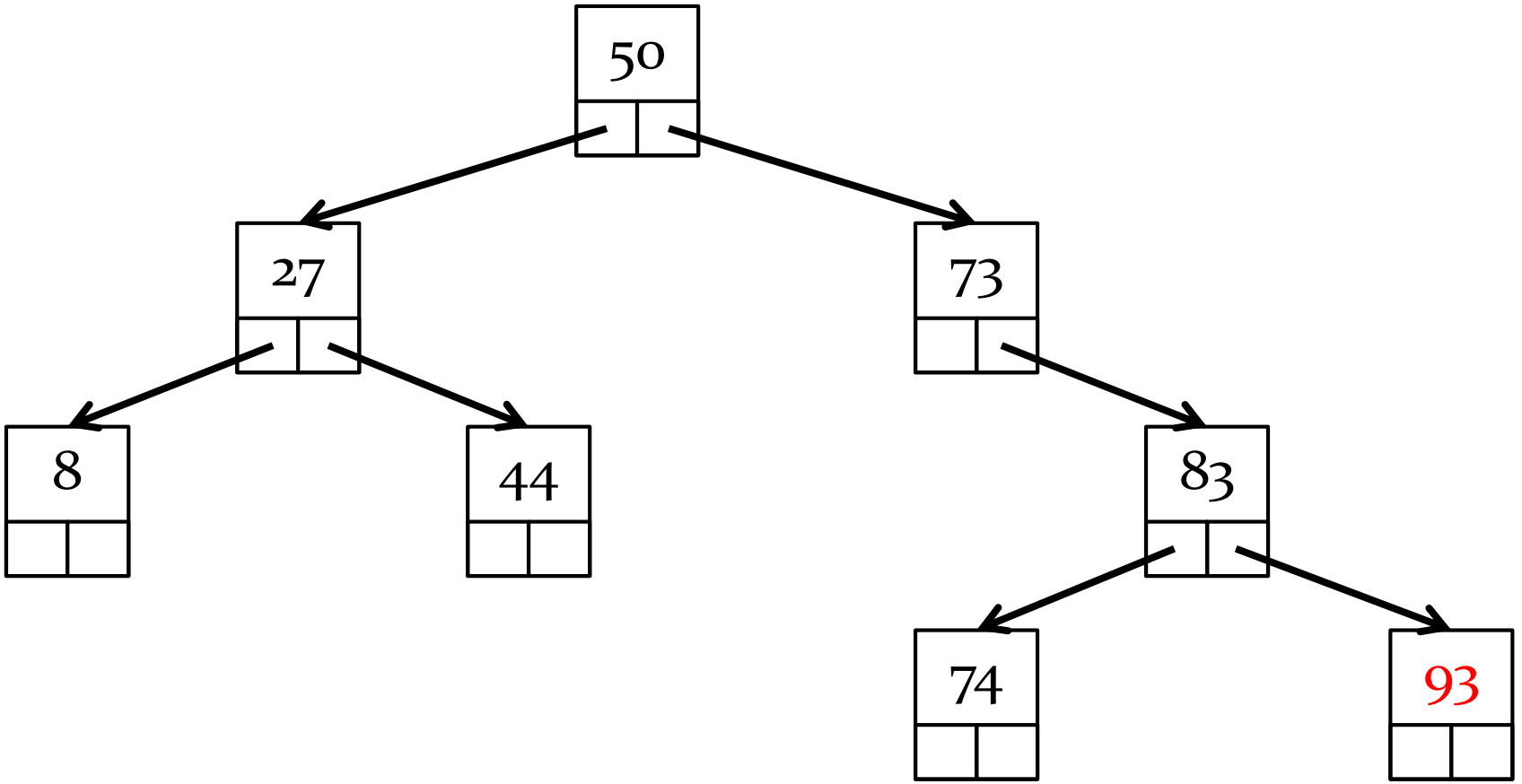

50
27
73
8
44
83
44 == 93?
74
93


50
27
73
73 == 93?
8
44
83
74
93


50
27
73
8
44
83
83 == 93?
74
93



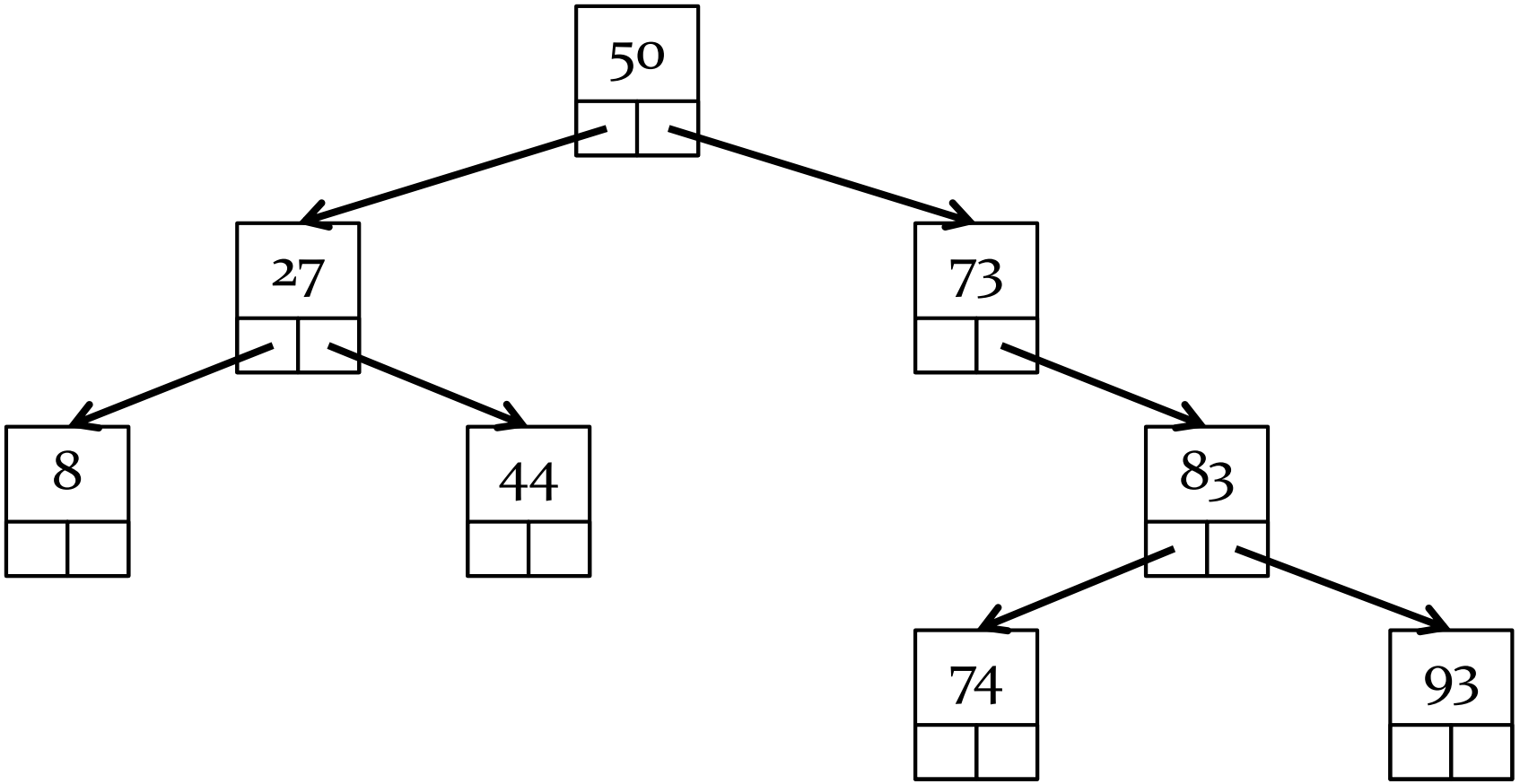
74 == 93?



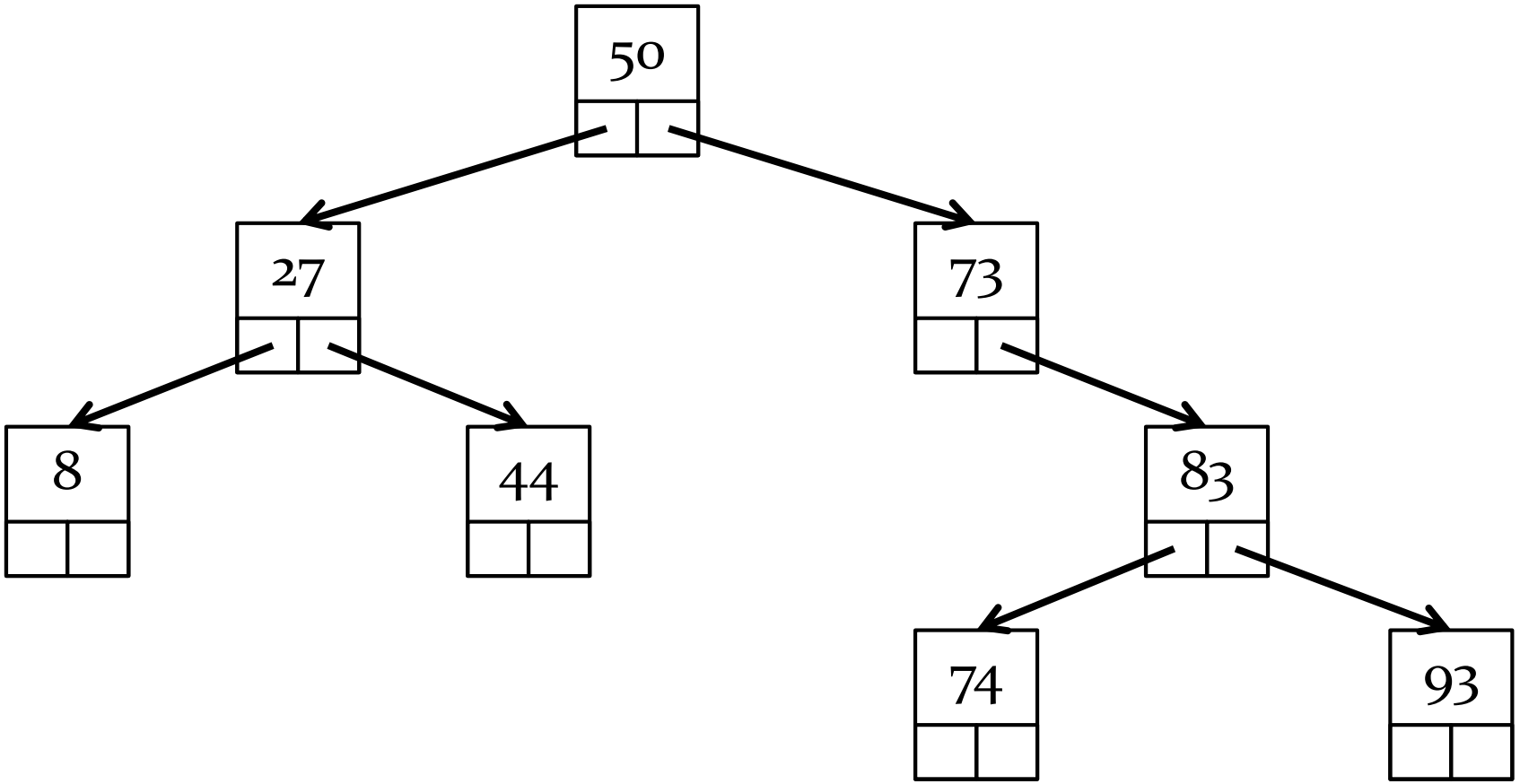
93 == 93?

Iteration

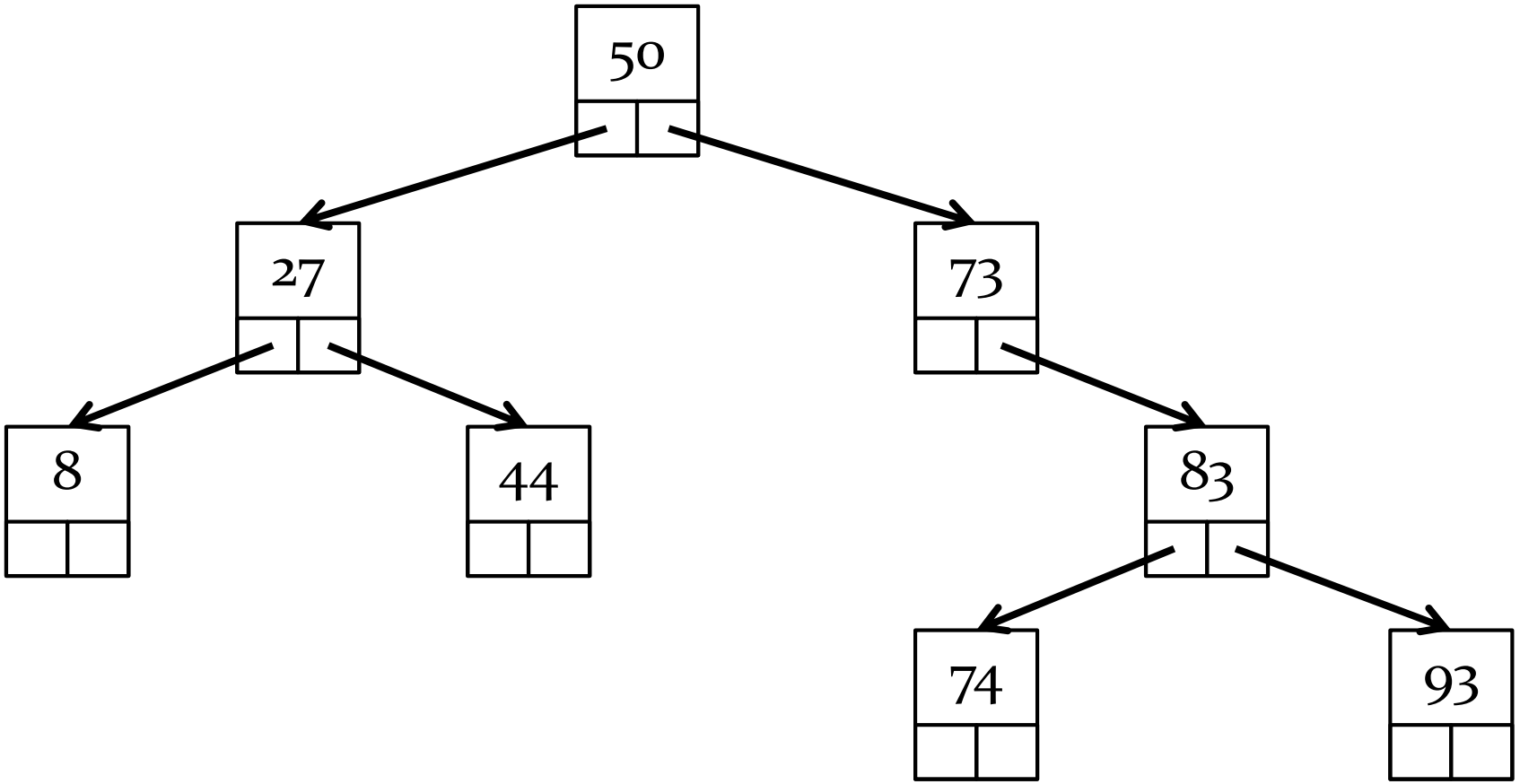
- visiting every element of the tree can also be done recursively
- 3 possibilities based on when the root is visited
  - inorder
    - visit left child, then root, then right child
  - preorder
    - visit root, then left child, then right child
  - postorder
    - visit left child, then right child, then root



inorder: 8, 27, 44, 50, 73, 74, 83, 93

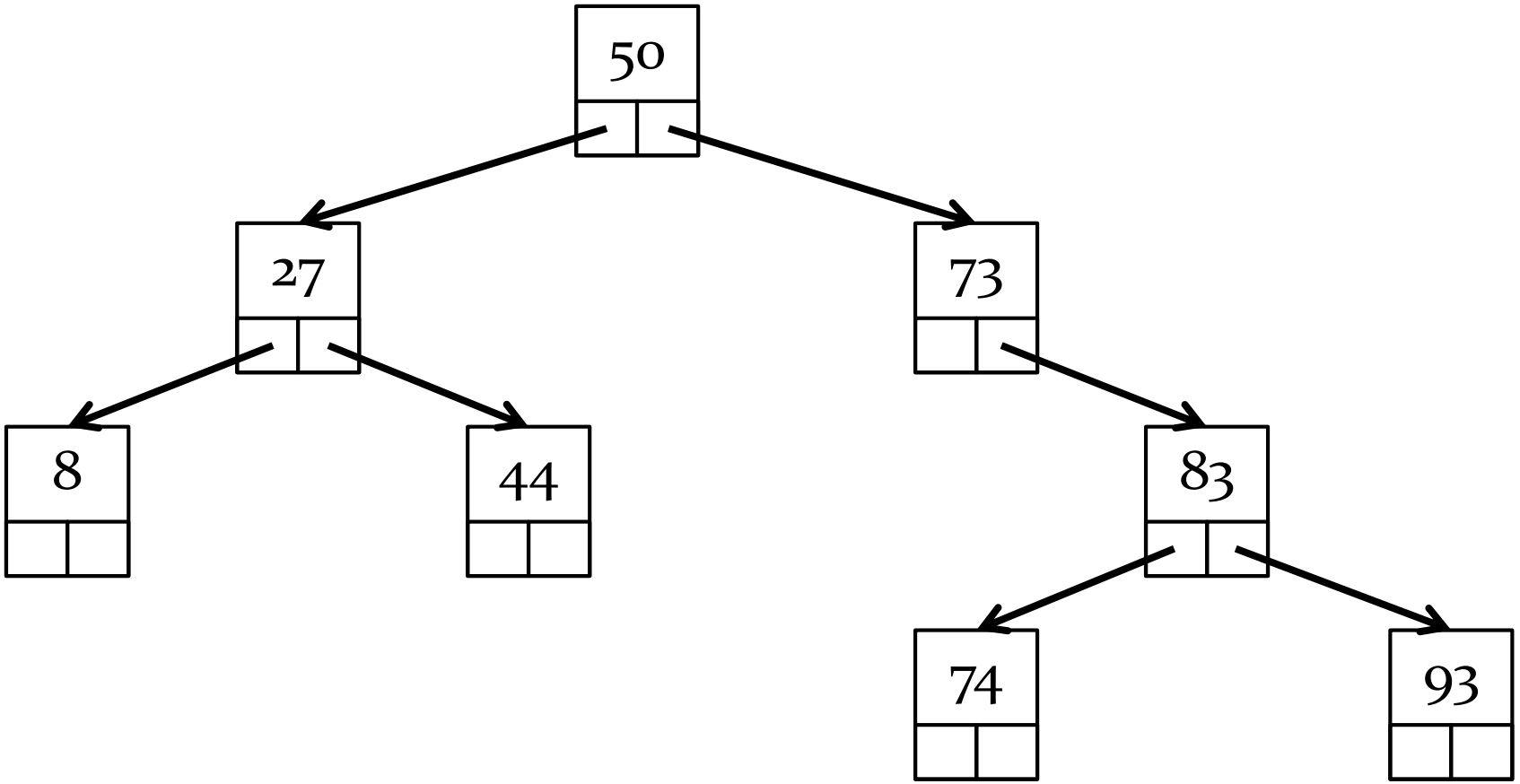


preorder: 50, 27, 8, 44, 73, 83, 74, 93



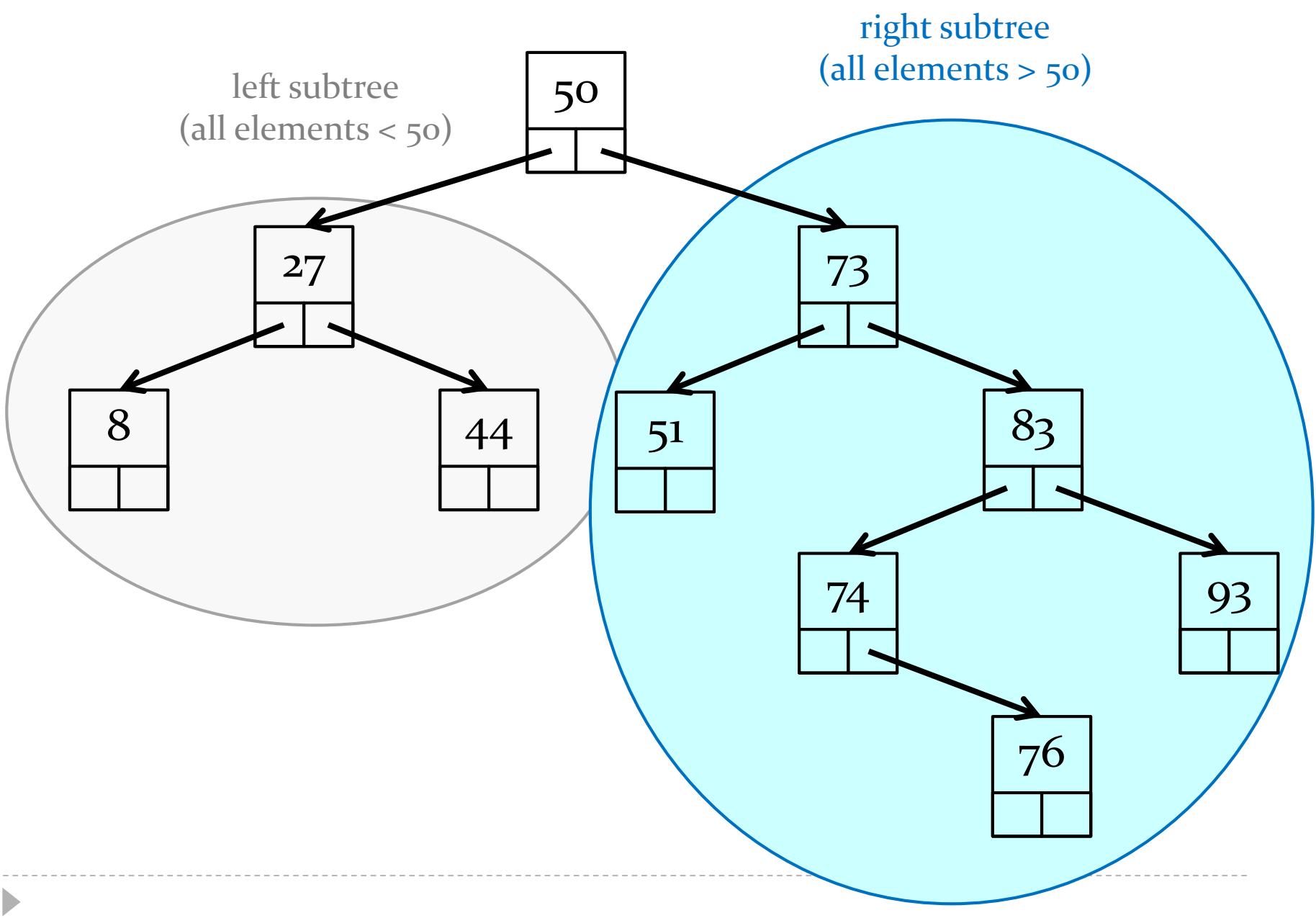
postorder: 8, 44, 27, 74, 93, 83, 73, 50

Binary Search Trees


50
27
73
8
44
83
74
93

Binary Search Trees (BST)

- the tree from the previous slide is a special kind of binary tree called a *binary search tree*
- in a binary search tree:
  1. all nodes in the left subtree have data elements that are less than the data element of the root node
  2. all nodes in the right subtree have data elements that are greater than the data element of the root node
  3. rules 1 and 2 apply recursively to every subtree


left subtree
(all elements < 50)
right subtree
(all elements > 50)
50
27
73
8
44
51
83
74
93
76

Implementing a BST

- what types of data elements can a BST hold?
  - hint: we need to be able to perform comparisons such as less than, greater than, and equal to with the data elements

```
public class BinarySearchTree<E extends Comparable<? super E>> {
```

E must implement **Comparable<G>** where **G** is either **E** or an ancestor of **E**

Implementing a BST: Nodes

- we need a node class that:
  - has-a data element
  - has-a link to the left subtree
  - has-a link to the right subtree

```
public class BinarySearchTree<E extends Comparable<? super E>> {

  private static class Node<E> {
    private E data;
    private Node<E> left;
    private Node<E> right;

    /**
     * Create a node with the given data element. The left and right child
     * nodes are set to null.
     *
     * @param data
     *            the element to store
     */
    public Node(E data) {
      this.data = data;
      this.left = null;
      this.right = null;
    }
  }
```

Implementing a BST: Fields and Ctor

- a BST has-a root node

- creating an empty BST should set the root node to null

```java
/**
 * The root node of the binary search tree.
 */
private Node<E> root;

/**
 * Create an empty binary search tree.
 */
public BinarySearchTree() {
  this.root = null;
}
```

Implementing a BST: Adding elements

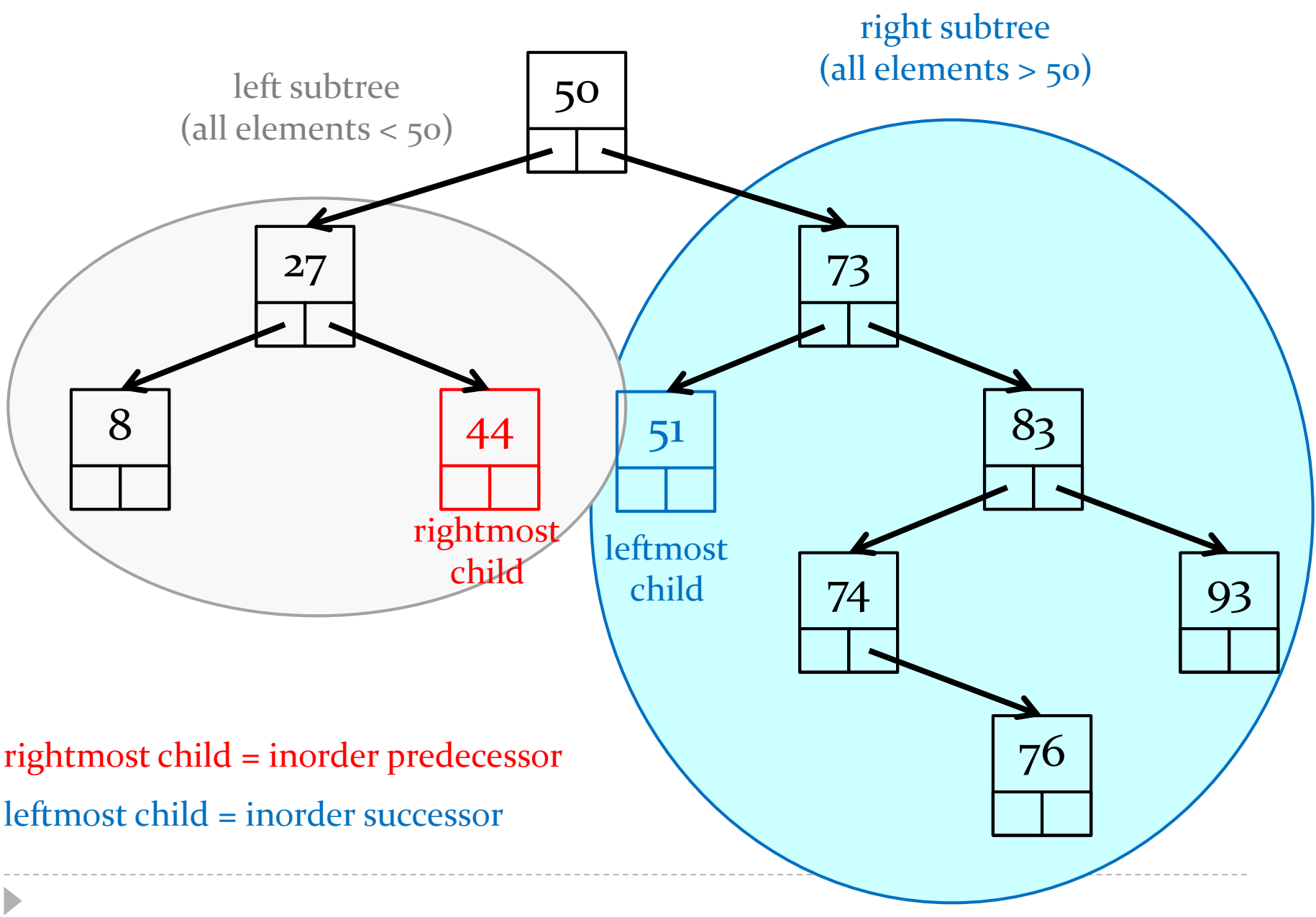
- the definition for a BST tells you everything that you need to know to add an element
- in a binary search tree:
  1. all nodes in the left subtree have data elements that are less than the data element of the root node
  2. all nodes in the right subtree have data elements that are greater than the data element of the root node
  3. rules 1 and 2 apply recursively to every subtree

```java
/**
 * Add an element to the tree. The element is inserted into the tree in a
 * position that preserves the definition of a binary search tree.
 *
 * @param element
 *            the element to add to the tree
 */
public void add(E element) {
  if (this.root == null) {
    this.root = new Node<E>(element);
  }
  else {
    BinarySearchTree.add(element, null, this.root);  // recursive static method
  }
}
```

```
/**
 * Add an element to the subtree rooted at <code>root</code>. The element is inserted into the tree in a
 * position that preserves the definition of a binary search tree.
 *
 * @param element        the element to add to the subtree
 * @param parent         the parent node to the subtree
 * @param root           the root of the subtree
 */
private static <E extends Comparable<? super E>> void add(E element, Node<E> parent, Node<E> root) {
  if (root == null && element.compareTo(parent.data) < 0) {
    parent.left = new Node<E>(element);
  }
  else if (root == null) {
    parent.right = new Node<E>(element);
  }
  else if (element.compareTo(root.data) < 0) {
    BinarySearchTree.add(element, root, root.left);
  }
  else {
    BinarySearchTree.add(element, root, root.right);
  }
}
```

Predecessors and Successors in a BST

- in a BST there is something special about a node's:
  - left subtree right-most child
  - right subtree left-most child



rightmost child = inorder predecessor

leftmost child = inorder successor

Predecessors and Successors in a BST

- in a BST there is something special about a node's:
  - left subtree right-most child = inorder predecessor
    - the node containing the largest value *less* than the root
  - right subtree left-most child = inorder successor
    - the node containing the smallest value *greater* than the root

- it is easy to find the predecessor and successor nodes if you can find the nodes containing the maximum and minimum elements in a subtree

```java
/**
 * Find the node in a subtree that has the smallest data element.
 *
 * @param subtreeRoot
 *            the root of the subtree
 * @return the node in the subtree that has the smallest data element.
 */
public static <E> Node<E> minimumInSubtree(Node<E> subtreeRoot) {
  if (subtreeRoot.left() == null) {
    return subtreeRoot;
  }
  return BinarySearchTree.minimumInSubtree(subtreeRoot.left);
}
```

```java
/**
 * Find the node in a subtree that has the largest data element.
 *
 * @param subtreeRoot
 *            the root of the subtree
 * @return the node in the subtree that has the largest data element.
 */
public static <E> Node<E> maximumInSubtree(Node<E> subtreeRoot) {
  if (subtreeRoot.right() == null) {
    return subtreeRoot;
  }
  return BinarySearchTree.maximumInSubtree(subtreeRoot.right);
}
```

```java
/**
 * Find the node in a subtree that is the predecessor to the root of the
 * subtree. If the predecessor node exists, then it is the node that has the
 * largest data element in the left subtree of <code>subtreeRoot</code>.
 *
 * @param subtreeRoot
 *          the root of the subtree
 * @return the node in a subtree that is the predecessor to the root of the
 *         subtree, or <code>null</code> if the root of the subtree has no
 *         predecessor
 */
public static <E> Node<E> predecessorInSubtree(Node<E> subtreeRoot) {
  if (subtreeRoot.left() == null) {
    return null;
  }
  return BinarySearchTree.maximumInSubtree(subtreeRoot.left);
}
```

```java
/**
 * Find the node in a subtree that is the successor to the root of the
 * subtree. If the successor node exists, then it is the node that has the
 * smallest data element in the right subtree of <code>subtreeRoot</code>.
 *
 * @param subtreeRoot
 *          the root of the subtree
 * @return the node in a subtree that is the successor to the root of the
 *         subtree, or <code>null</code> if the root of the subtree has no
 *         successor
 */
public static <E> Node<E> successorInSubtree(Node<E> subtreeRoot) {
  if (subtreeRoot.right() == null) {
    return null;
  }
  return BinarySearchTree.minimumInSubtree(subtreeRoot.right);
}
```

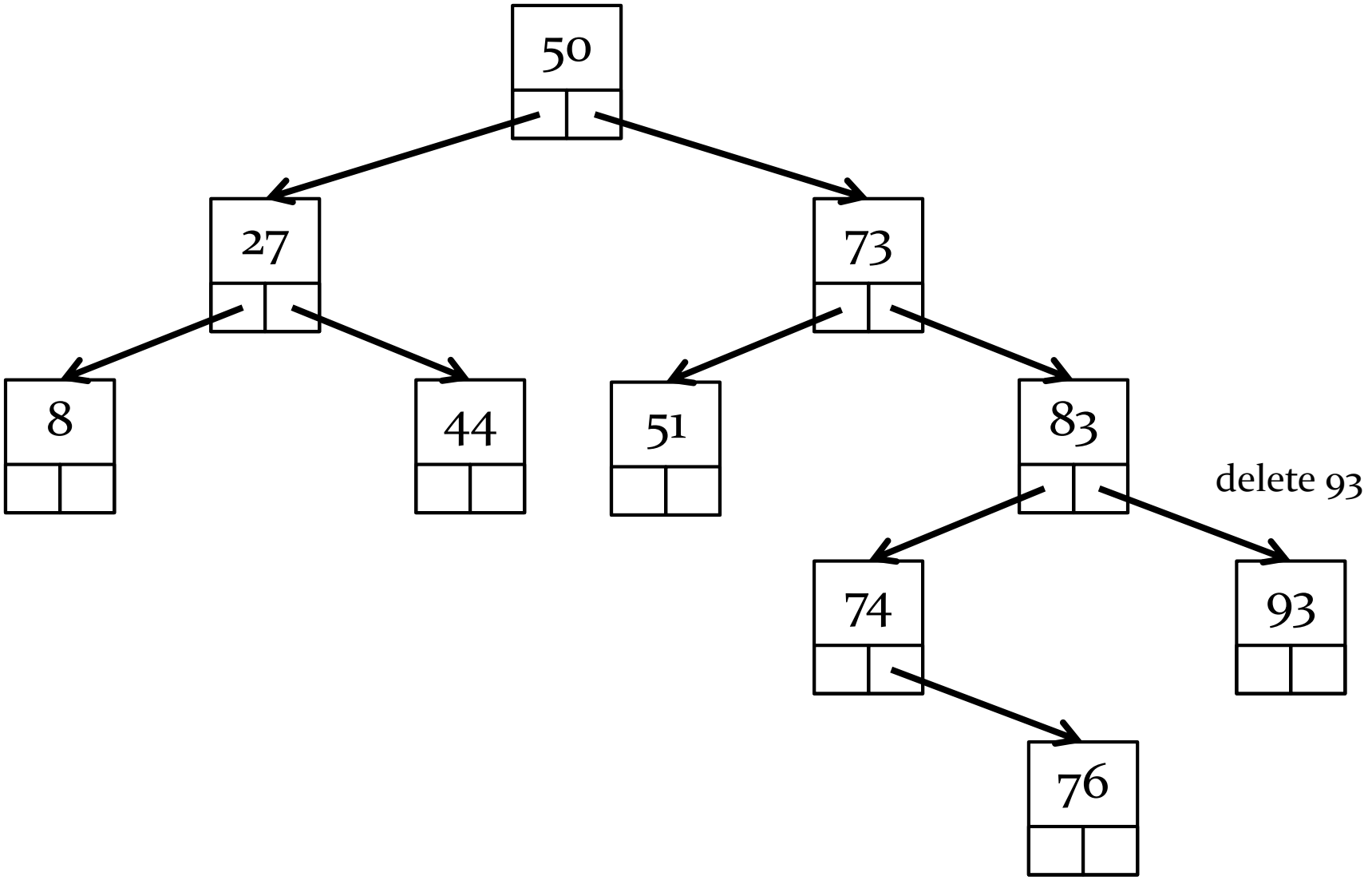
Deletion from a BST

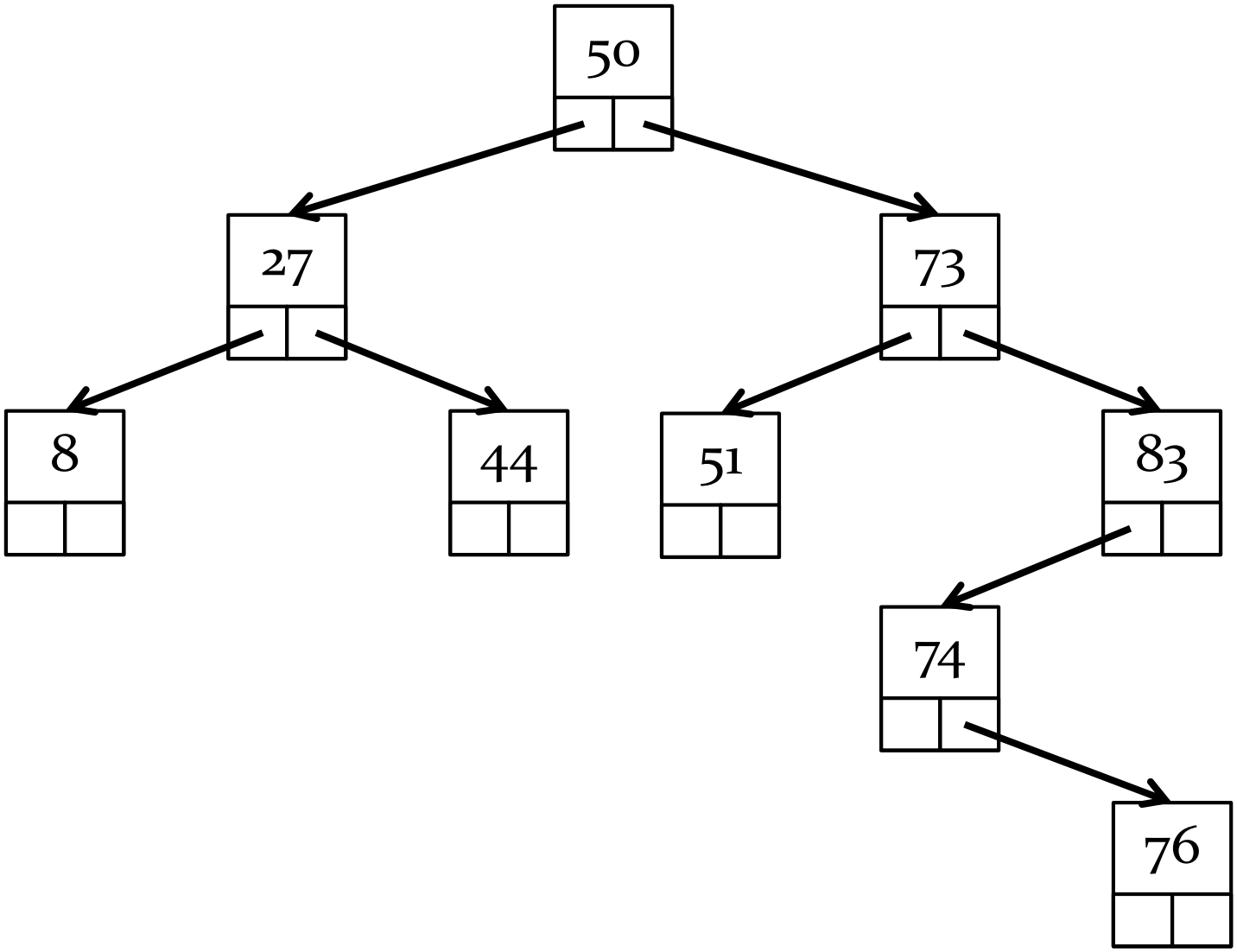
- to delete a node in a BST there are 3 cases to consider:
  1. deleting a leaf node
  2. deleting a node with one child
  3. deleting a node with two children

Deleting a Leaf Node

- deleting a leaf node is easy because the leaf has no children
  - simply remove the node from the tree

- e.g., delete 93

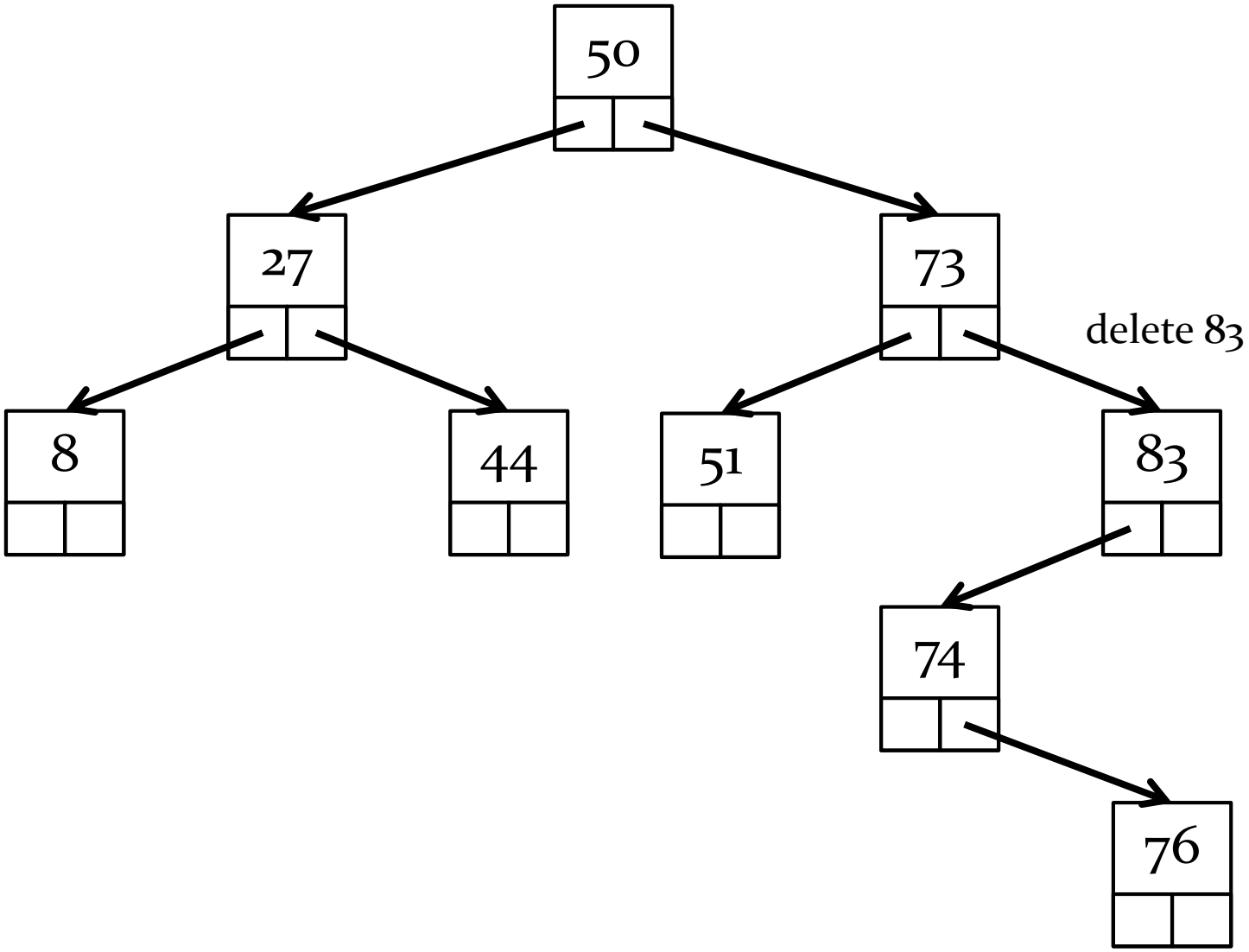

50
27
73
8
44
51
83
delete 93
74
93
76

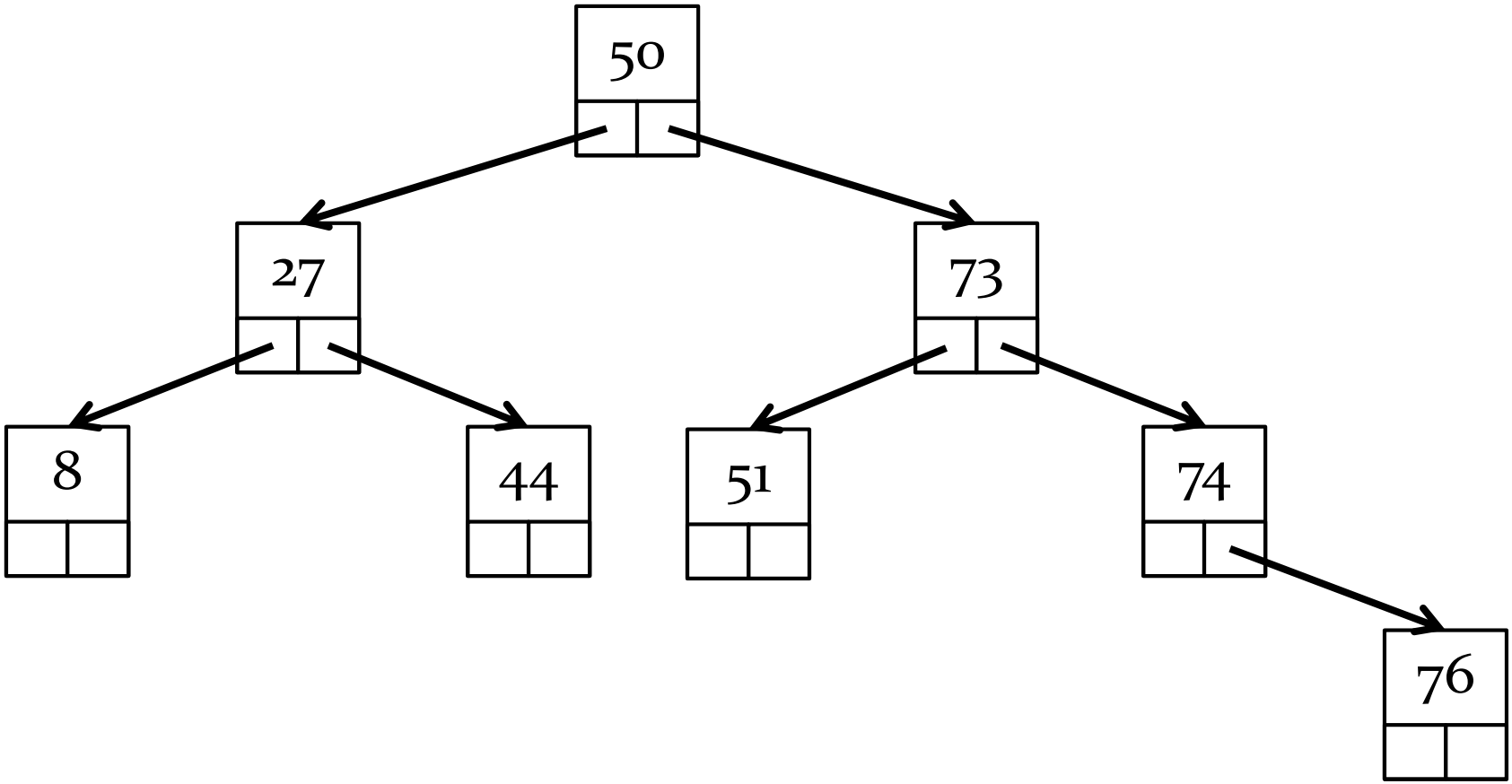

50
27
73
8
44
51
83
74
76

Deleting a Node with One Child

- deleting a node with one child is also easy because of the structure of the BST
  - remove the node by replacing it with its child

- e.g., delete 83


50
27
73
delete 83
8
44
51
83
74
76


50
27
73
8
44
51
74
76

Deleting a Node with Two Children

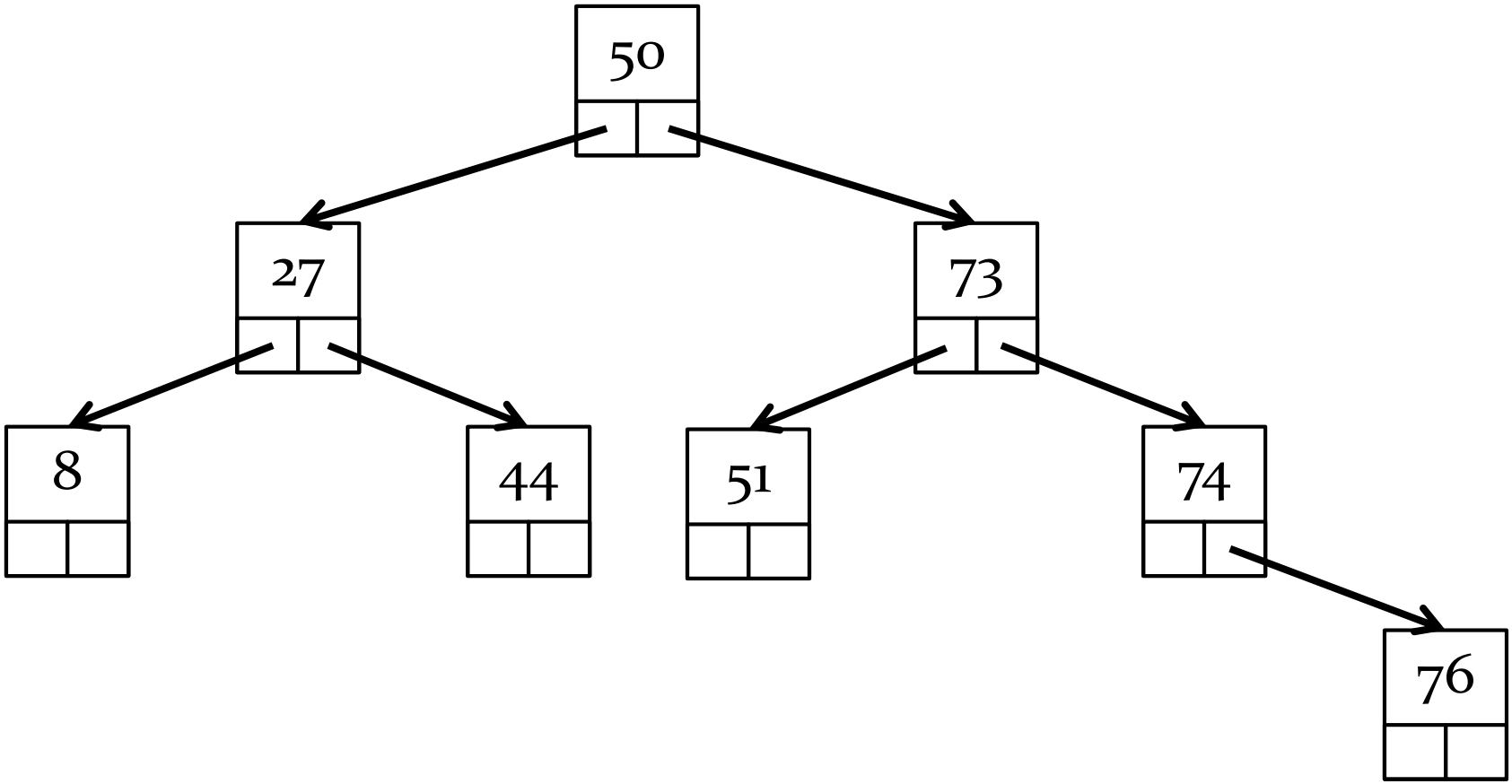
- deleting a node with two children is a little trickier
  - can you see how to do it?

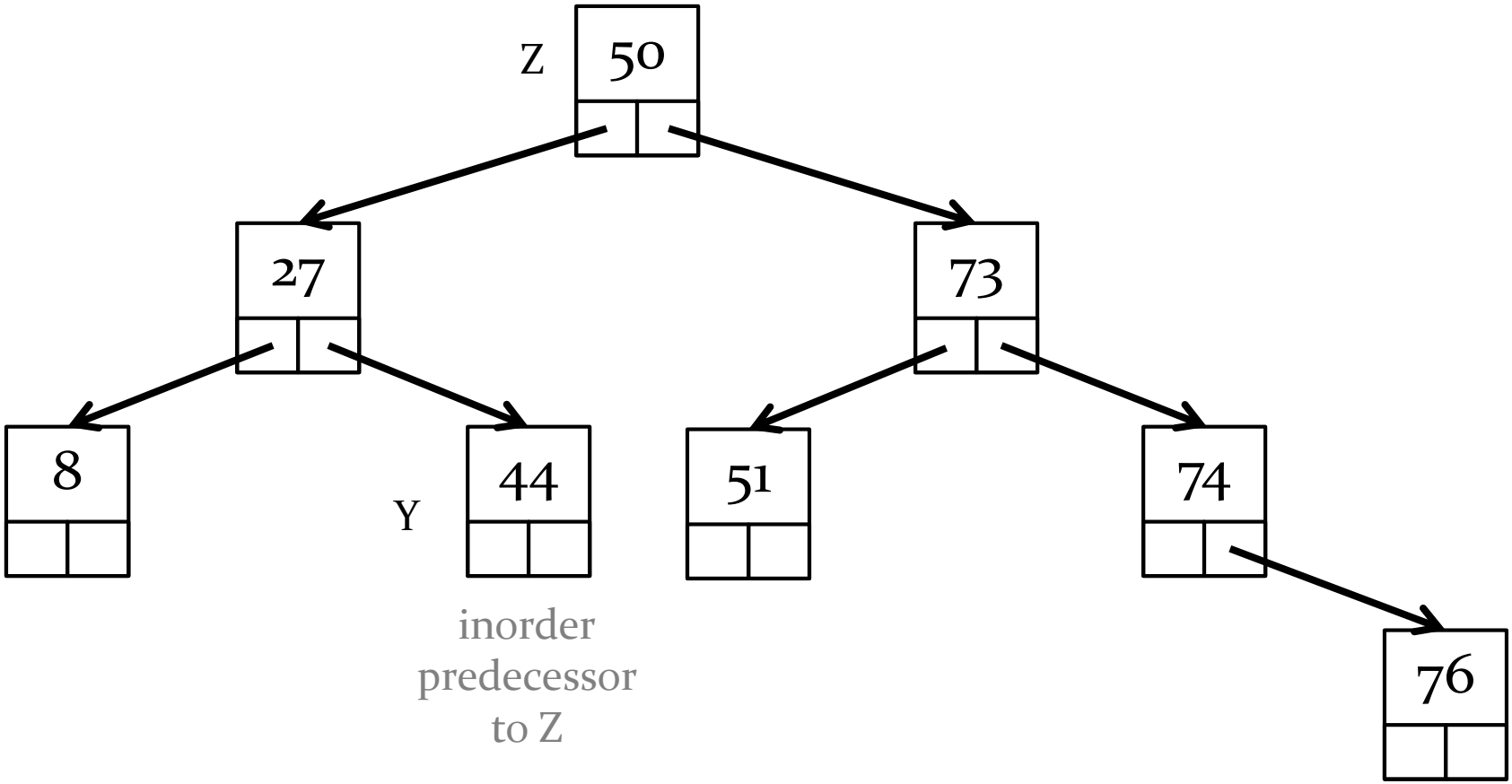
Deleting a Node with Two Children

- replace the node with its inorder predecessor OR inorder successor
  - call the node to be deleted Z
  - find the inorder predecessor OR the inorder successor
    - call this node Y
  - copy the data element of Y into the data element of Z
  - delete Y

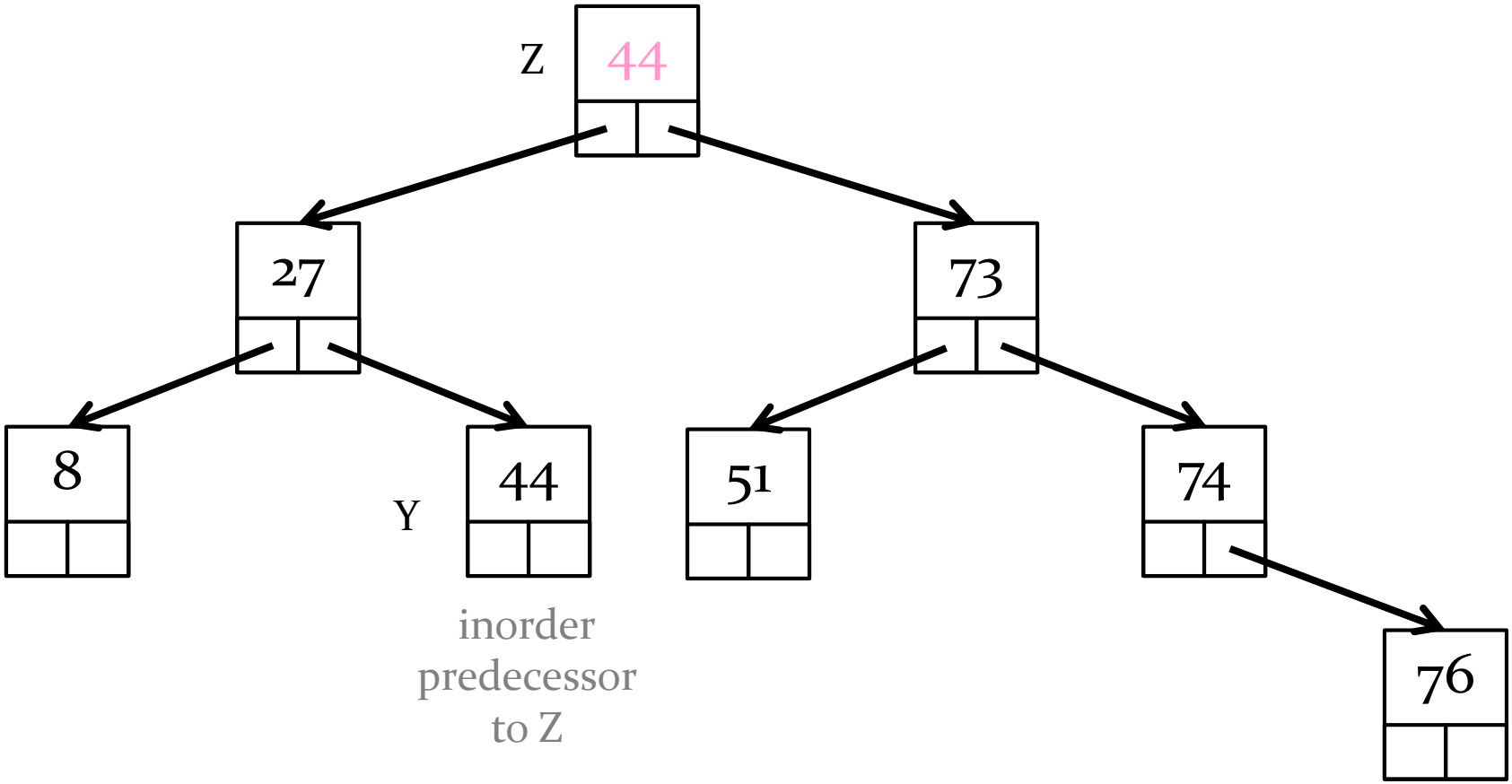
- e.g., delete 50

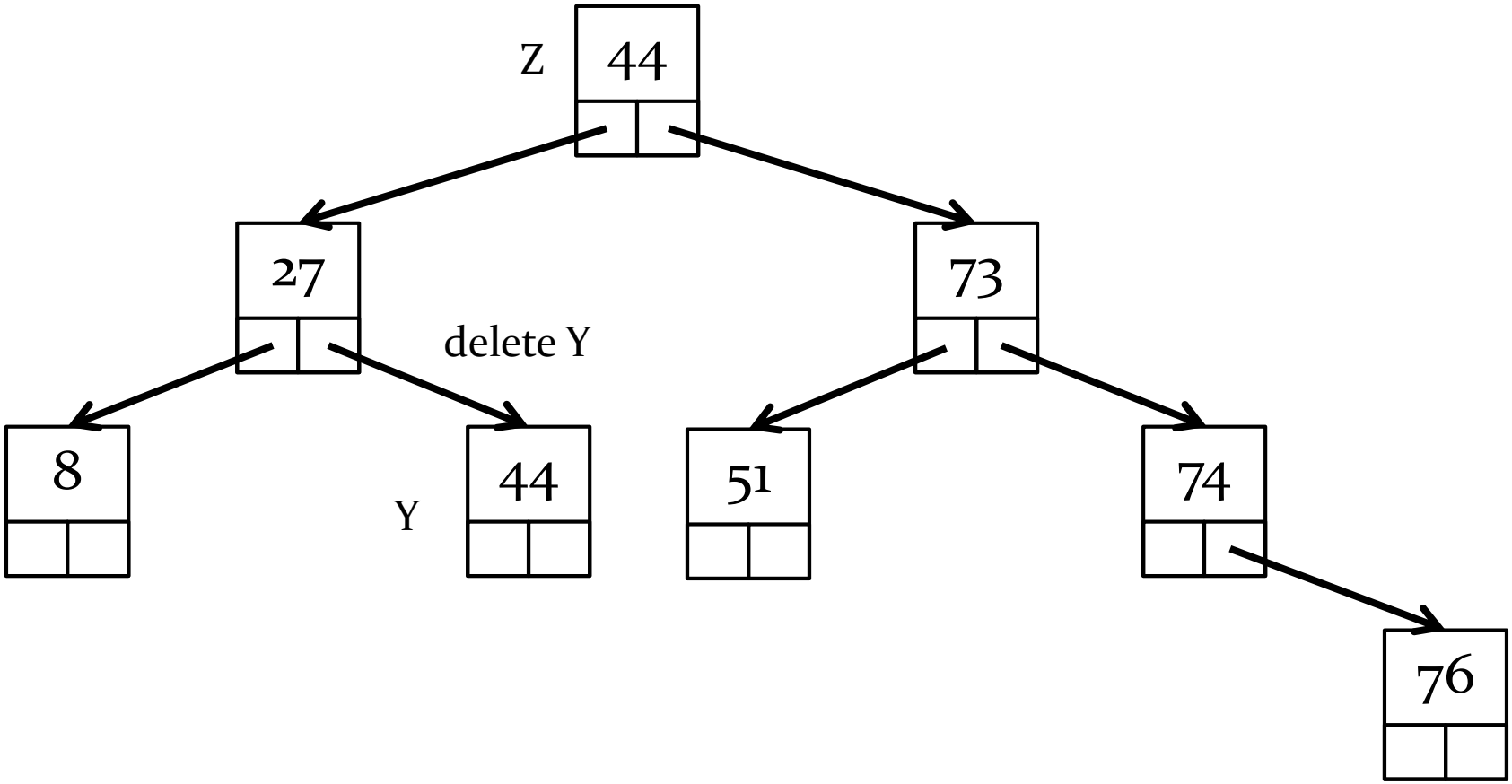
delete 50 using inorder predecessor

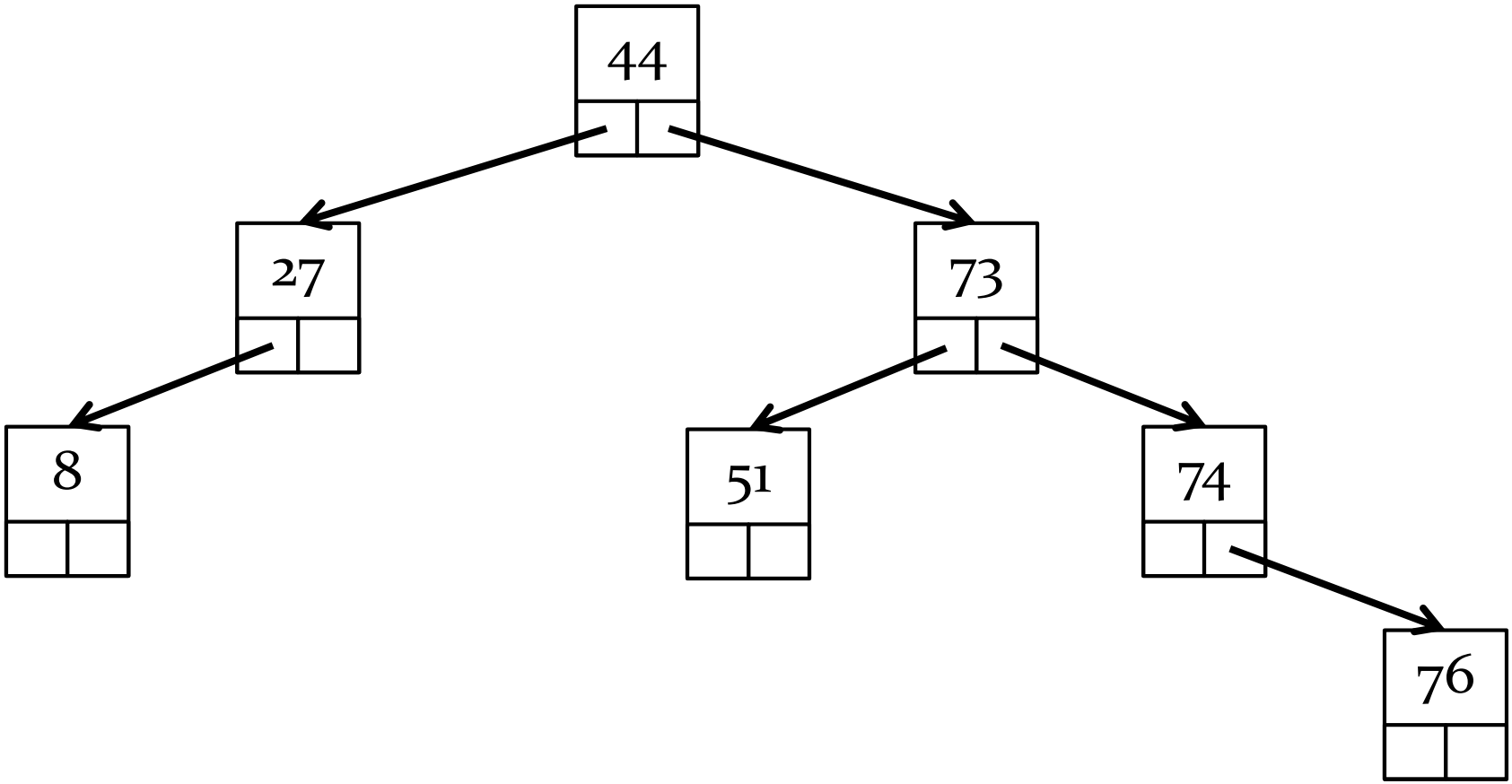



Z
50
27
73
8
Y
44
51
74
76
inorder
predecessor
to Z

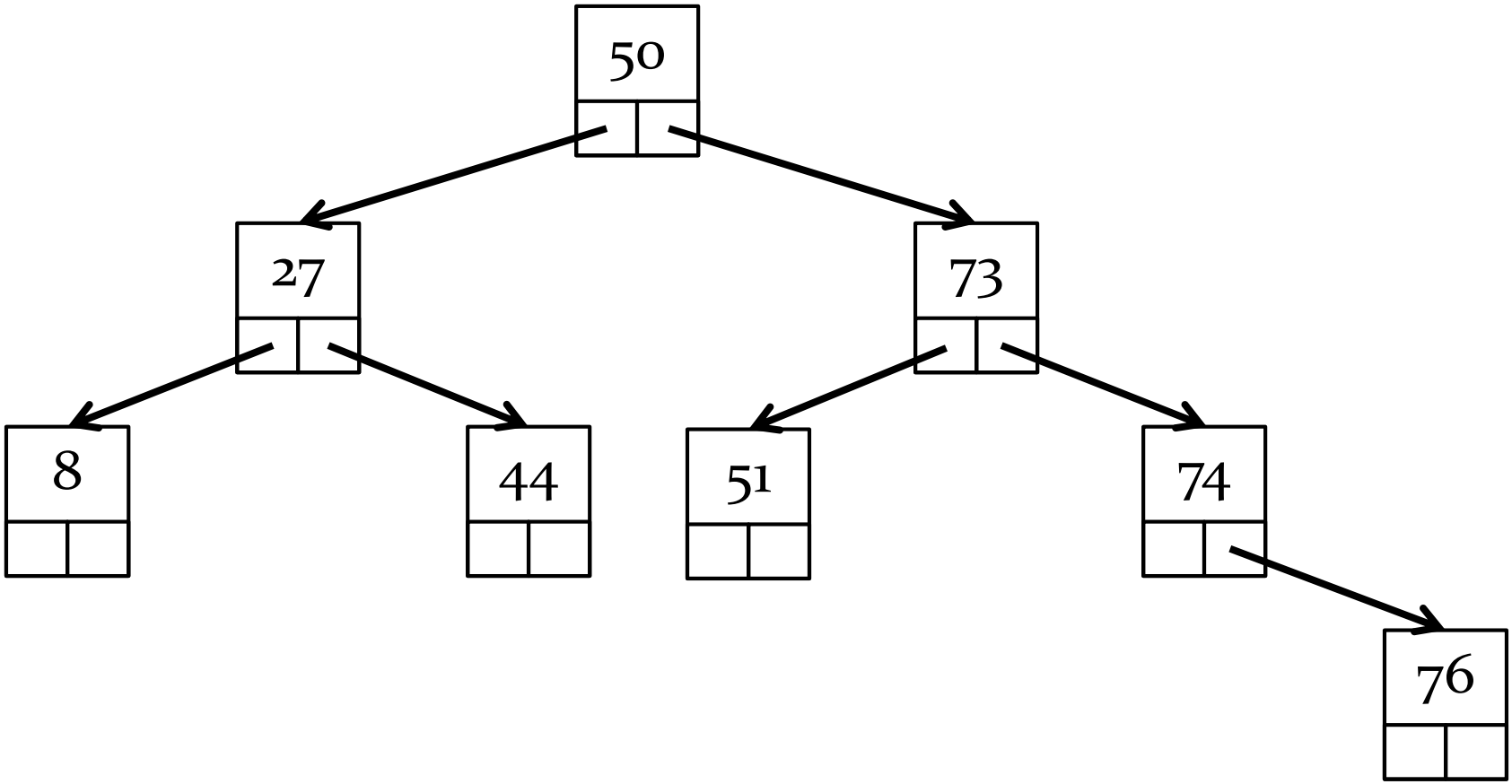
copy Y data to Z data

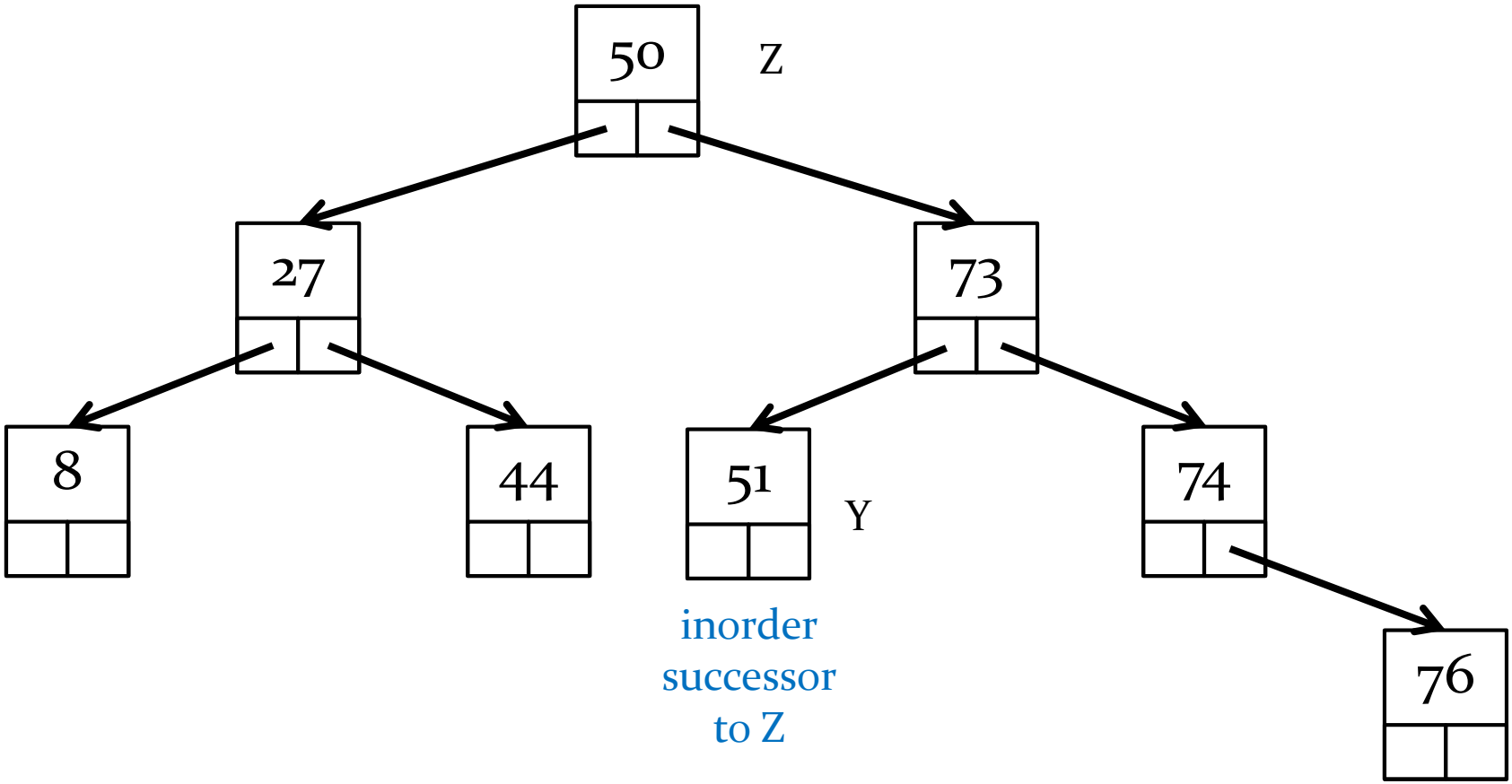


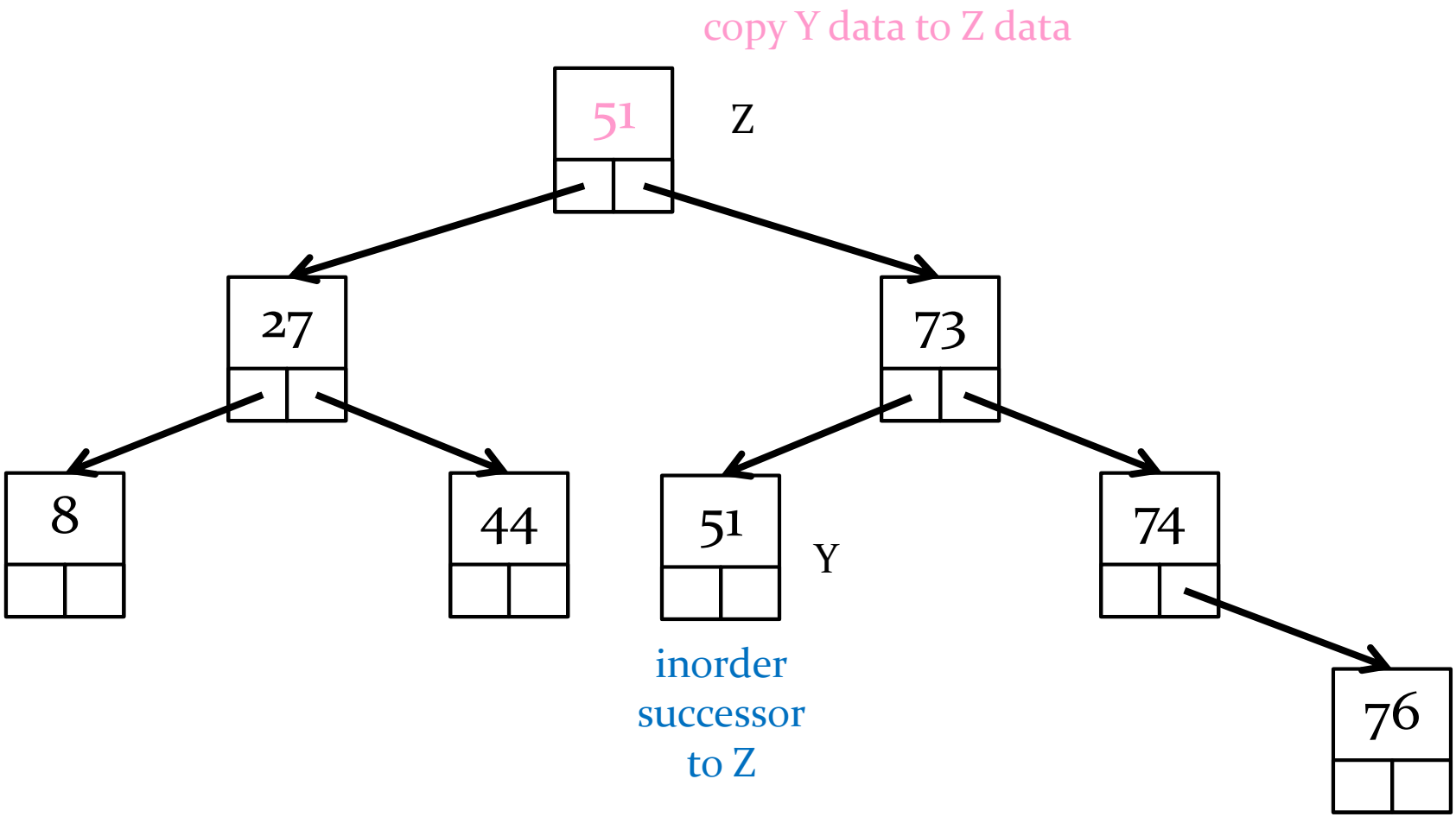

Z
44
27
delete Y
73
8
Y
44
51
74
76

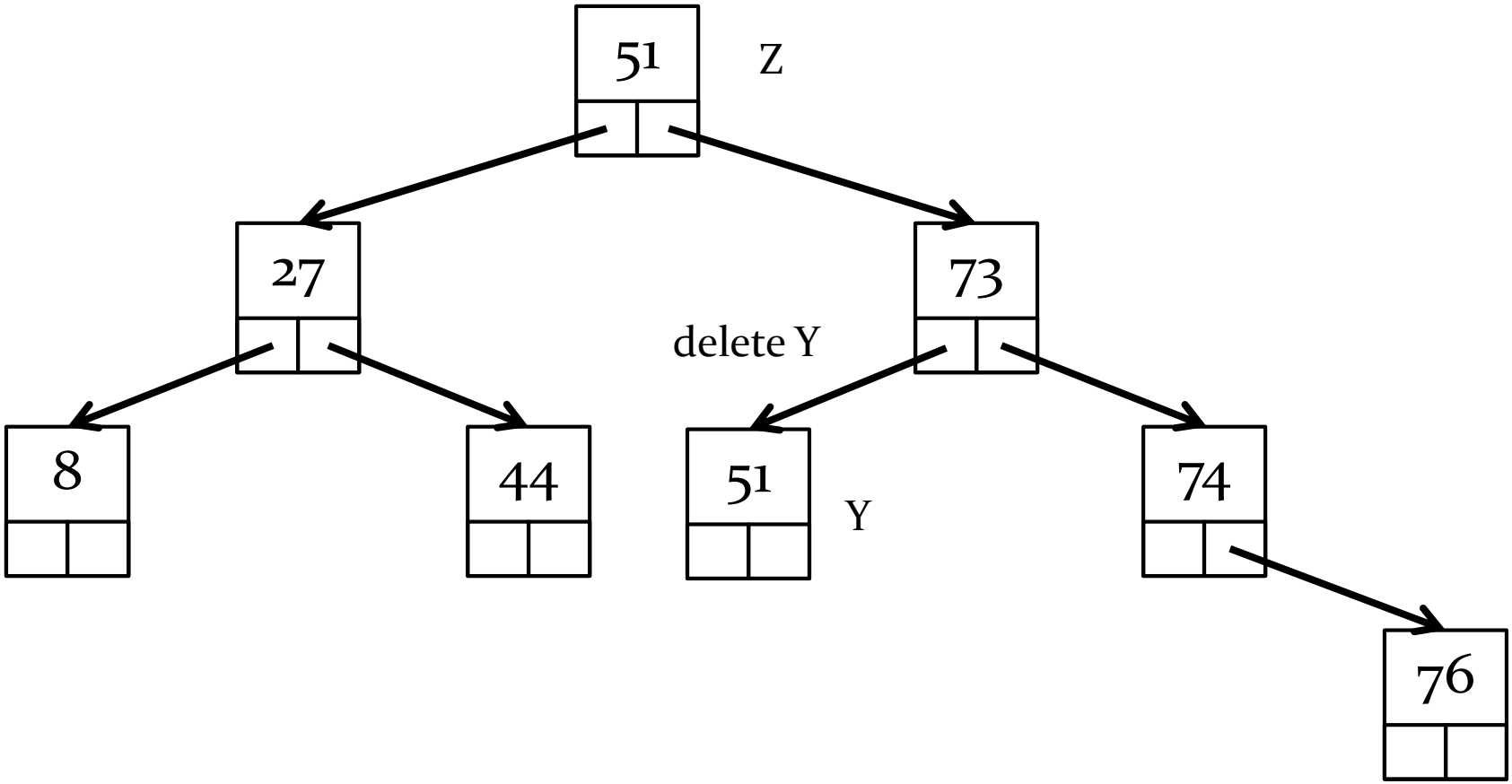

44
27
73
8
51
74
76

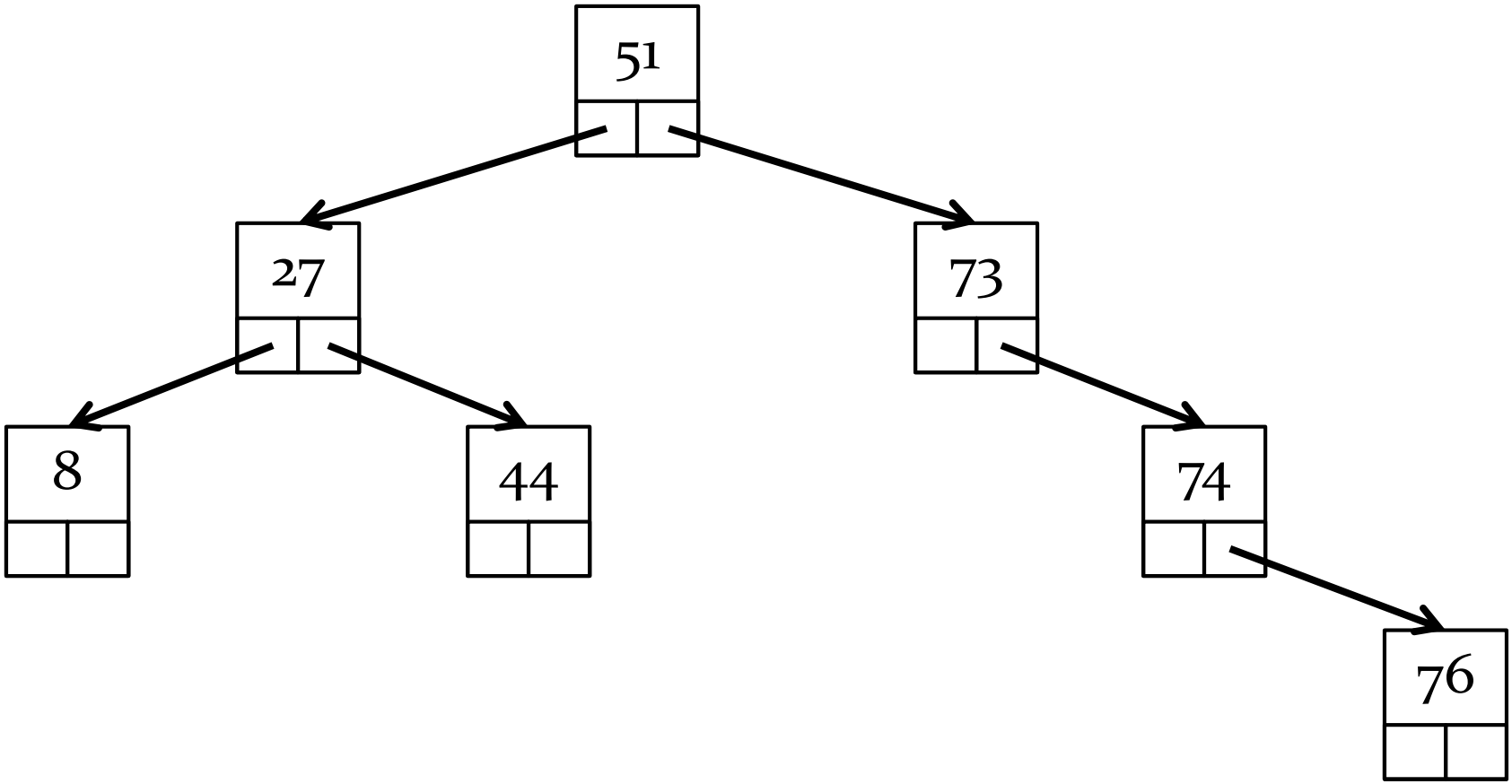
delete 50 using inorder successor




50
Z
27
73
8
44
51
Y
74
76
inorder
successor
to Z


copy Y data to Z data
51
Z
27
73
8
44
51
Y
74
inorder
successor
to Z
76


51
Z
27
73
delete Y
8
44
51
Y
74
76

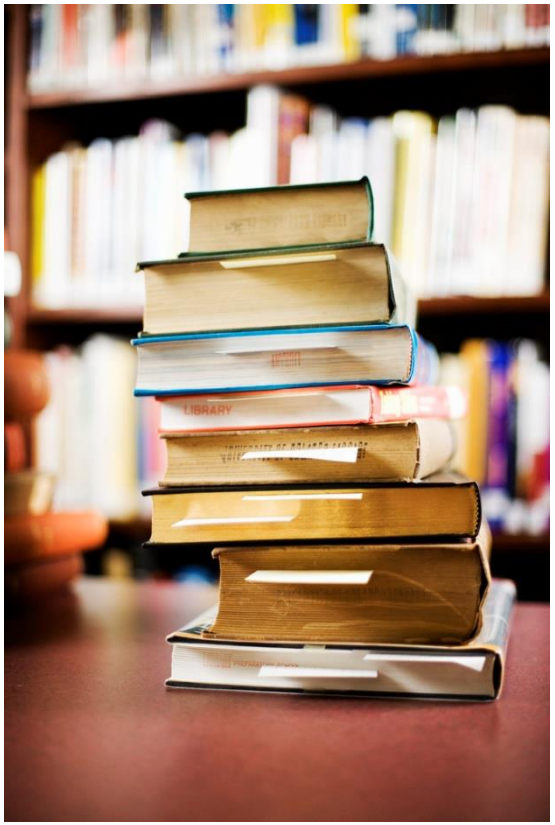

51
27
73
8
44
74
76

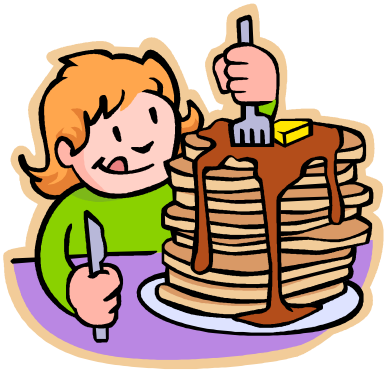
More Data Structures (Part 1)

Stacks

Stack

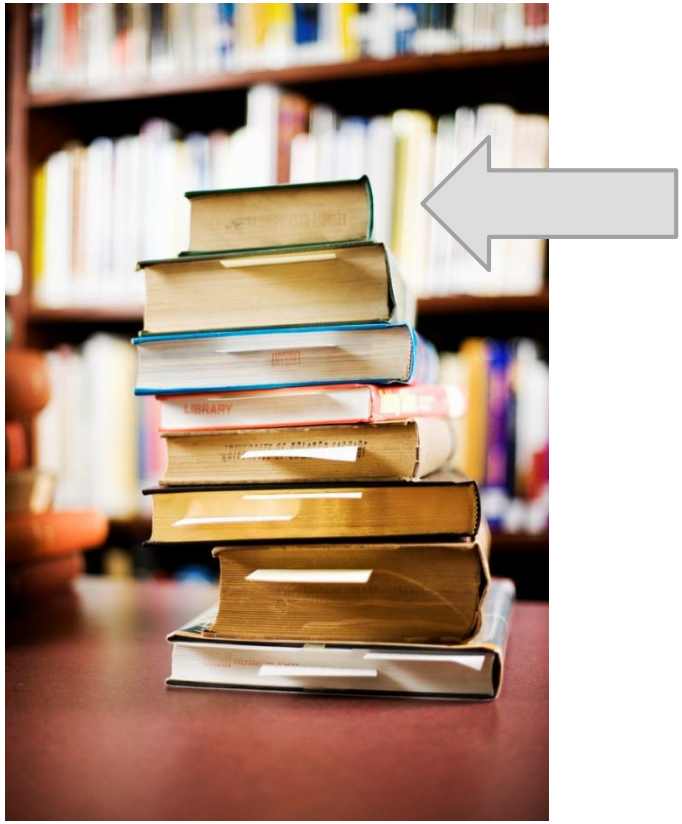
- examples of stacks





Top of Stack

- top of the stack

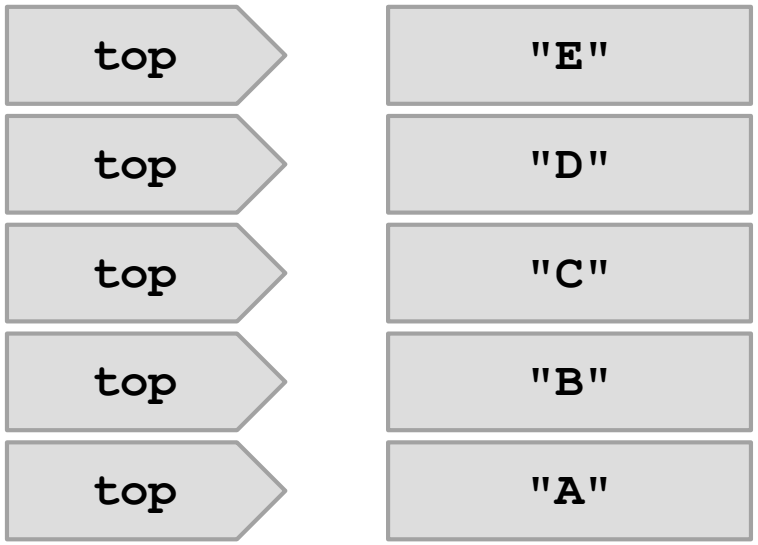


Stack Operations

- classically, stacks only support two operations
  1. push
     - add to the top of the stack
  2. pop
     - remove from the top of the stack
     - throws an exception if there is nothing on the stack

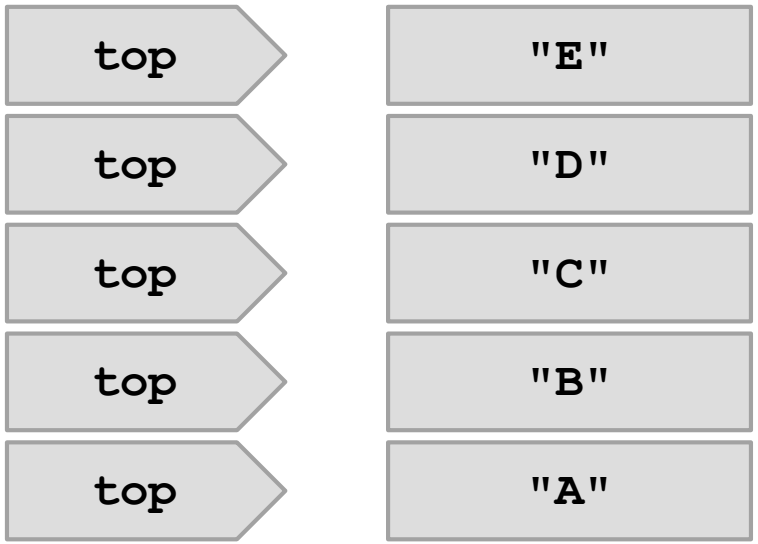
Push

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()`



Applications

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

Example: Reversing a sequence

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

Don't do this

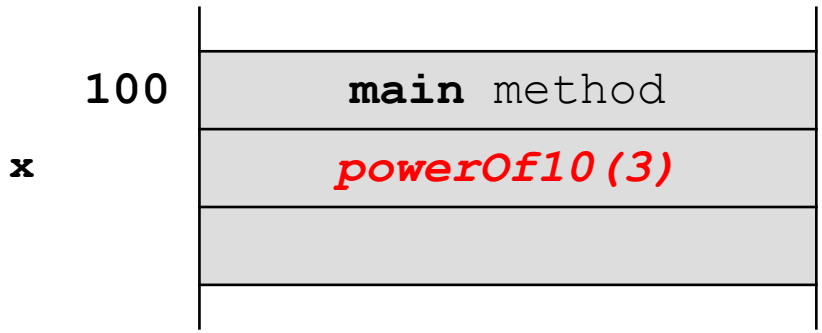
```
public static <E> List<E> reverse(List<E> t) {
  List<E> result = new ArrayList<E>();
  Stack<E> st = new Stack<E>();
  for (E e : t) {
    st.push(e);
  }
  while (!st.isEmpty()) {
    result.add(st.pop());
  }
  return result;
}
```

Converting a Recursive Method

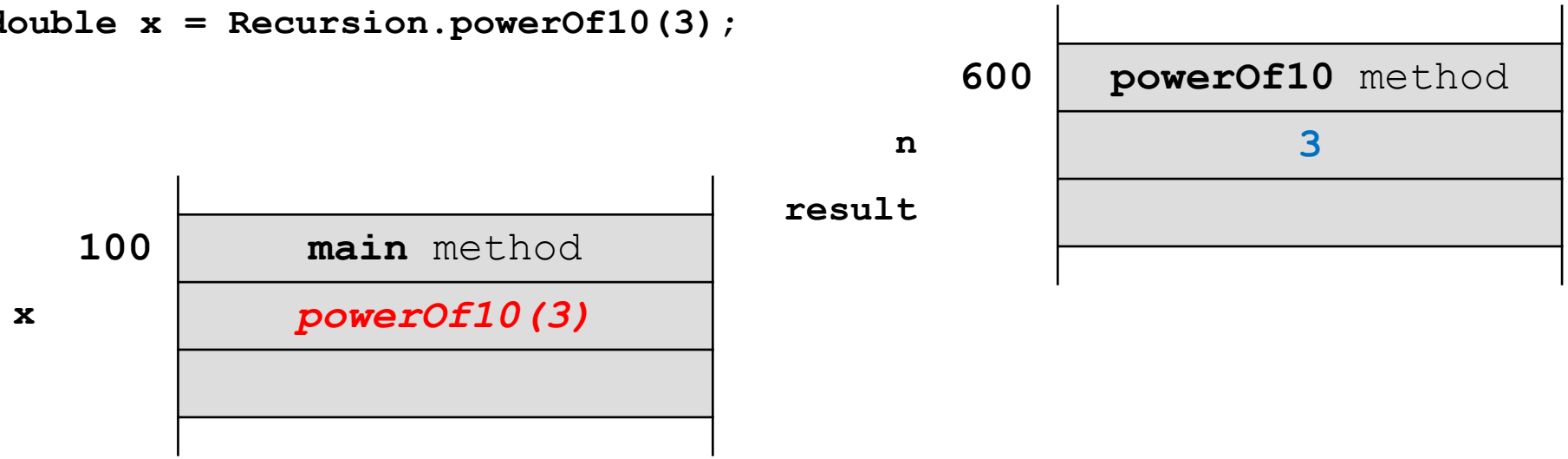
- a stack can be used to convert a recursive method to a non-recursive method
- the key to understanding how to do this lies in the memory diagram for a recursive memory
  - the following slides are from the Day 25 lecture

```
public static double powerOf10(int n) {
  double result;
  if (n < 0) {
    result = 1.0 / powerOf10(-n);
  }
  else if (n == 0) {
    result = 1.0;
  }
  else {
    result = 10 * powerOf10(n - 1);
  }
  return result;
}
```

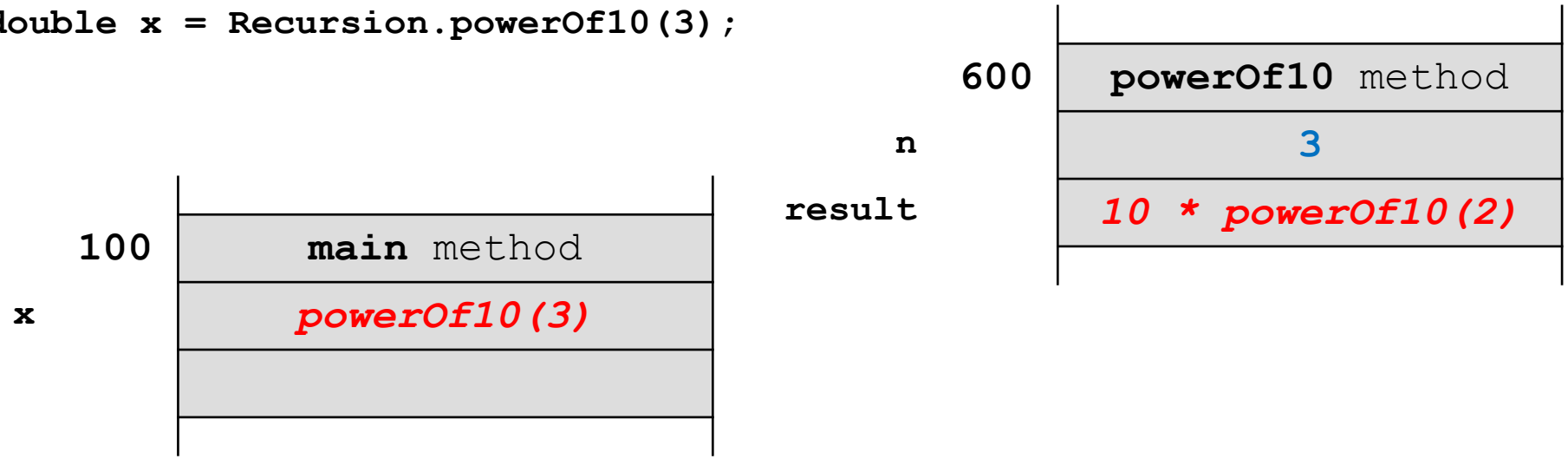
```
double x = Recursion.powerOf10(3);
```



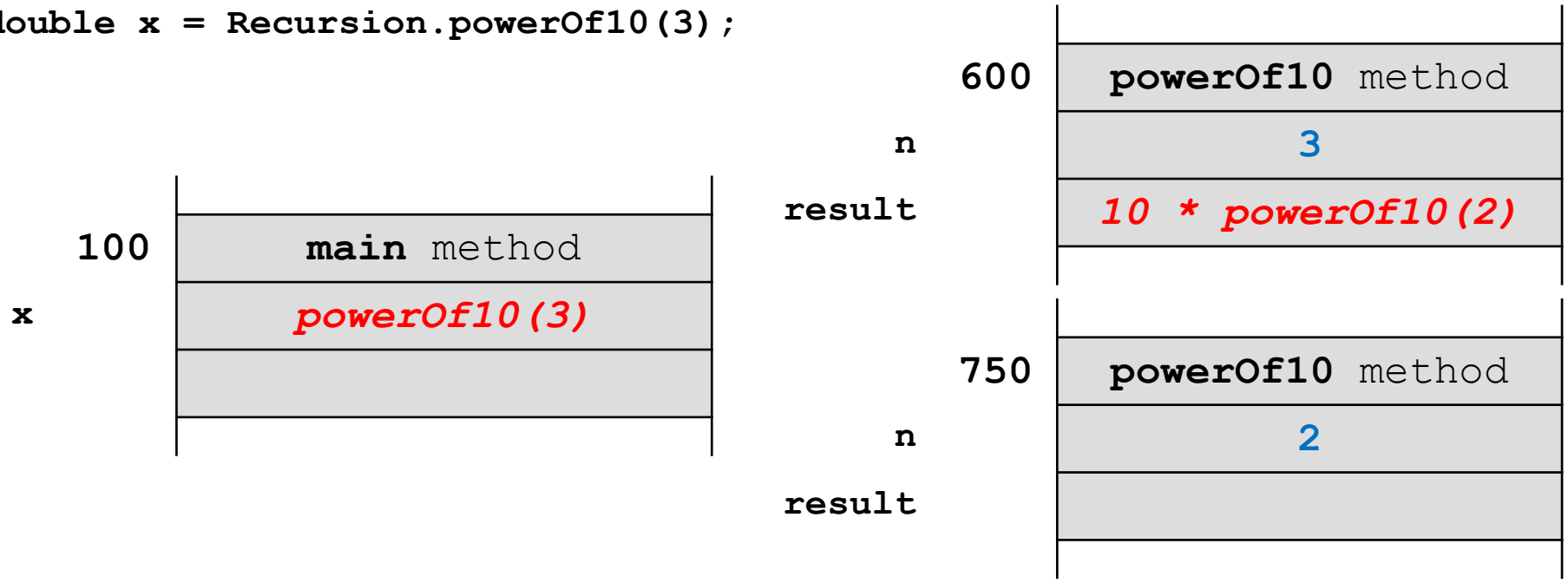
```
double x = Recursion.powerOf10(3);
```



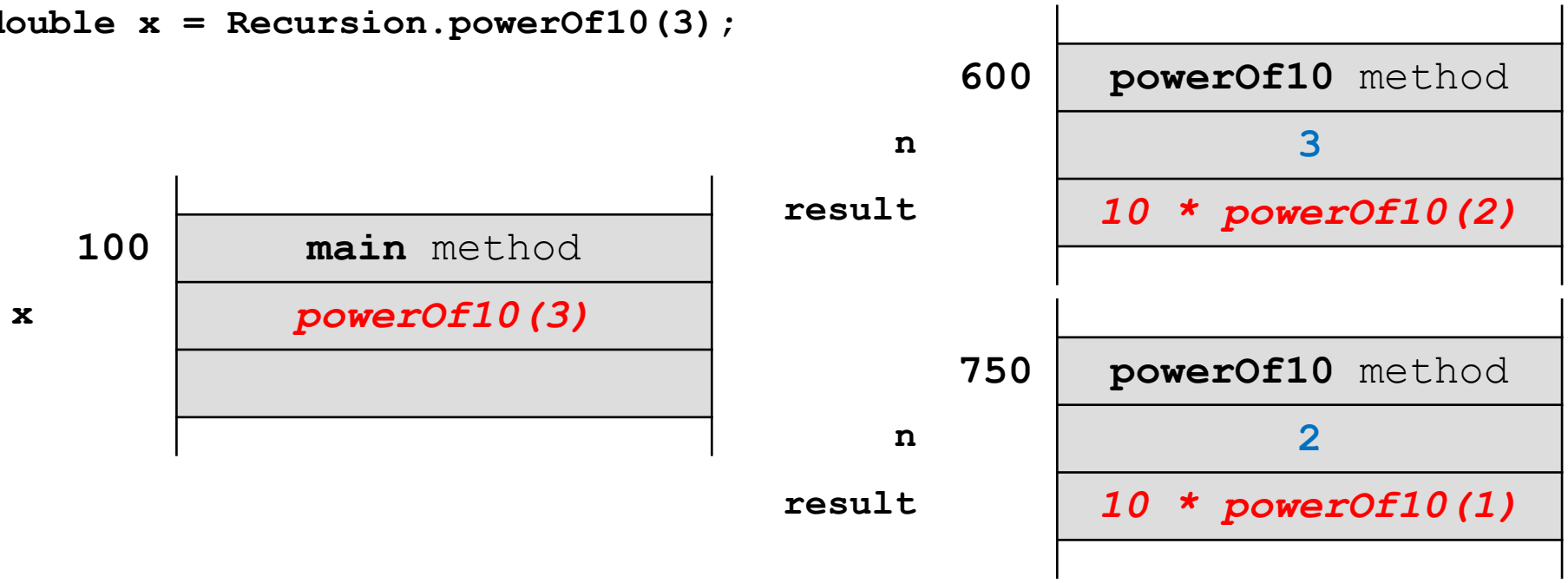
```
double x = Recursion.powerOf10(3);
```



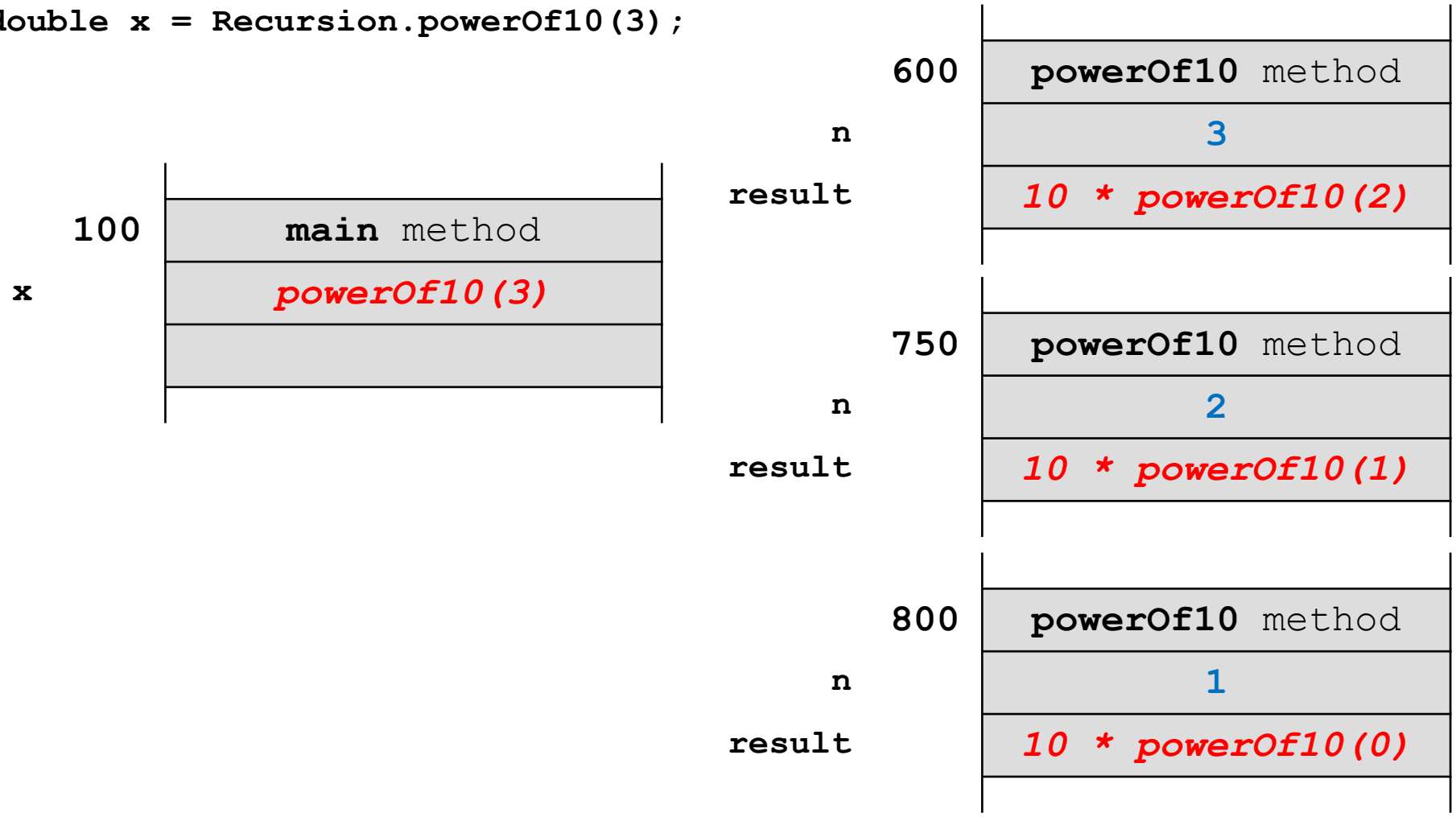
```
double x = Recursion.powerOf10(3);
```



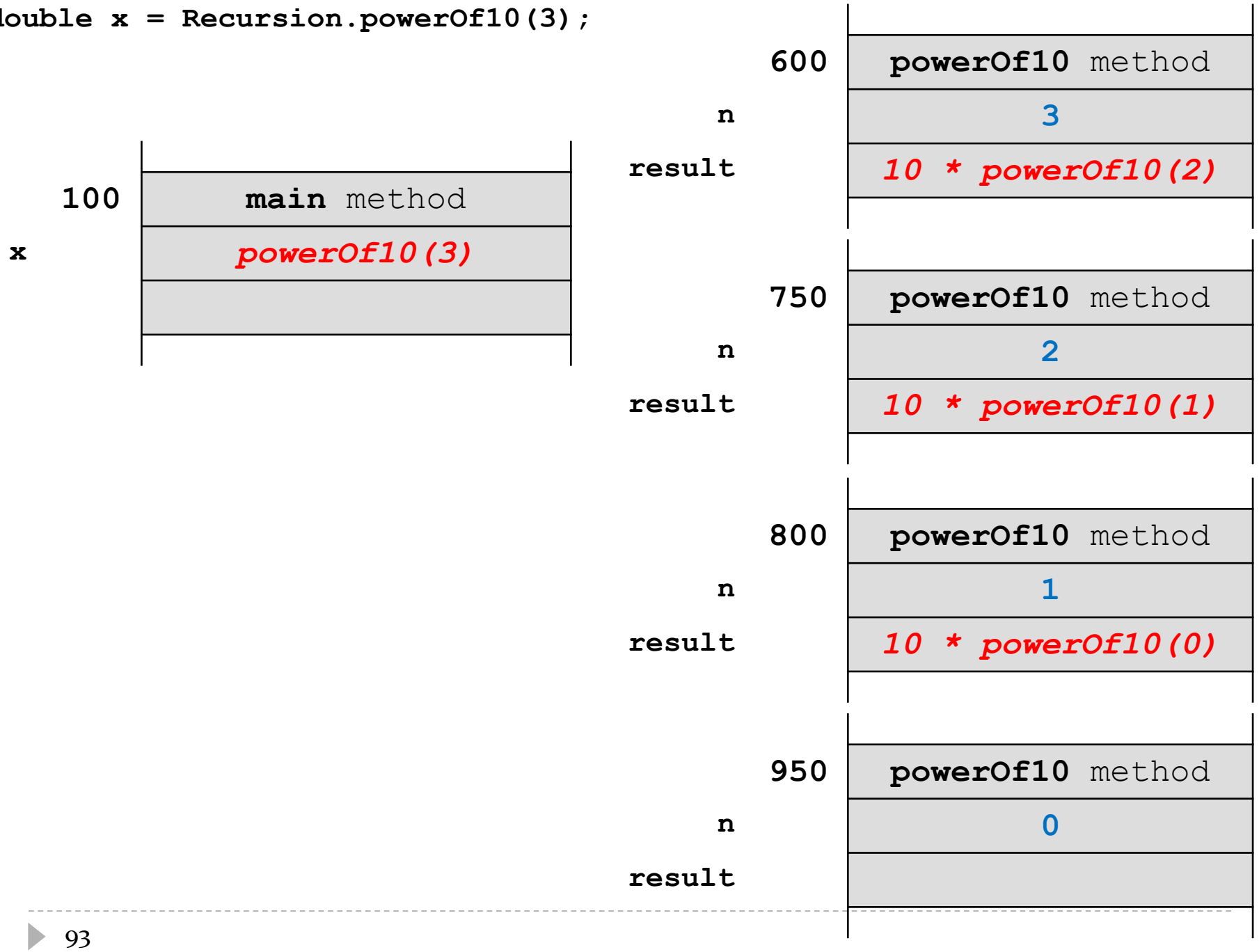
```
double x = Recursion.powerOf10(3);
```



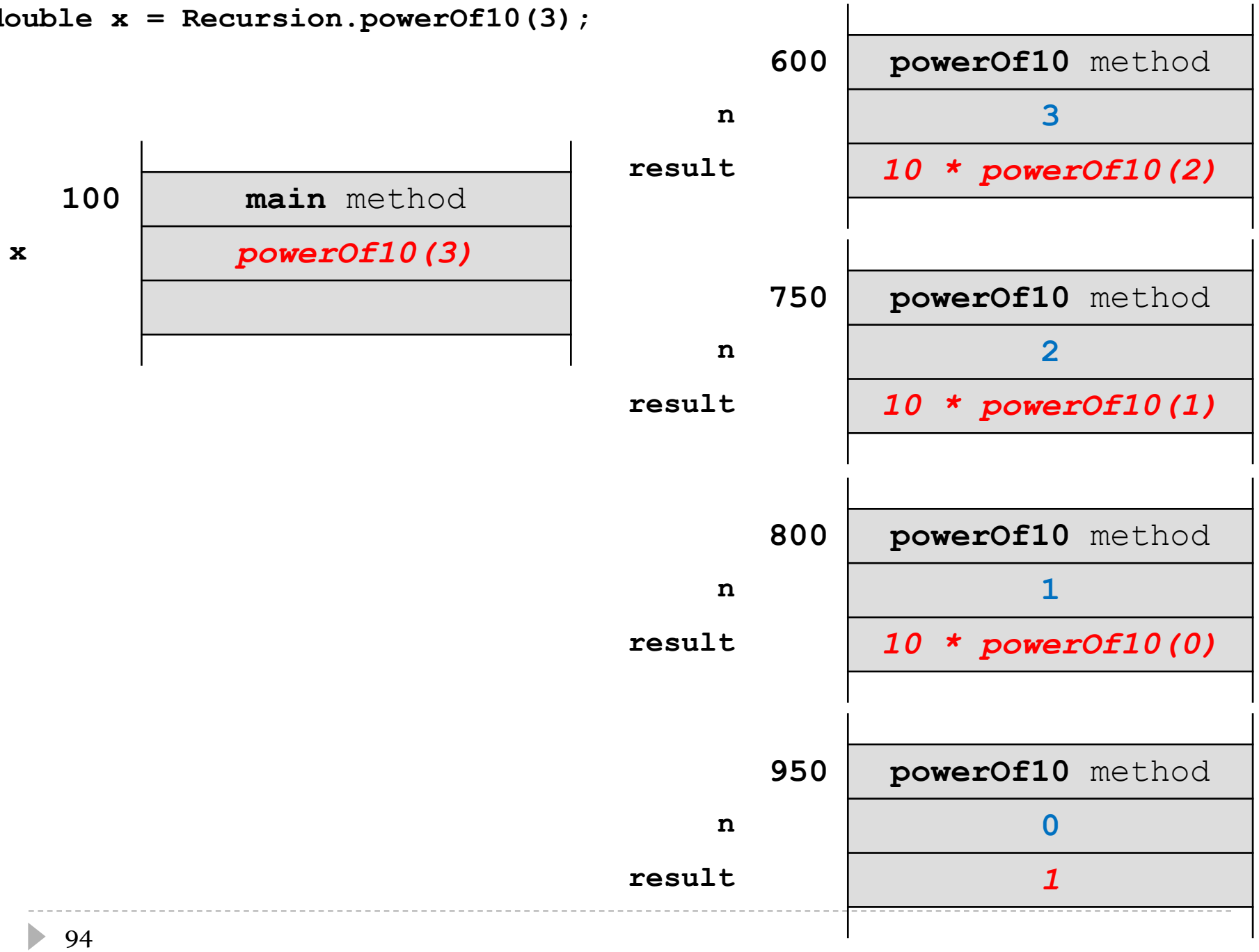
```
double x = Recursion.powerOf10(3);
```



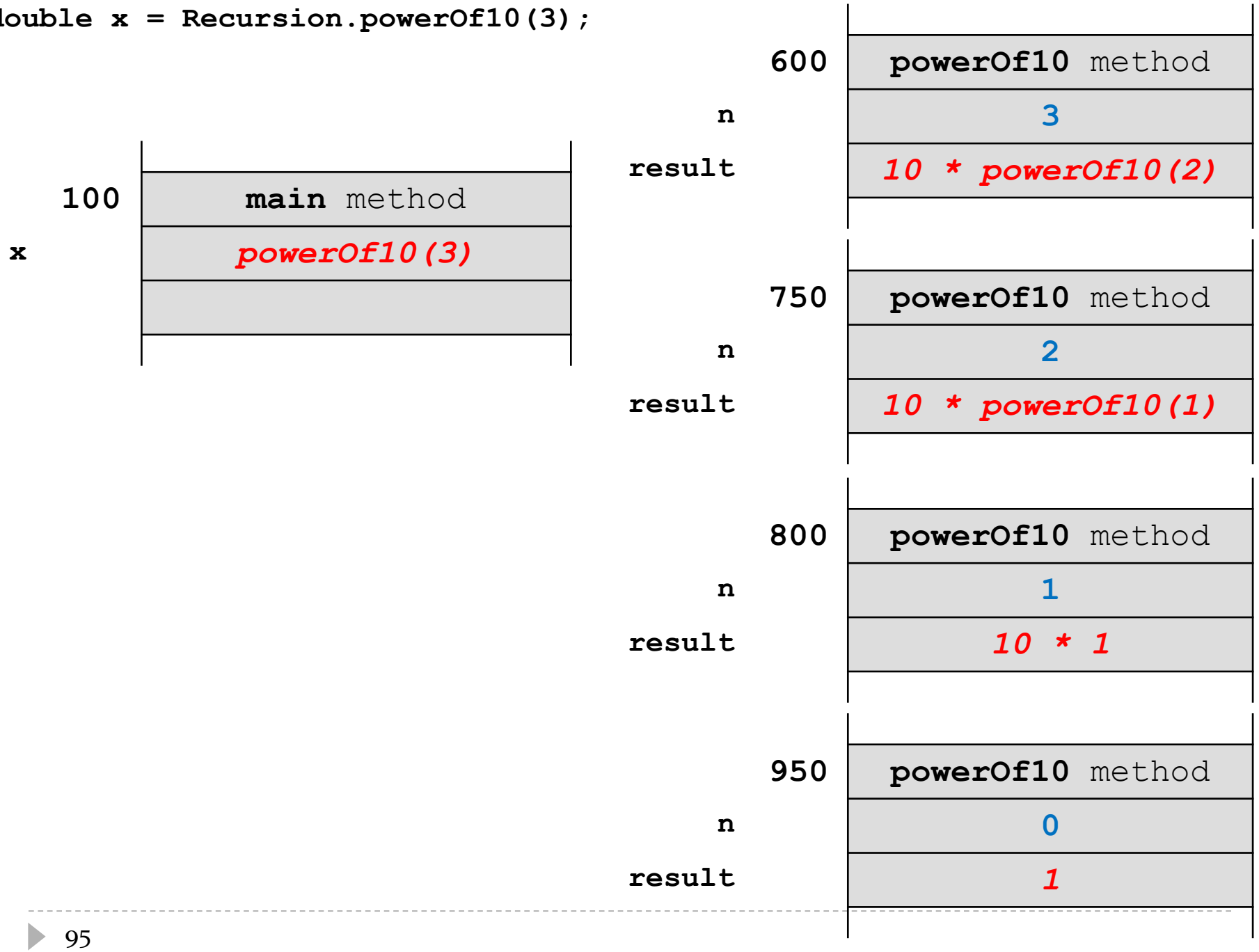
```
double x = Recursion.powerOf10(3);
```



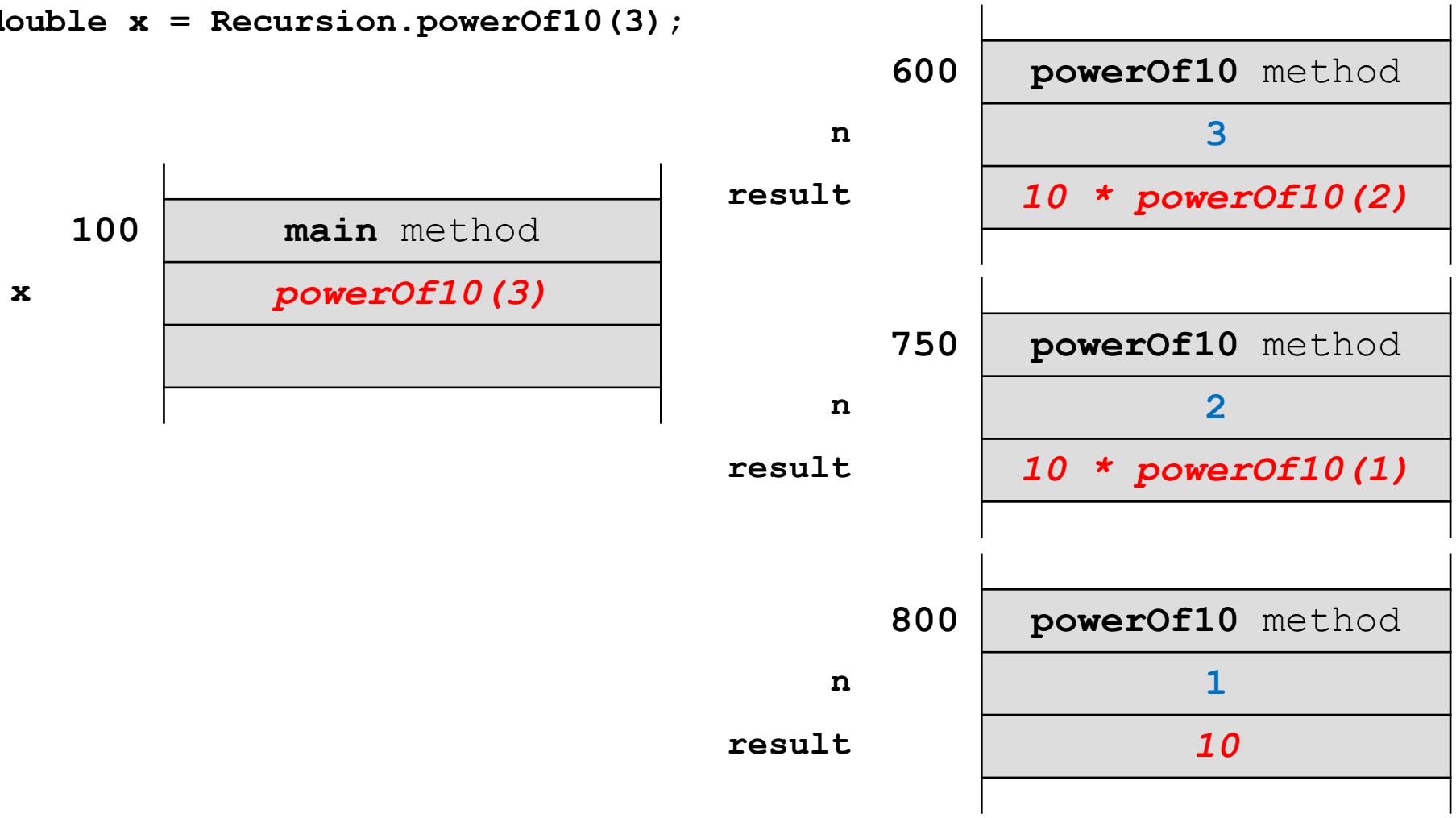
```
double x = Recursion.powerOf10(3);
```



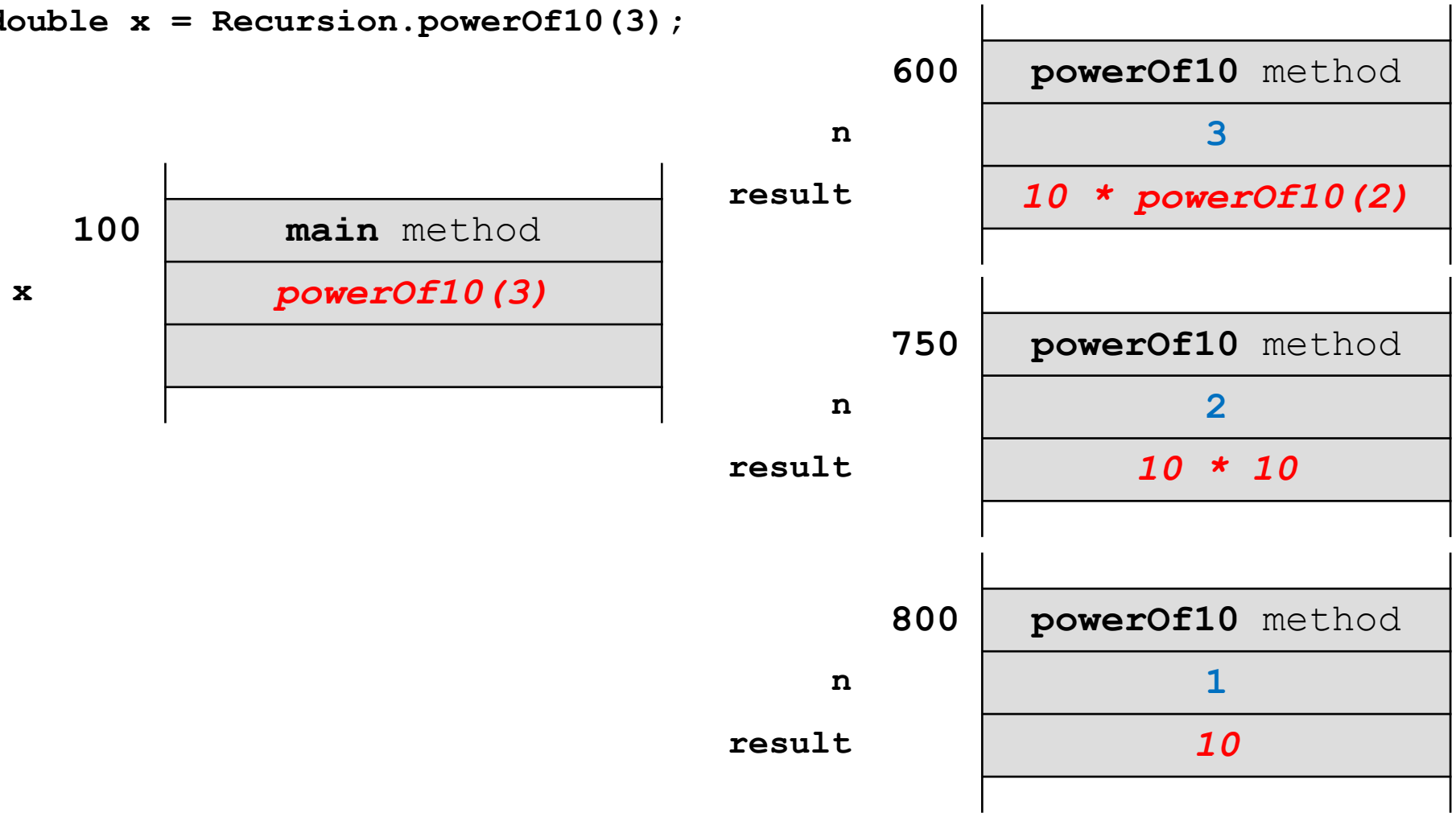
```
double x = Recursion.powerOf10(3);
```



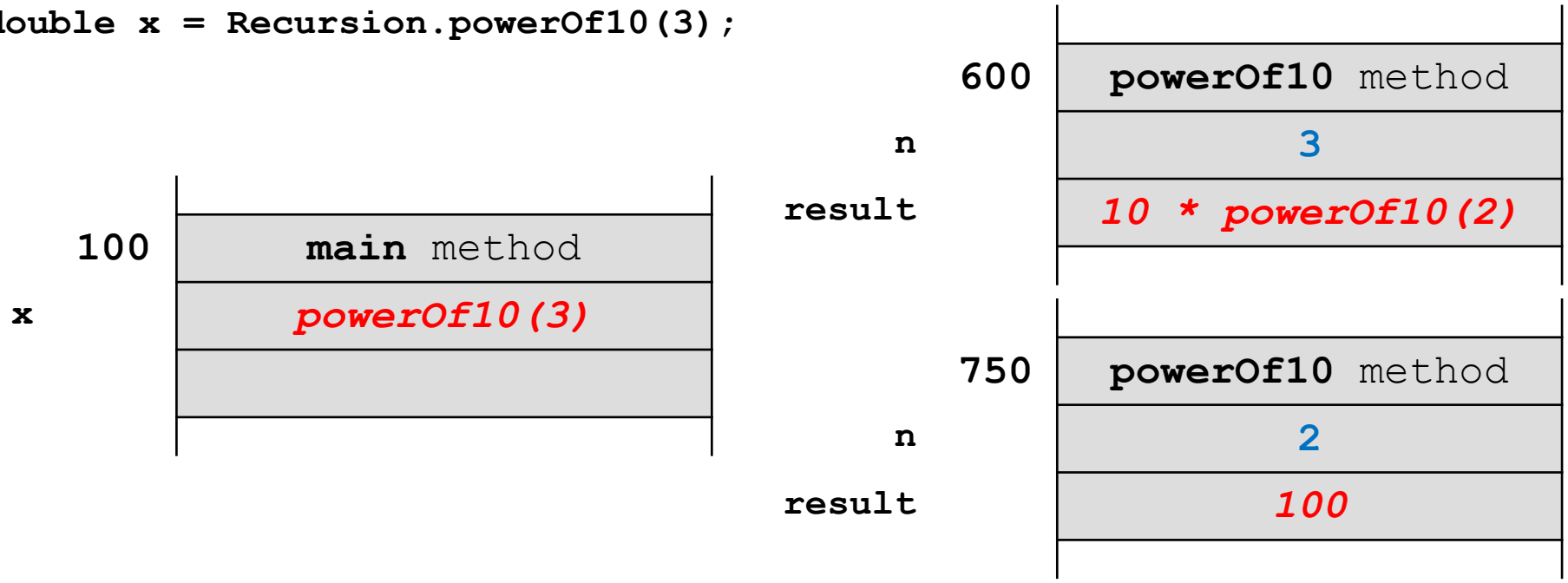
```
double x = Recursion.powerOf10(3);
```



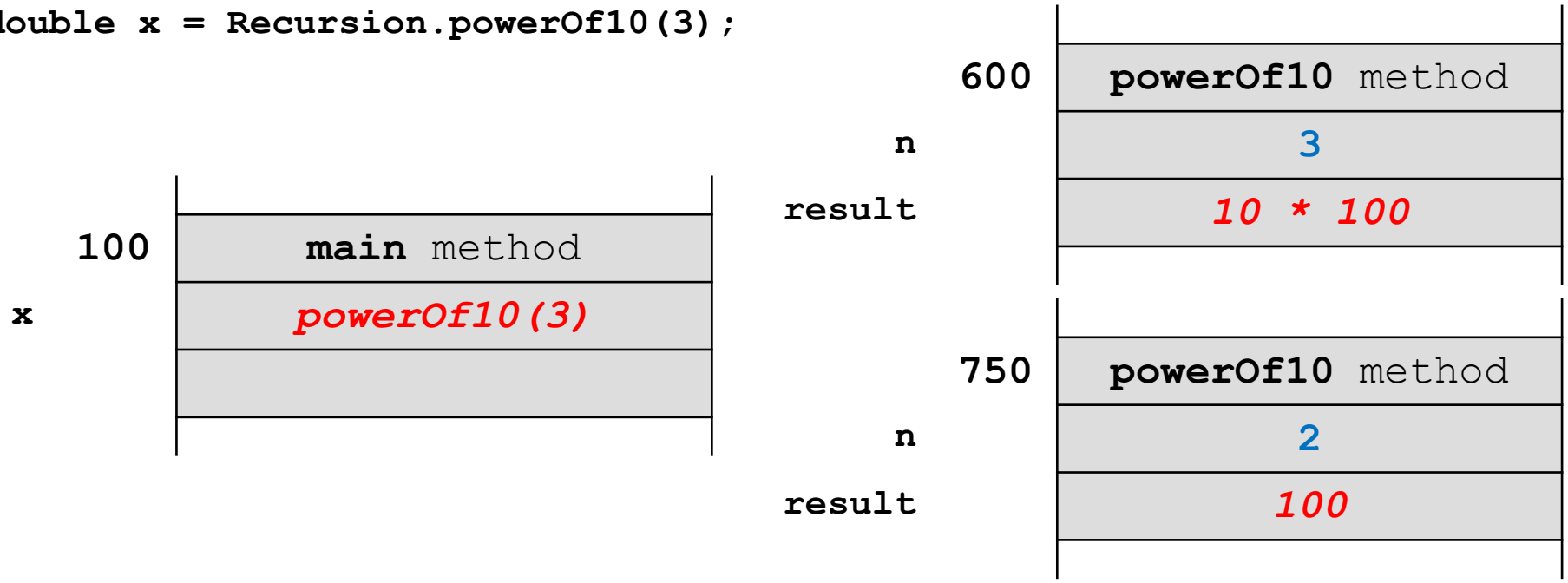
```
double x = Recursion.powerOf10(3);
```



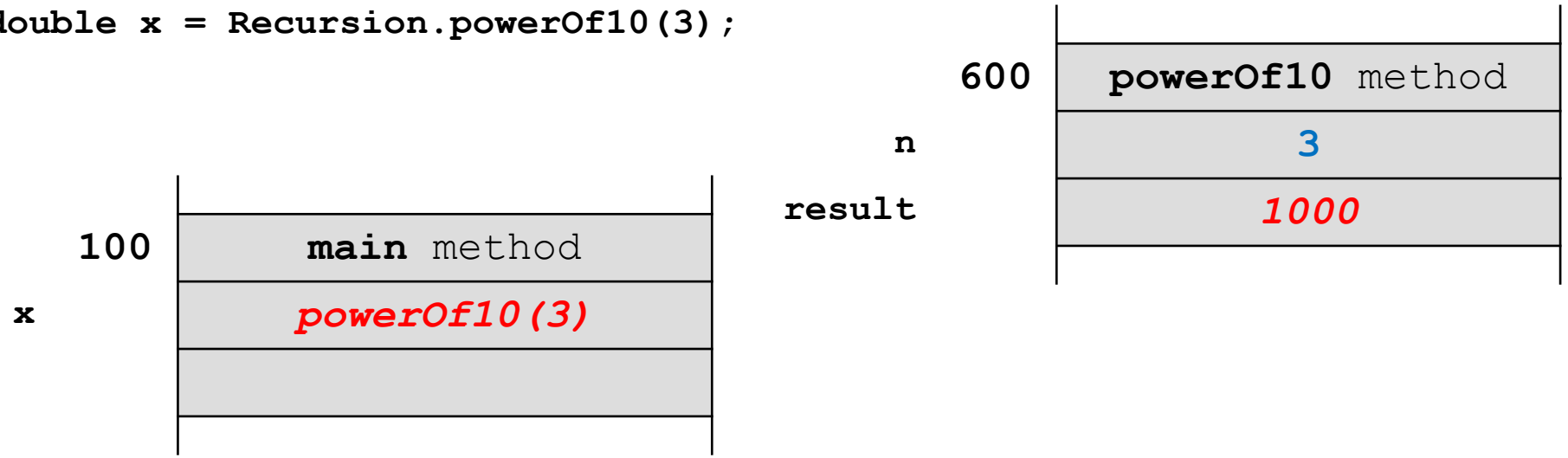
```
double x = Recursion.powerOf10(3);
```



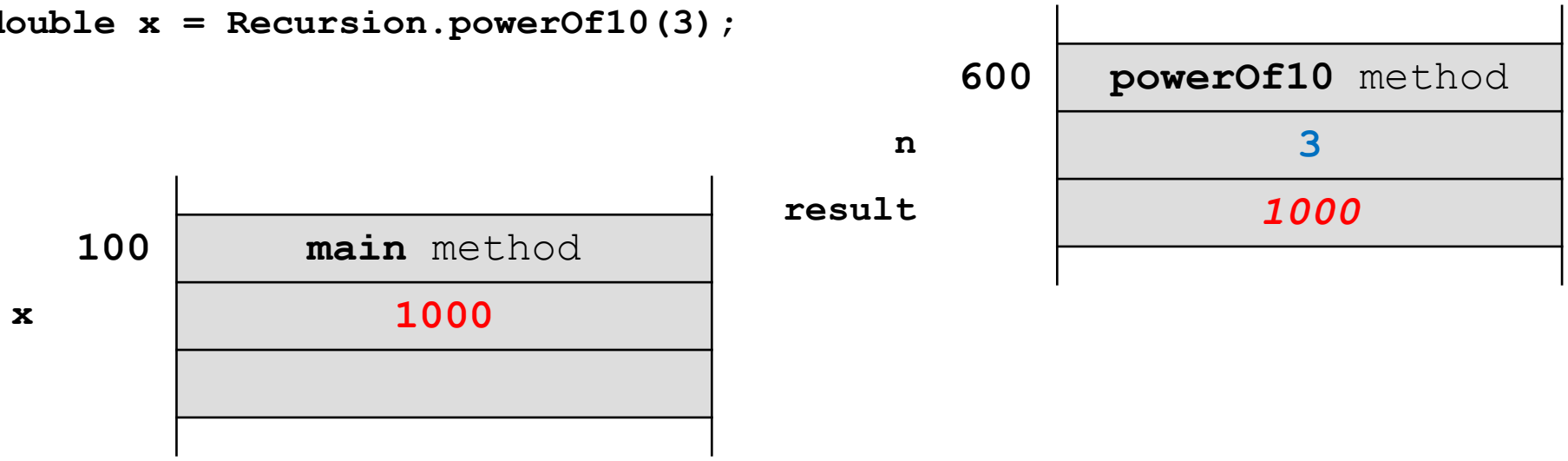
```
double x = Recursion.powerOf10(3);
```



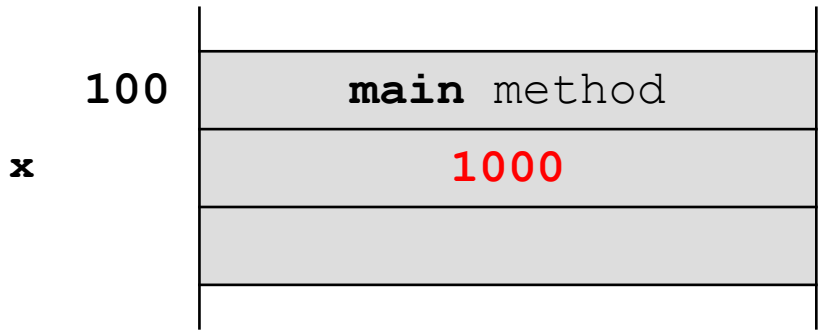
```
double x = Recursion.powerOf10(3);
```



```
double x = Recursion.powerOf10(3);
```



```
double x = Recursion.powerOf10(3);
```



Using a stack

- we to simulate what the JVM is doing during the recursive method
  - each recursive call that is not a base case pushes a 10 onto the stack
  - a recursive call that reaches a base case pushes a 1 onto the stack
  - pop the stack until it is empty, multiplying the values as you go

```
public static int powerOf10(int n) {

  Stack<Integer> t = new Stack<Integer>();
  // recursive calls
  while (n > 0) {
    t.push(10);
    n--;
  }
  // base case: n == 0
  t.push(1);

  // accumulate the result
  int result = t.pop();
  while (!t.isEmpty()) {
    result *= t.pop();
  }
  return result;
}
```

Converting a Recursive Method

- using a stack in the previous example is too complicated for the problem that we were trying to solve
  - but it illustrates the basic idea of using a stack to convert a recursive method to an iterative method
- more complicated examples require greater sophistication in exactly what is pushed onto the stack
  - see the make-up lab (Part 1) when it becomes available

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

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

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

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

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

Implementation with Array

- the ArrayList version of stack hints at how to implement a stack using a plain array
  - however, an array always holds a fixed number of elements
    - you cannot add to the end of the array without creating a new array
    - you cannot remove elements from the array without creating a new array
- instead of adding and removing from the end of the array, we need to keep track of which element of the array represents the current top of the stack
  - we need a field for this index

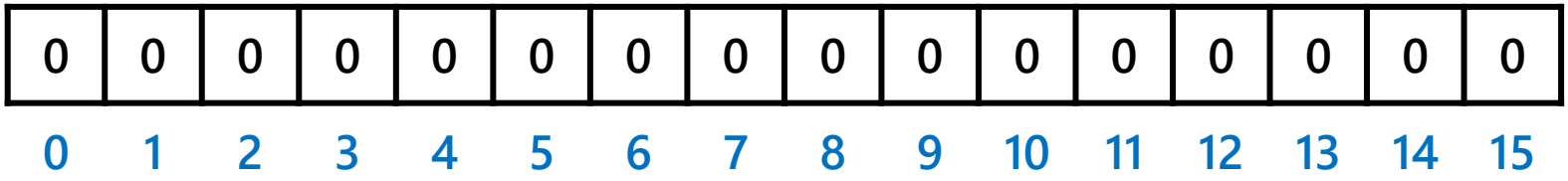
```
import java.util.Arrays;
import java.util.EmptyStackException;

public class IntStack {
  // the initial capacity of the stack
  private static final int DEFAULT_CAPACITY = 16;

  // the array that stores the stack
  private int[] stack;

  // the index of the top of the stack (equal to -1 for an empty stack)
  private int topIndex;
```

```java
/**
 * Create an empty stack.
 */
public IntStack() {
  this.stack = new int[IntStack.DEFAULT_CAPACITY];
  this.topIndex = -1;
}
```

Implementation with Array

```
IntStack t = new IntStack();
```

this.topIndex == -1



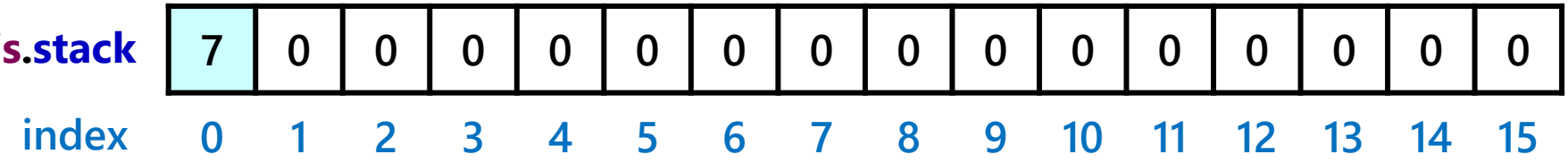
Implementation with Array

- pushing a value onto the stack:
  - increment **this.topIndex**
  - set the value at **this.stack[this.topIndex]**

Implementation with Array

```
IntStack t = new IntStack();
t.push(7);
```

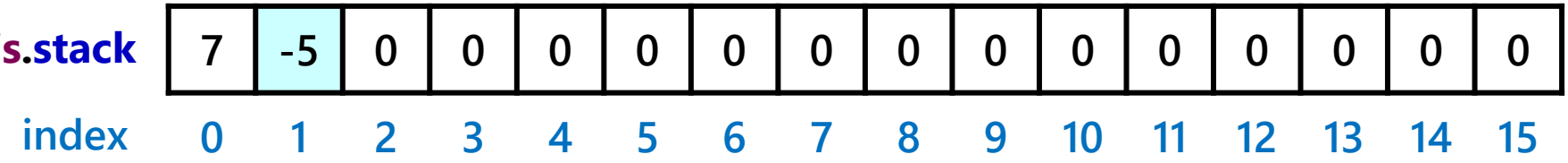
this.topIndex == 0



Implementation with Array

```
IntStack t = new IntStack();
t.push(7);
t.push(-5);
```

this.topIndex == 1



Implementation with Array

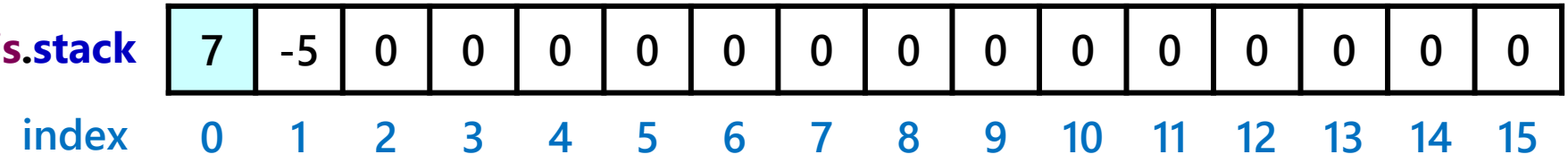
- popping a value from the stack:
  - get the value at **this.stack[this.topIndex]**
  - decrement **this.topIndex**
  - return the value

- notice that we do not need to modify the value stored in the array

Implementation with Array

```
IntStack t = new IntStack();
t.push(7);
t.push(-5);
int value = t.pop();   // value == -5
```

this.topIndex == 0

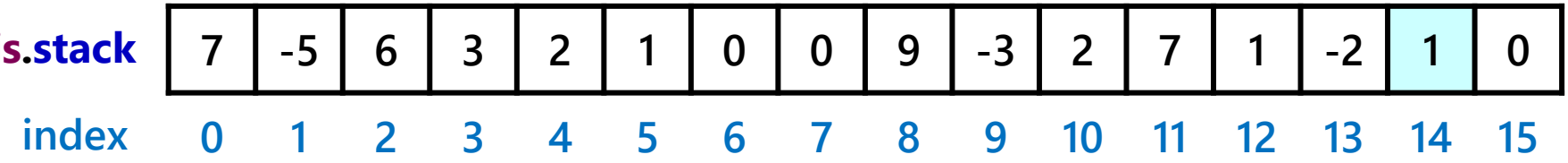


```
  /**
   * Pop and return the top element of the stack.
   *
   * @return the top element of the stack
   * @throws EmptyStackException if the stack is empty
   */
  public int pop() {
    // is the stack empty?
    if (this.topIndex == -1) {
      throw new EmptyStackException();
    }
    // get the element at the top of the stack
    int element = this.stack[this.topIndex];

    // adjust the top of stack index
    this.topIndex--;

    // return the element that was on the top of the stack
    return element;
  }
```

Implementation with Array

// stack state when we can safely do one more push

this.topIndex == 14



```
/**
 * Push an element onto the top of the stack.
 *
 * @param element the element to push onto the stack
 */
public void push(int element) {
  // is there capacity for one more element?
  if (this.topIndex < this.stack.length - 1) {
    // increment the top of stack index and insert the element
    this.topIndex++;
    this.stack[this.topIndex] = element;
  }
  else {
```

Adding Capacity

- if we run out of capacity in the current array we need to add capacity by doing the following:
  - make a new array with greater capacity
    - how much more capacity?
  - copy the old array into the new array
  - set **this.stack** to refer to the new array
  - push the element onto the stack

```
  else {
    // make a new array with double previous capacity
    int[] newStack = new int[this.stack.length * 2];

    // copy the old array into the new array
    for (int i = 0; i < this.stack.length; i++) {
      newStack[i] = this.stack[i];
    }

    // refer to the new array and push the element onto the stack
    this.stack = newStack;
    this.push(element);
  }
}
```

Adding Capacity

- when working with arrays, it is a common operation to have to create a new larger array when you run out of capacity in the existing array
- you should use **Arrays.*copyOf*** to create and copy an existing array into a new array

```
    else {
        int[] newStack = Arrays.copyOf(this.stack, this.stack.length * 2);
        // refer to the new array and push the element onto the stack
        this.stack = newStack;
        this.push(element);
    }
}
```