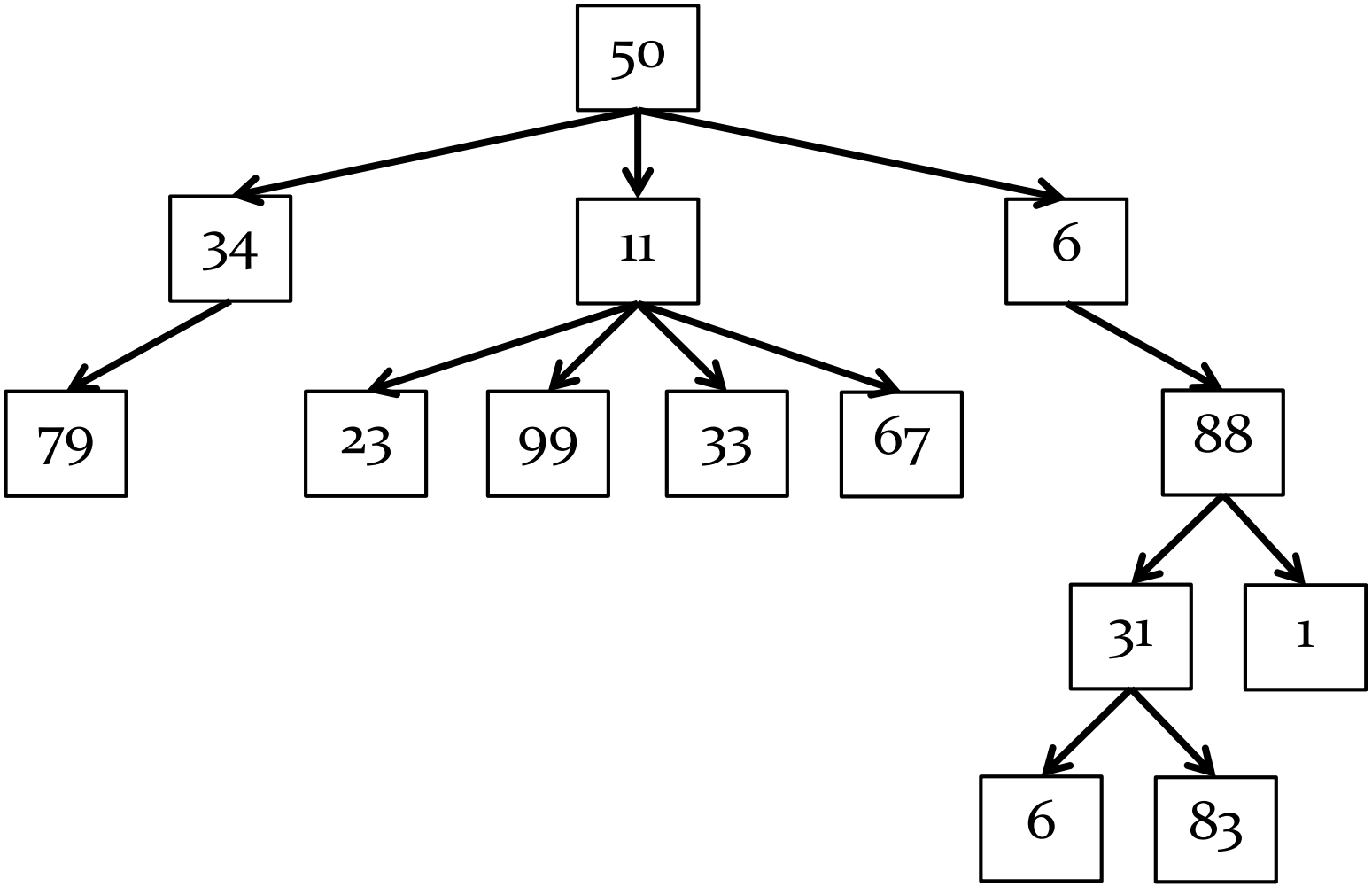
Trees

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, Lost Circles
  - transportation networks
    - e.g., http://ac.fltmaps.com/en
  - many other examples
    - http://www.visualcomplexity.com/vc/

Trees

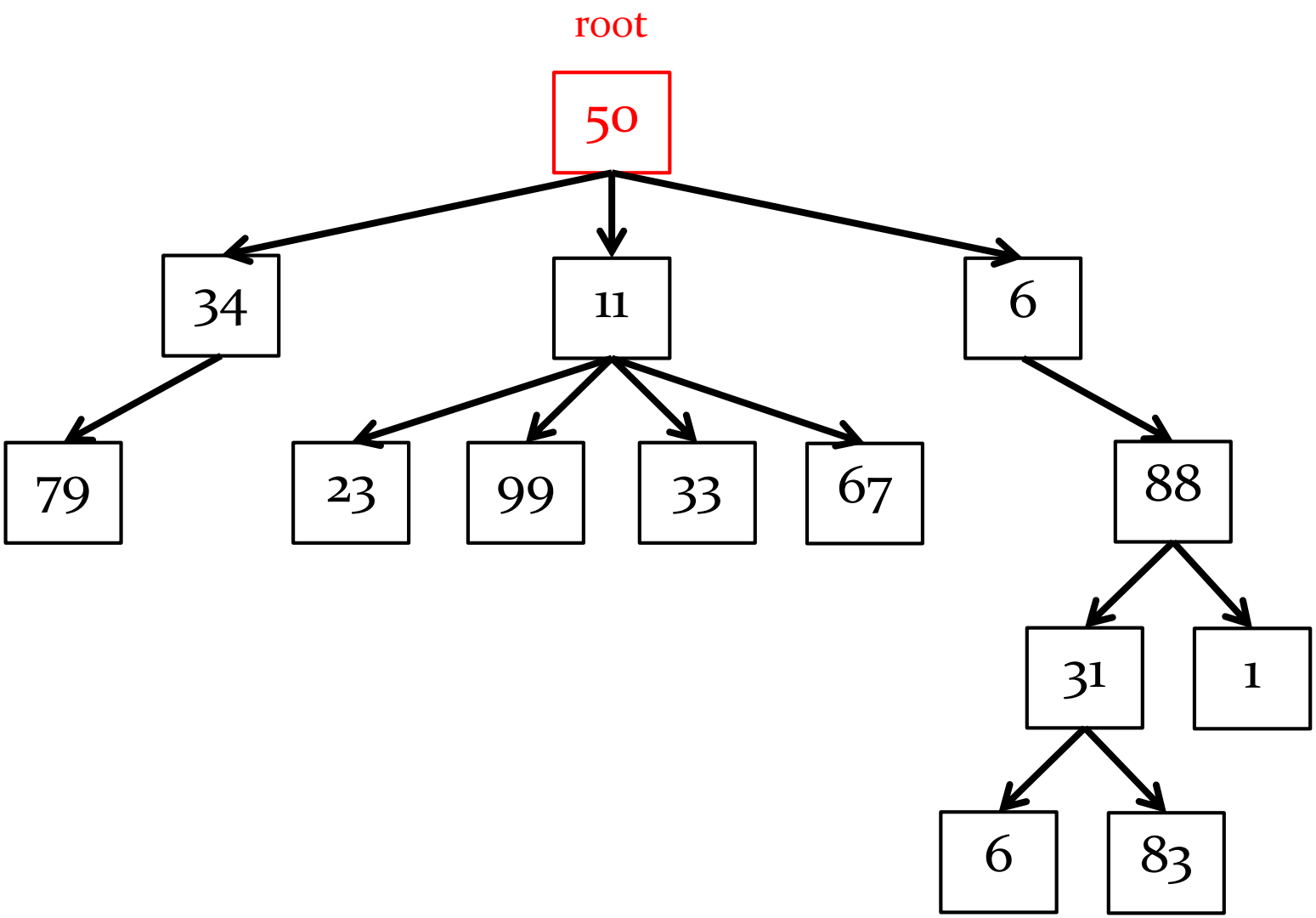
- trees are special cases of graphs
- a tree is a data structure made up of nodes
  - each node stores data
  - each node has links to zero or more nodes in the next level of the tree
    - children of the node
  - each node has exactly one parent node
    - except for the root node


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

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

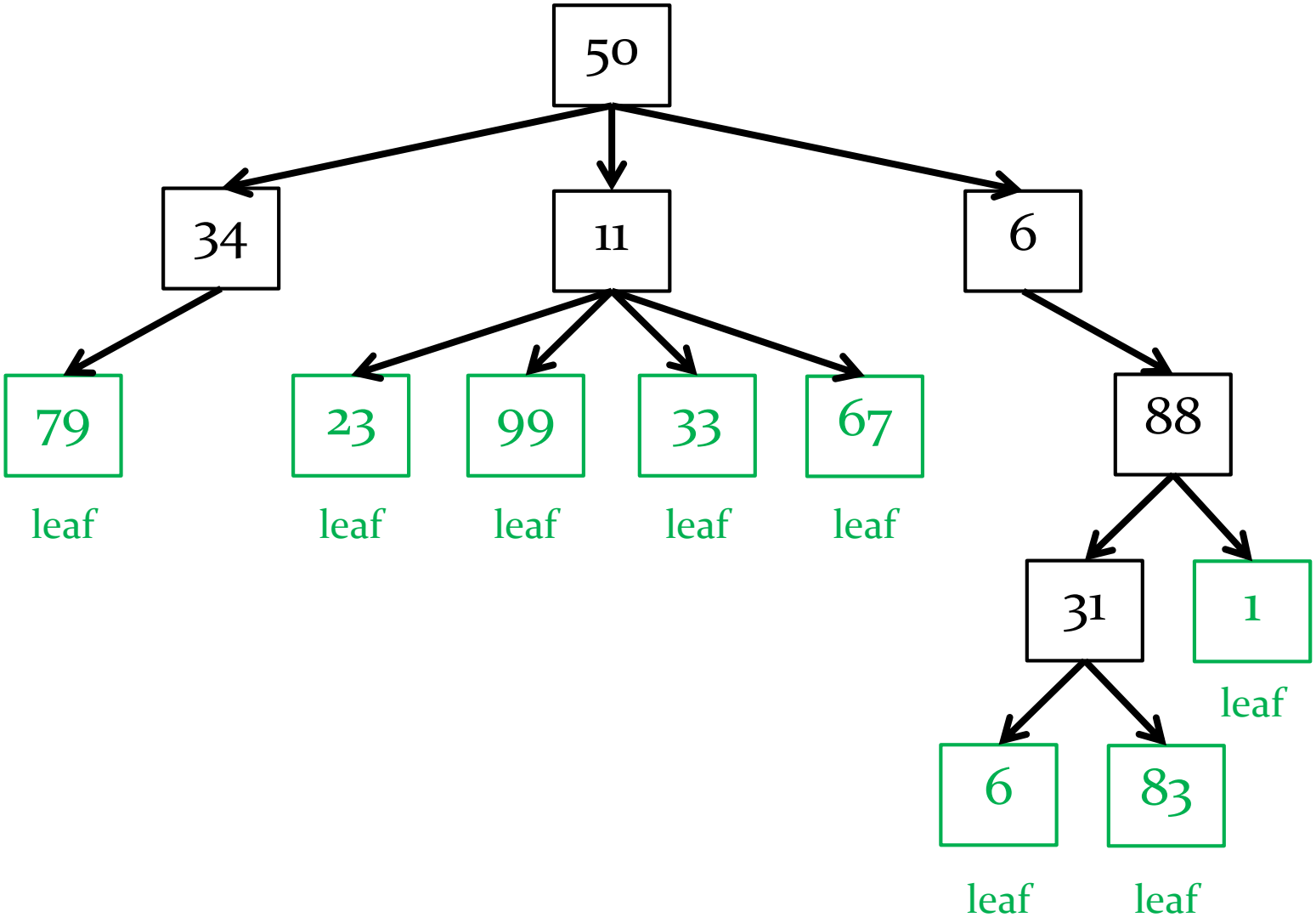
Trees

- 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

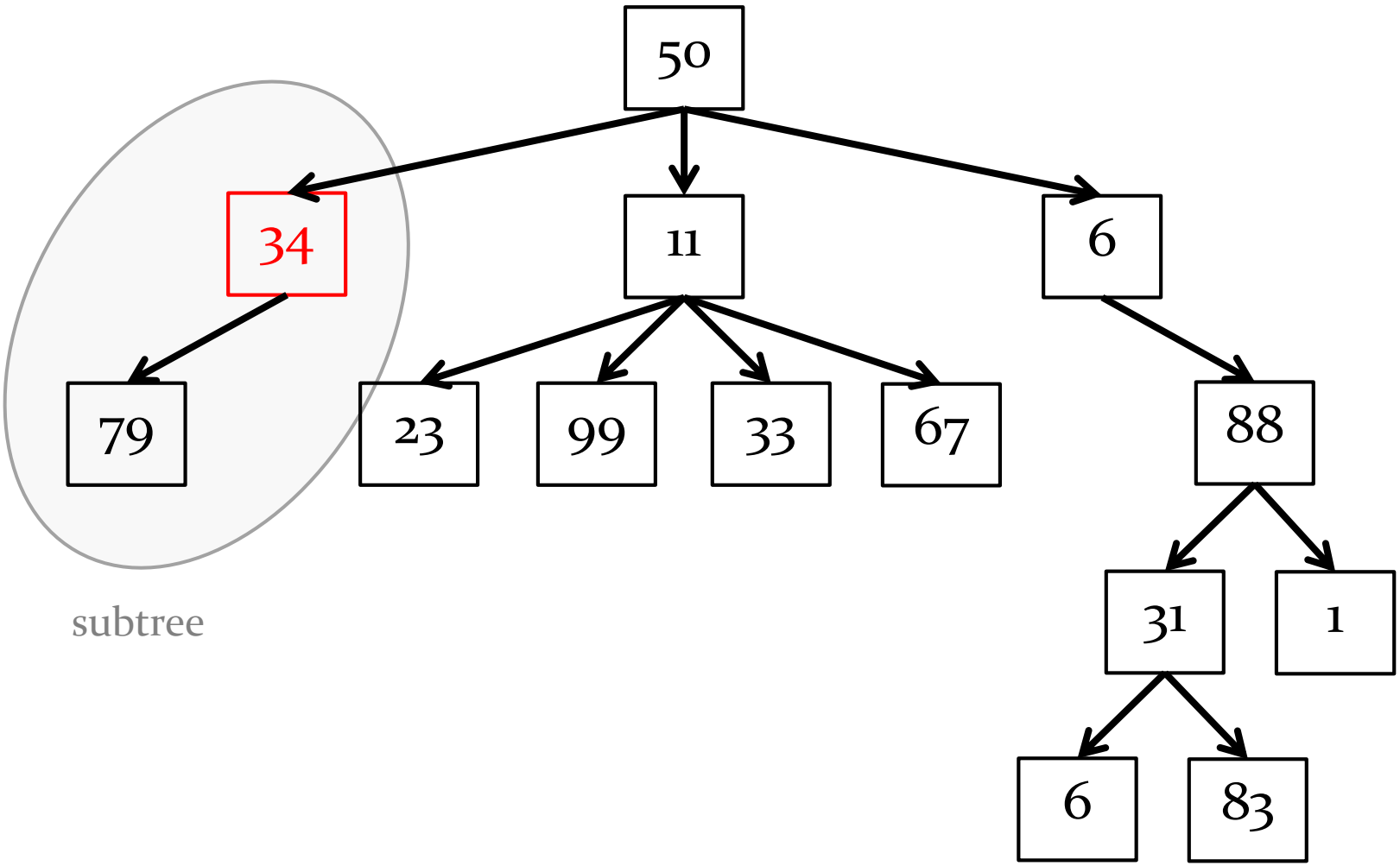
Trees

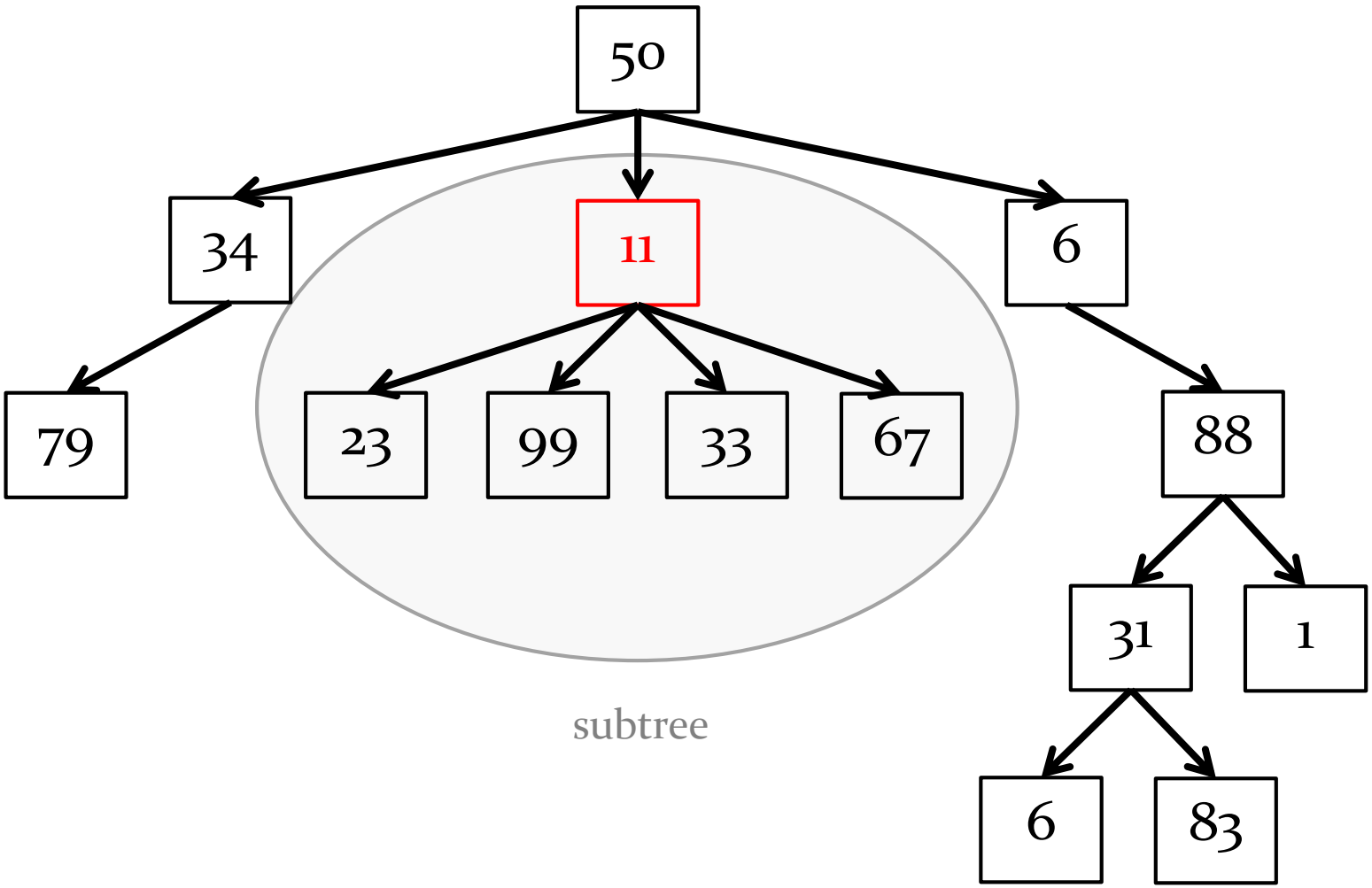
- a node without any children is called a leaf

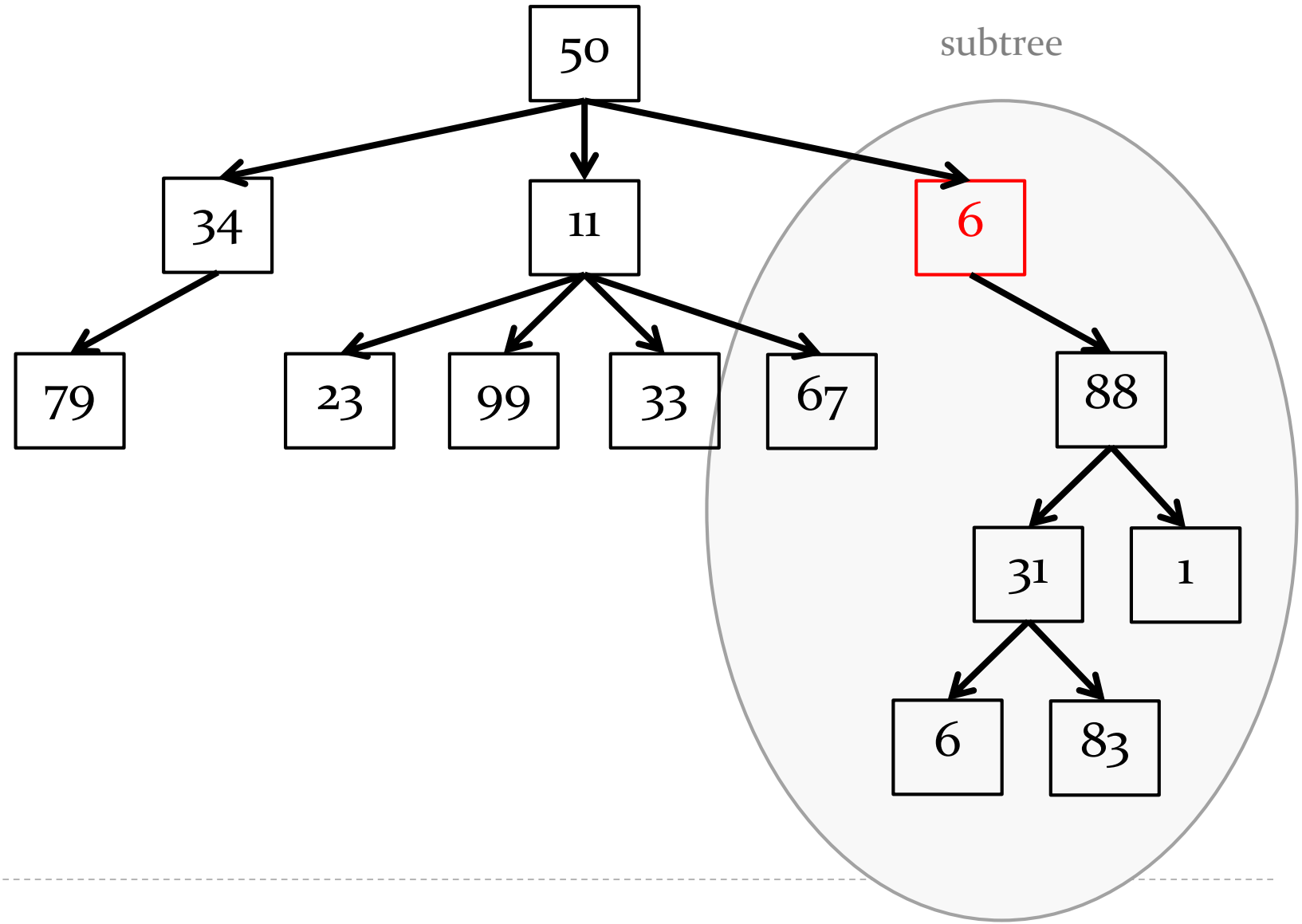

50
34
11
6
79
23
99
33
67
88
leaf
leaf
leaf
leaf
leaf
31
1
leaf
6
83
leaf
leaf

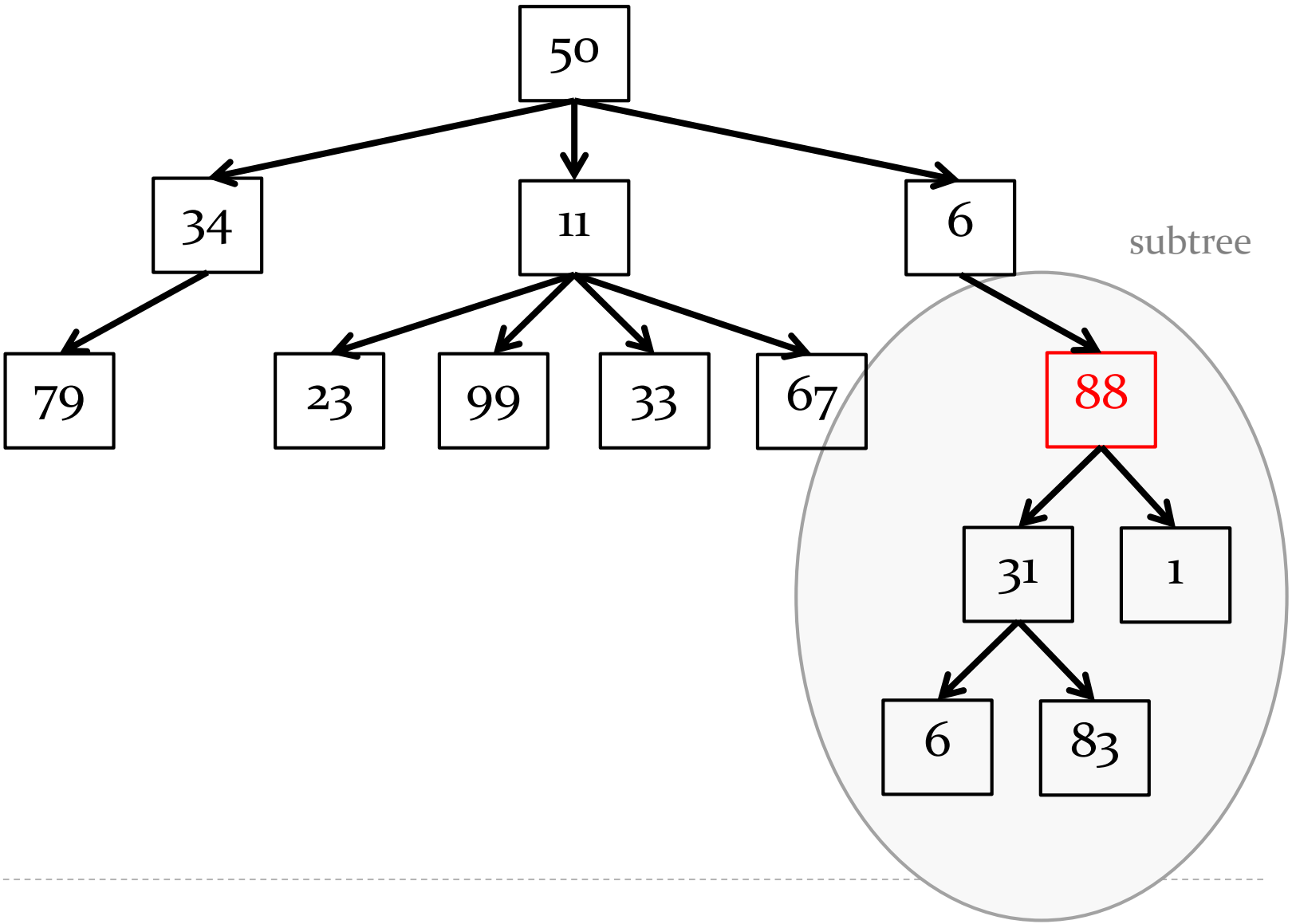
Trees

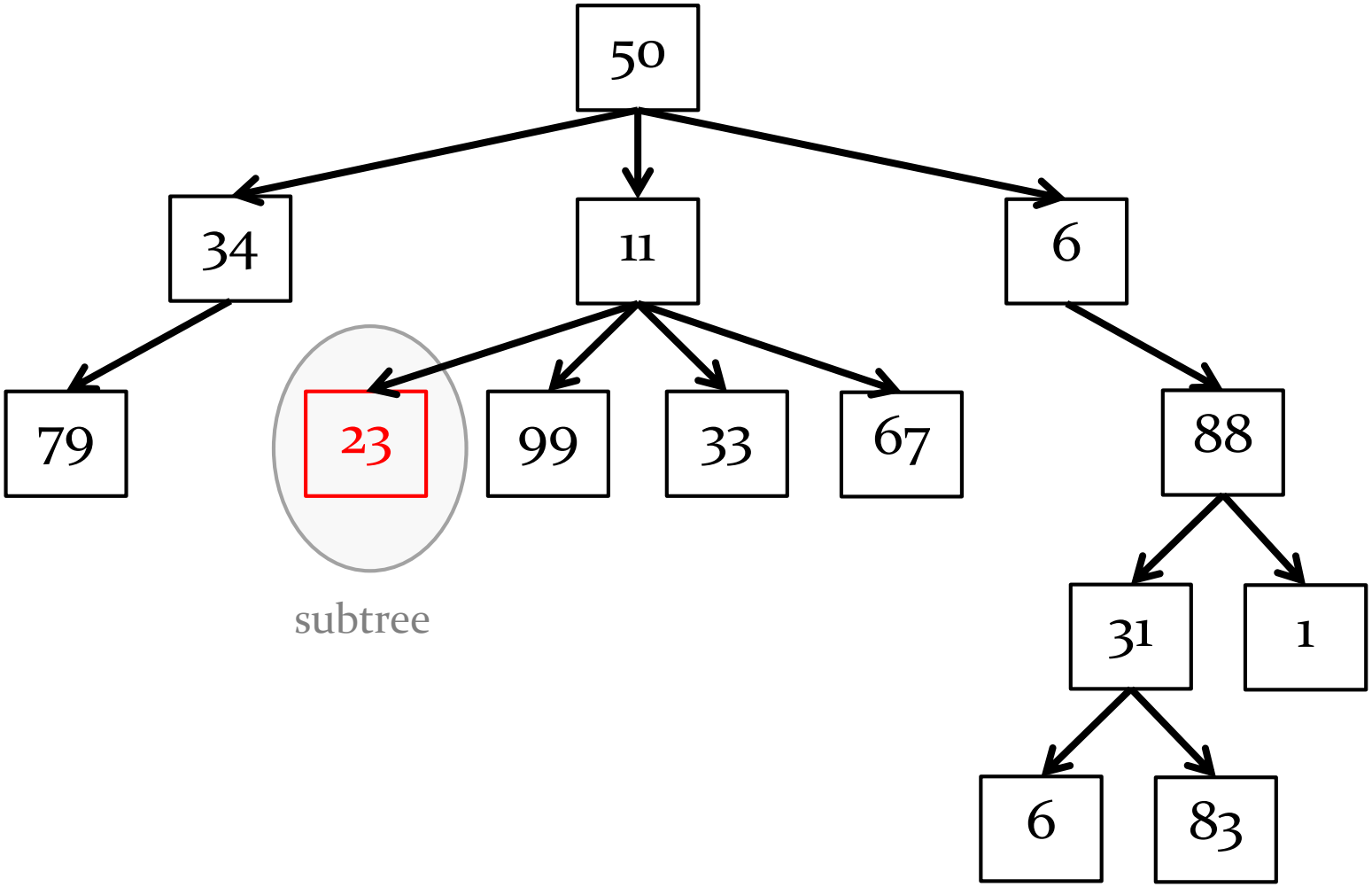
- the recursive structure of a tree means that every node is the root of a tree


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

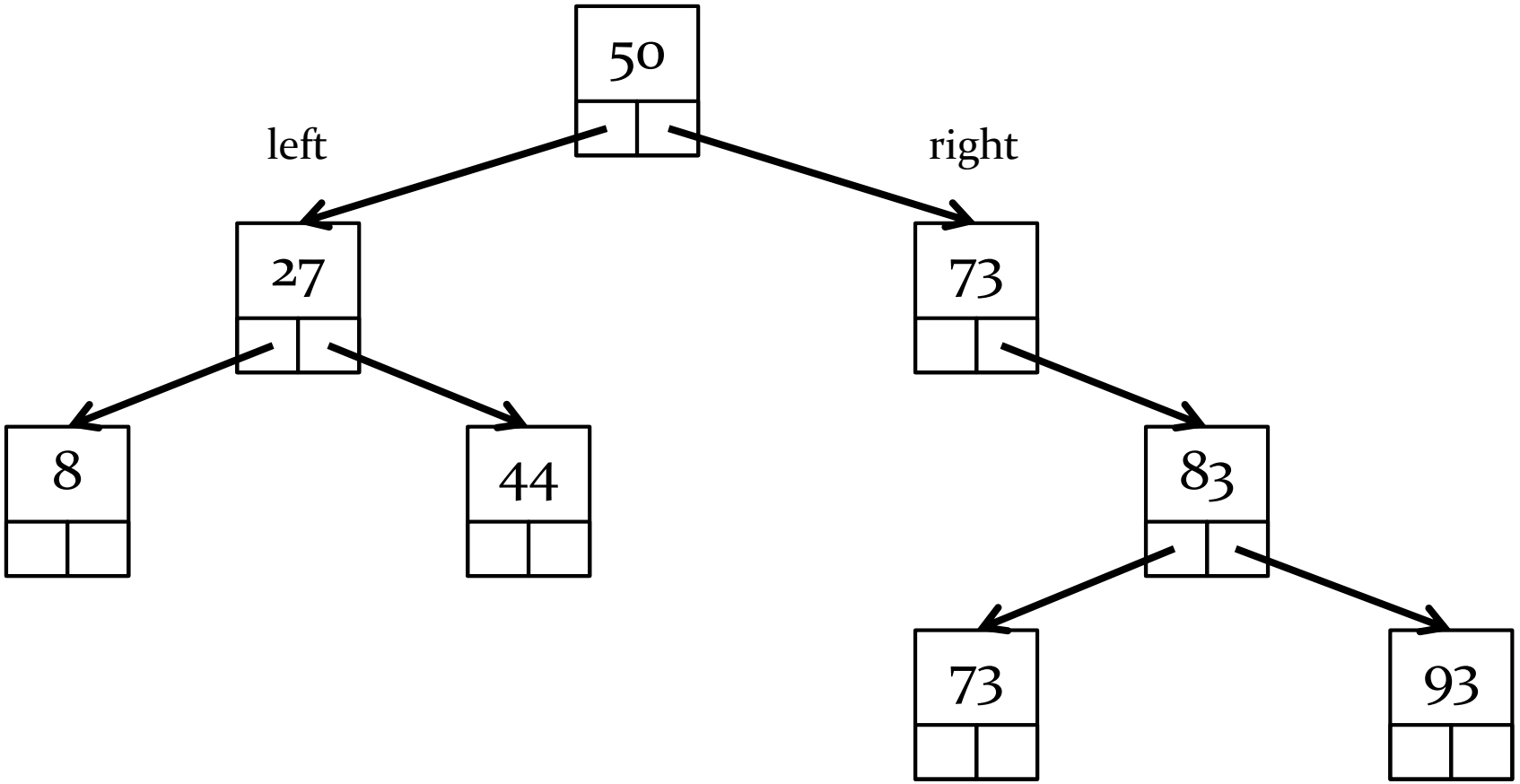

50
subtree
34
11
6
79
23
99
33
67
88
31
1
6
83

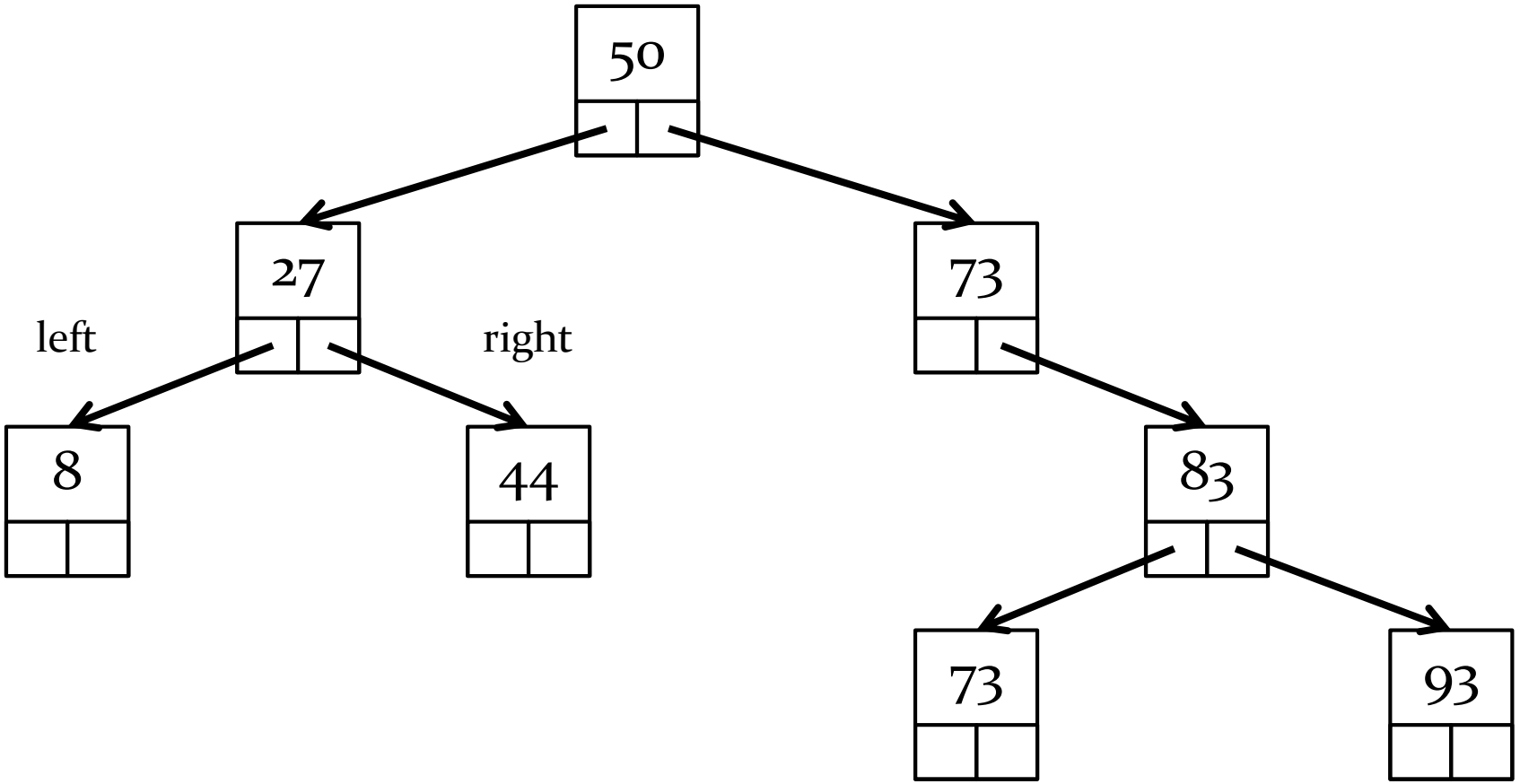

50
34
11
6
subtree
79
23
99
33
67
88
31
1
6
83

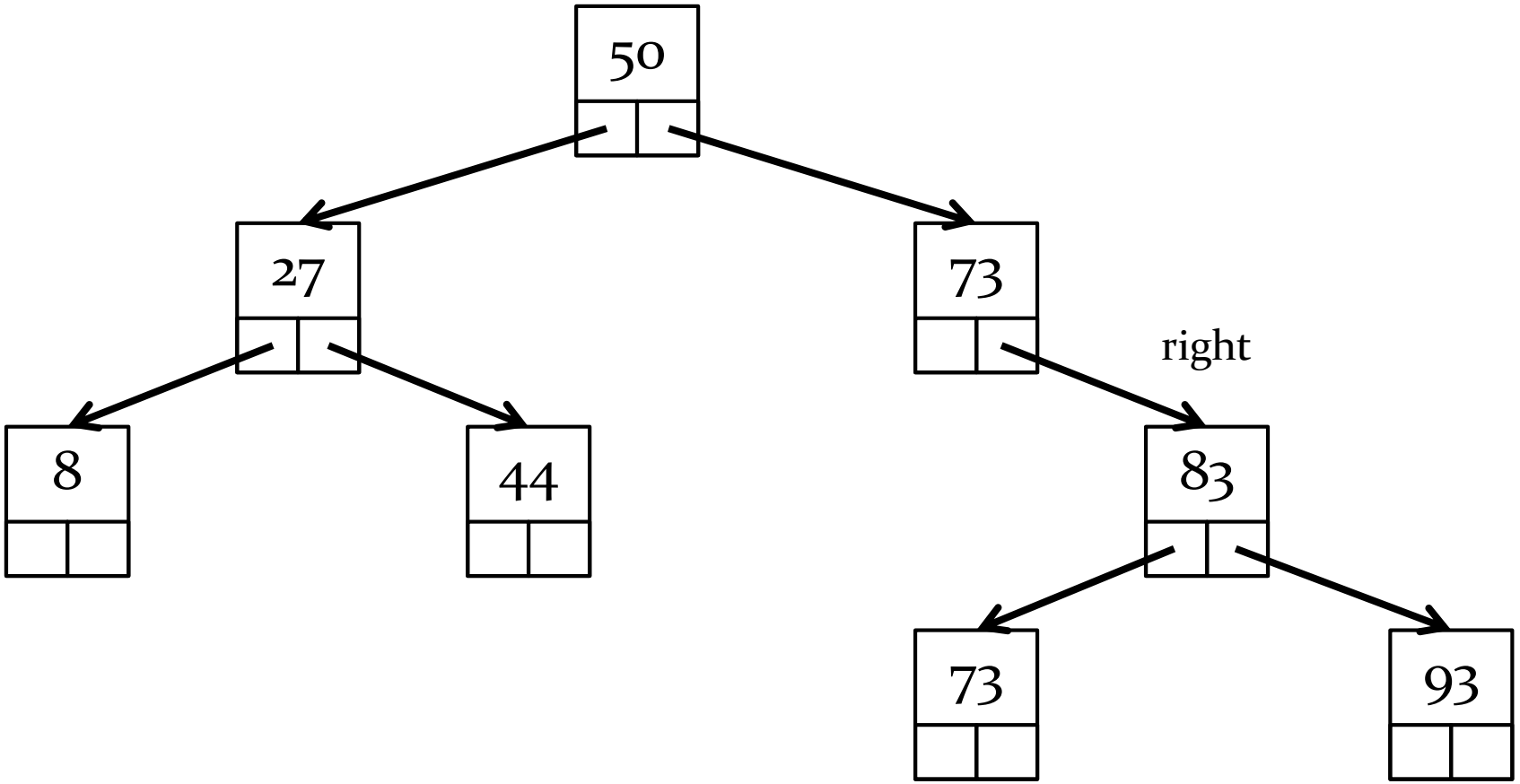

50
34
11
6
79
23
99
33
67
88
subtree
31
1
6
83

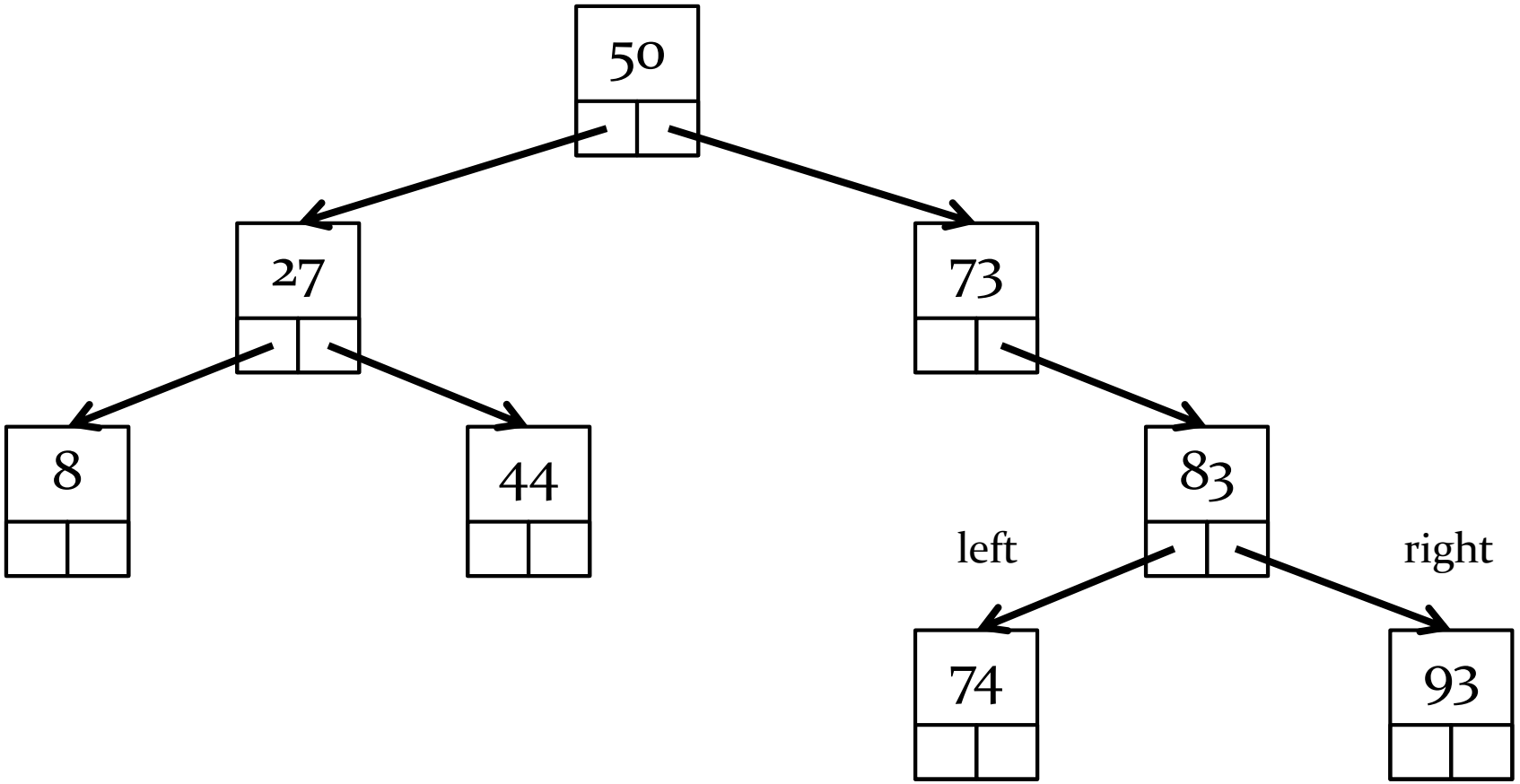
Binary Tree

- 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

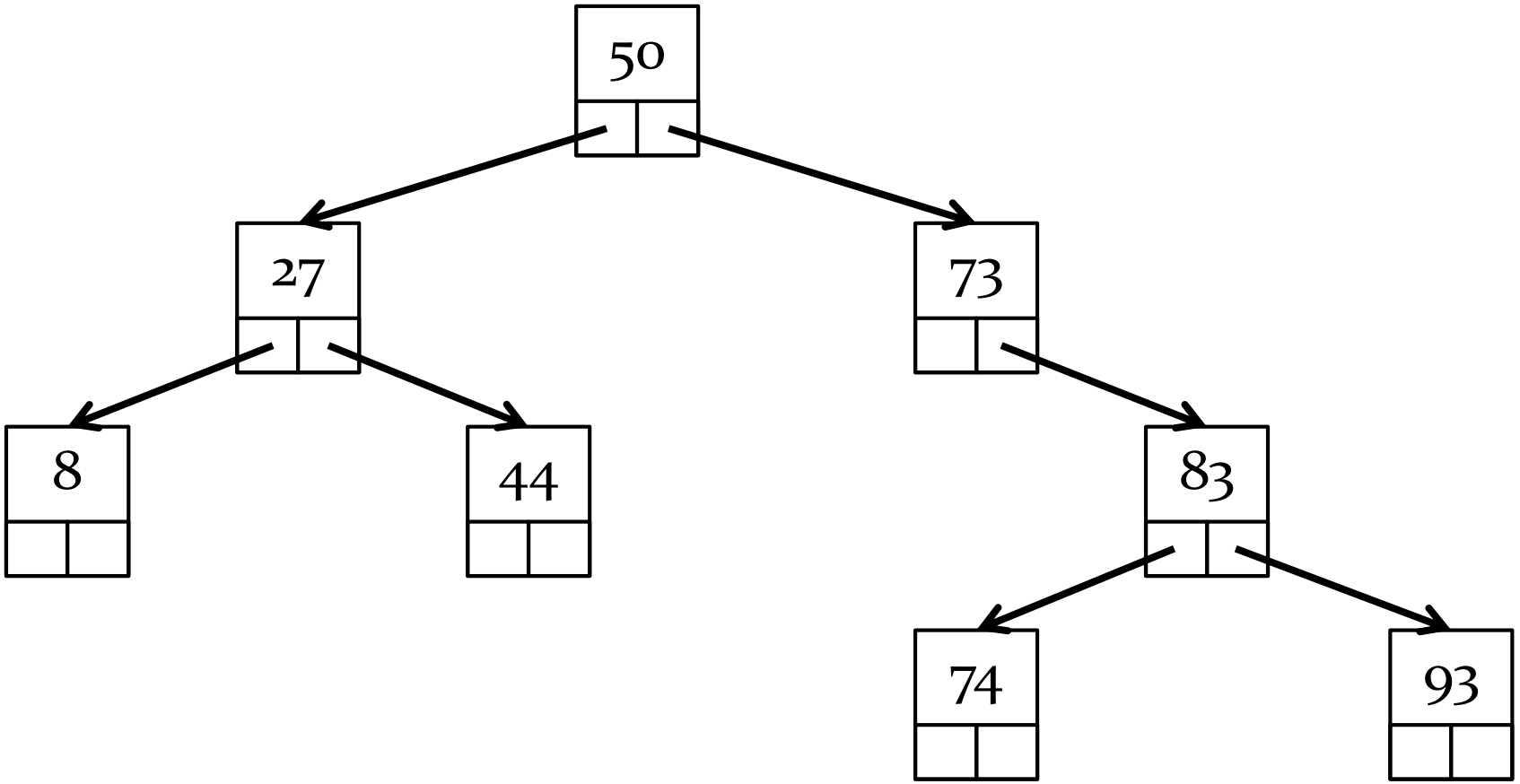
```
public static <E> boolean contains(E element) {
  return contains(element, this.root);
}
```

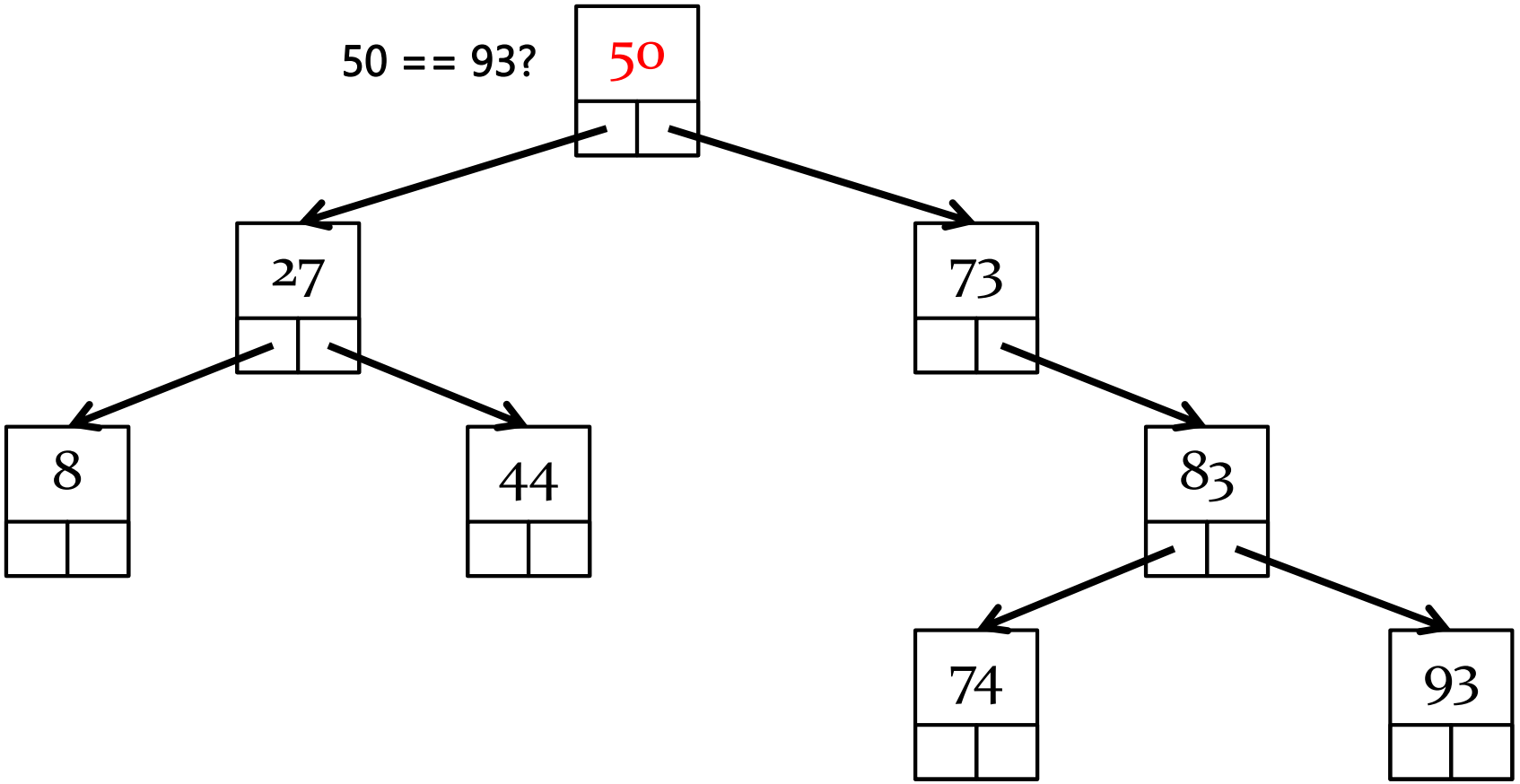
```
private static <E> boolean contains(E element, Node<E> node) {
  boolean inLeftTree = contains(element, node.left);
  if (inLeftTree) {
    return true;
  }
  boolean inRightTree = contains(element, node.right);
  return inRightTree;
}
```

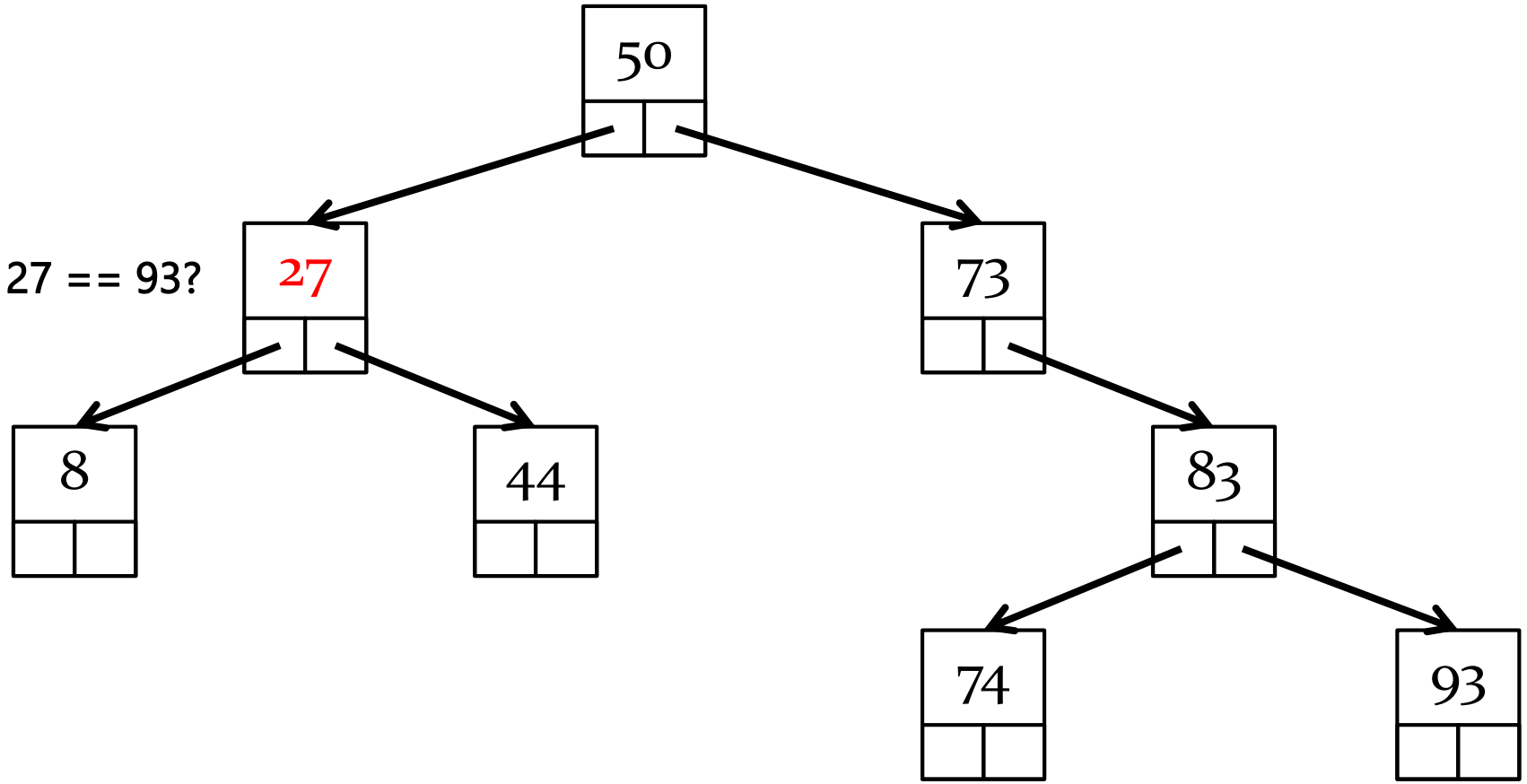
examine left subtree

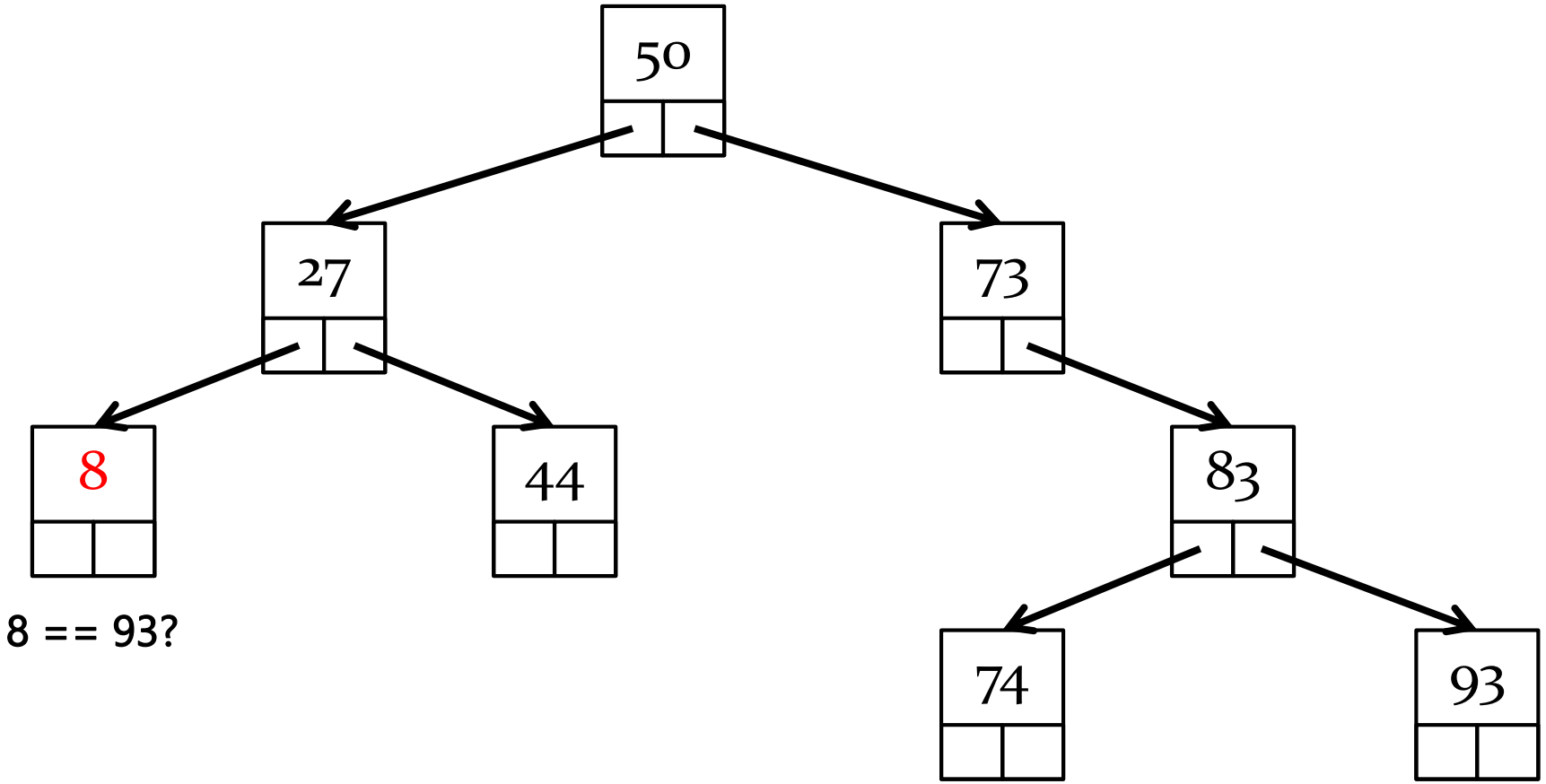
examine right subtree

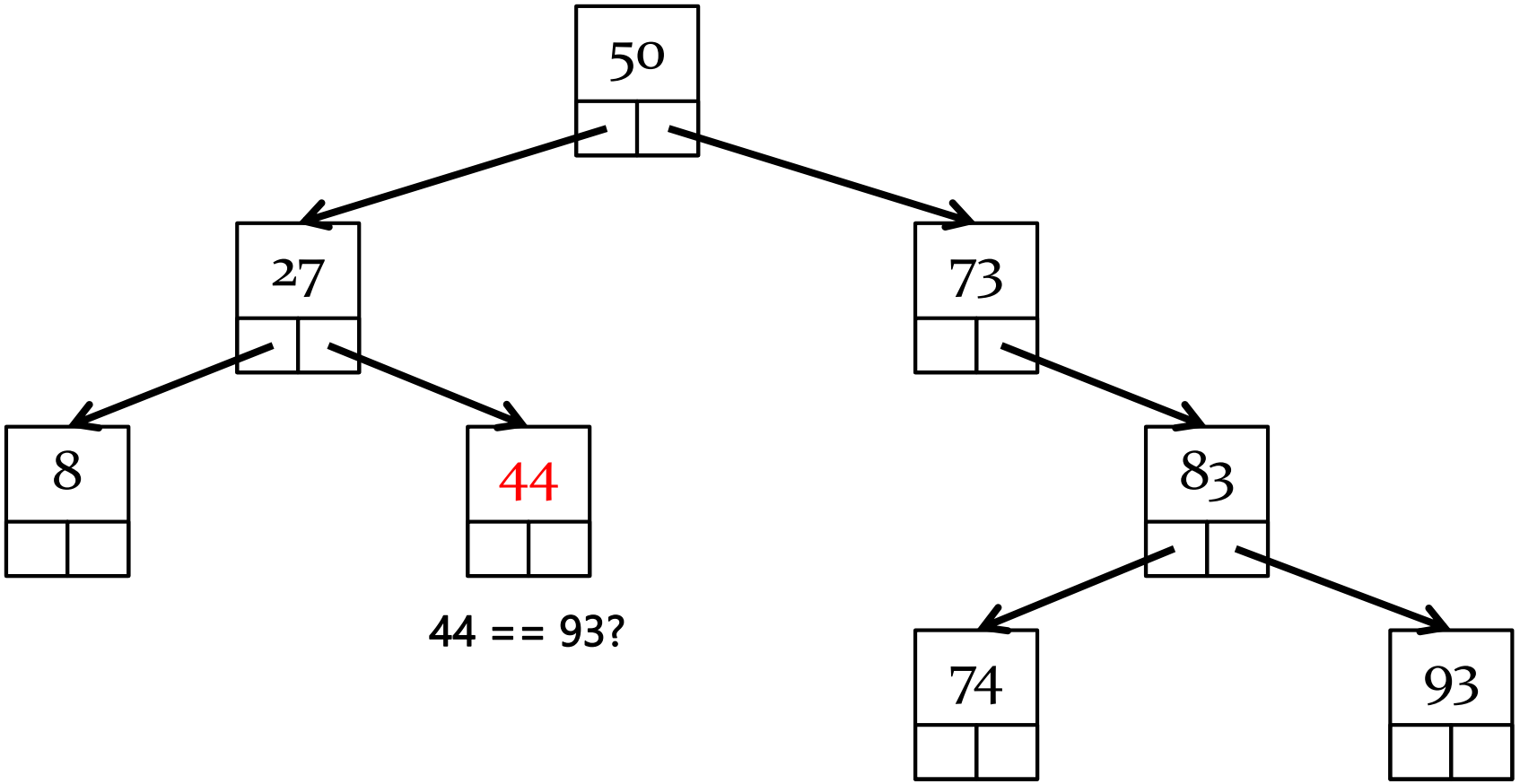
t.contains(93)

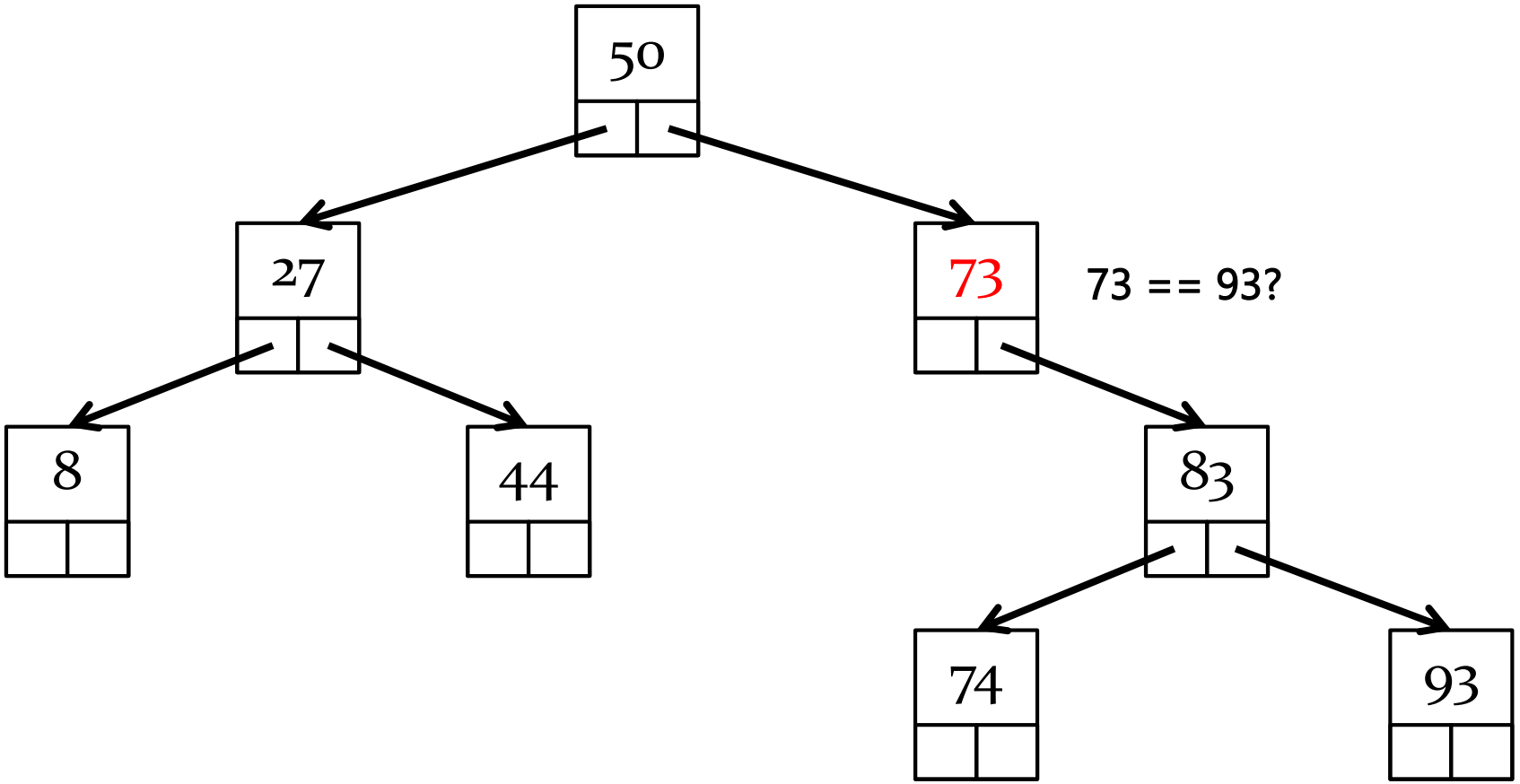


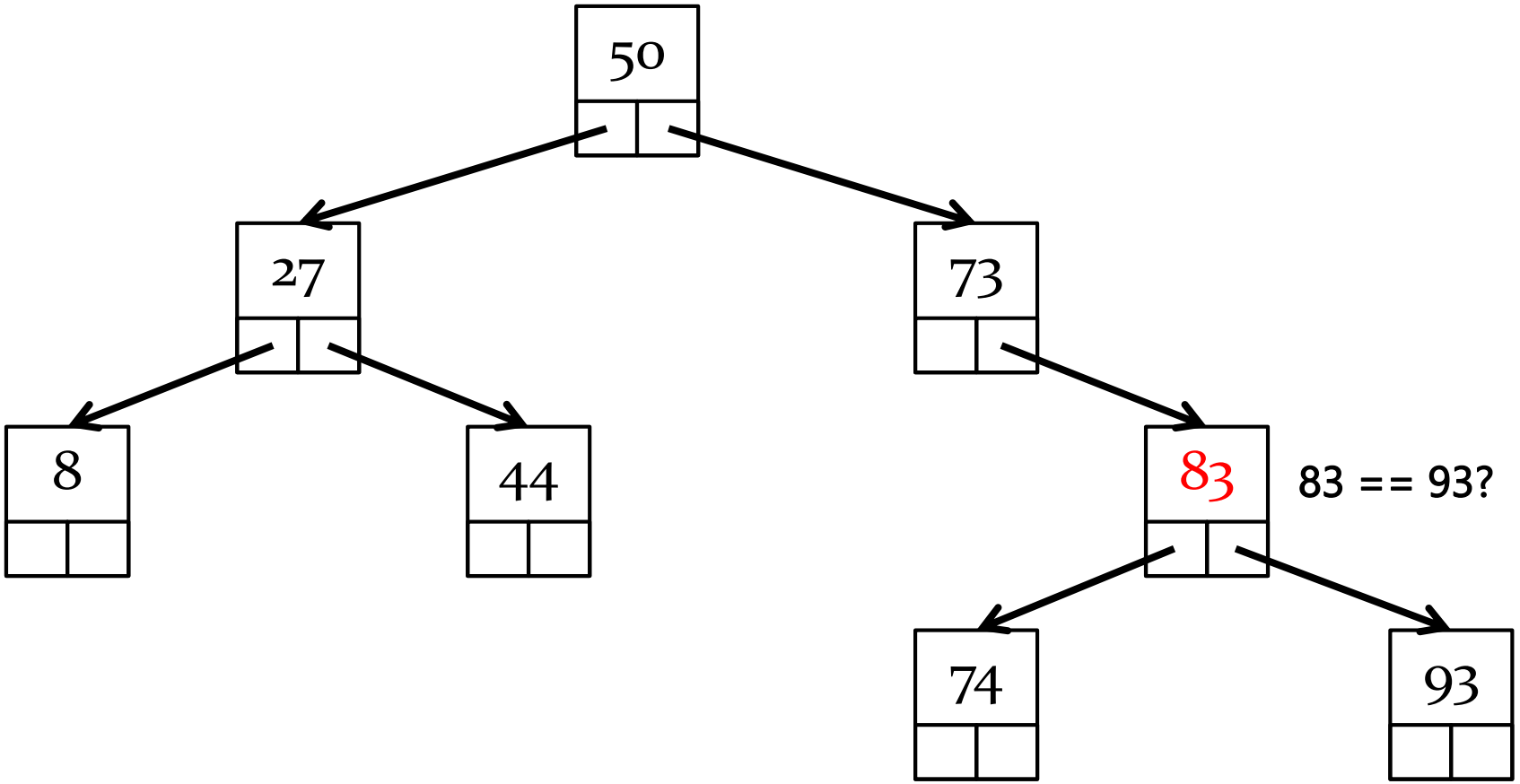

50 == 93?
50
27
73
8
44
83
74
93

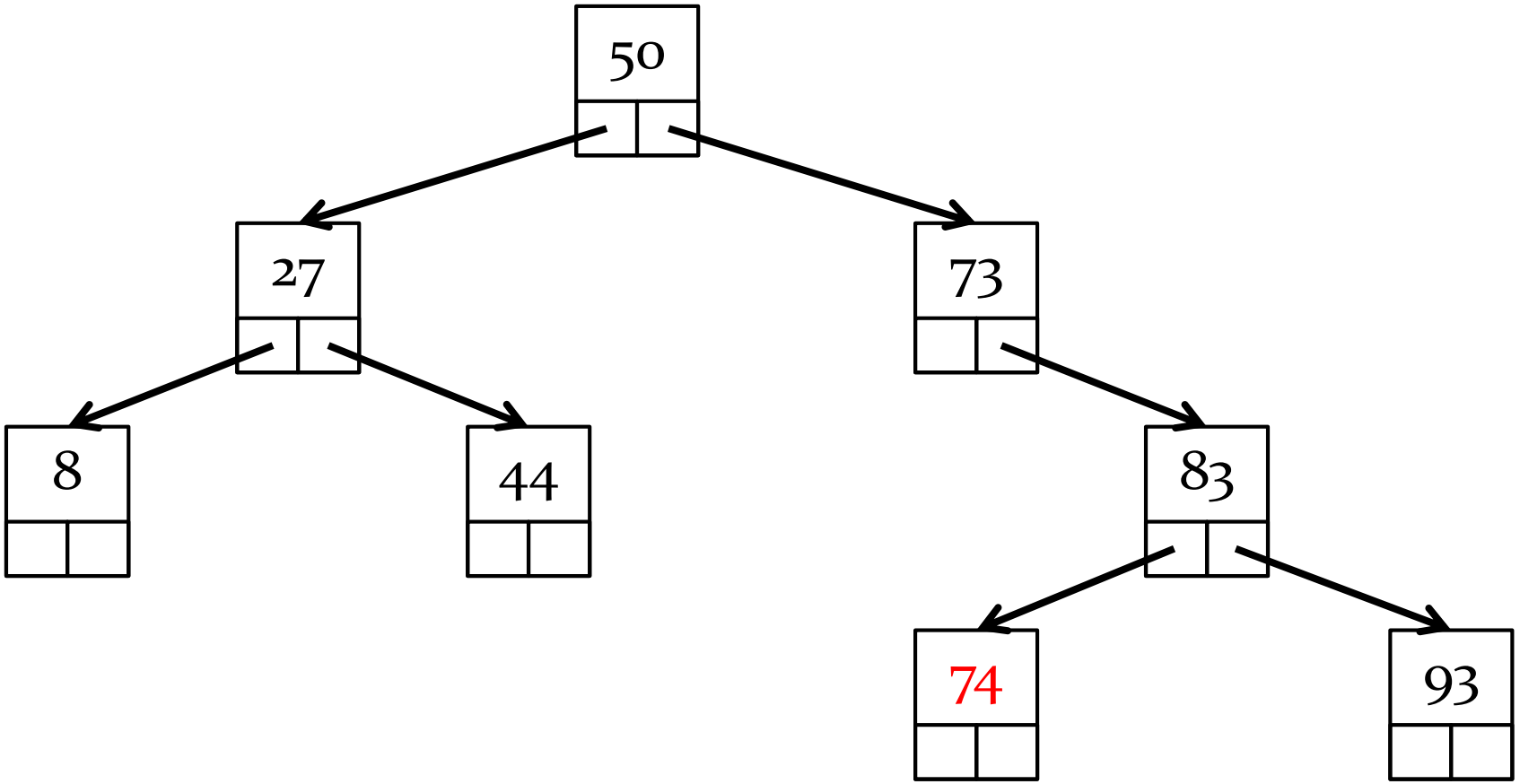

50
27 == 93?
27
73
8
44
83
74
93

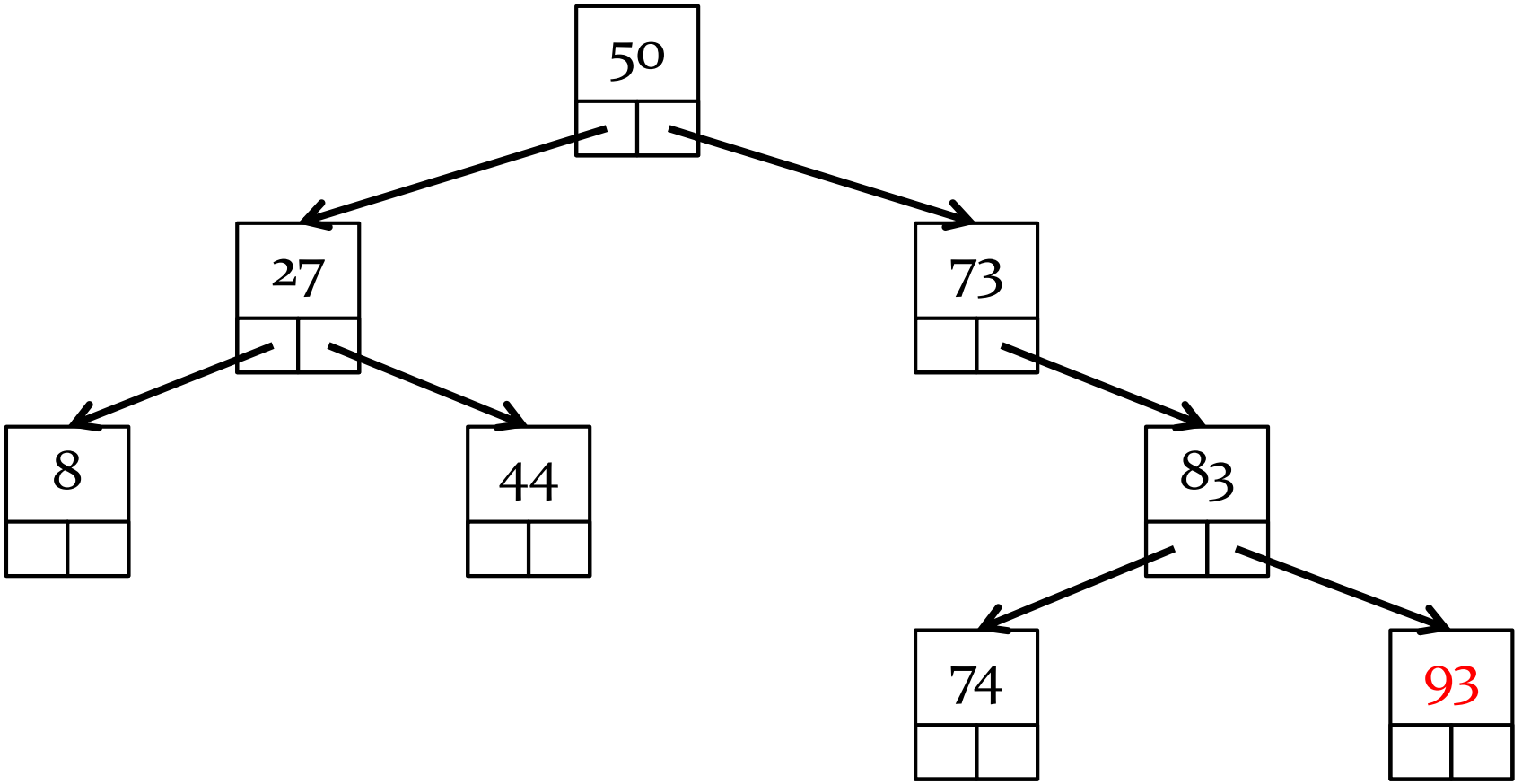

50
27
73
8
44
83
74
93
8 == 93?


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


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

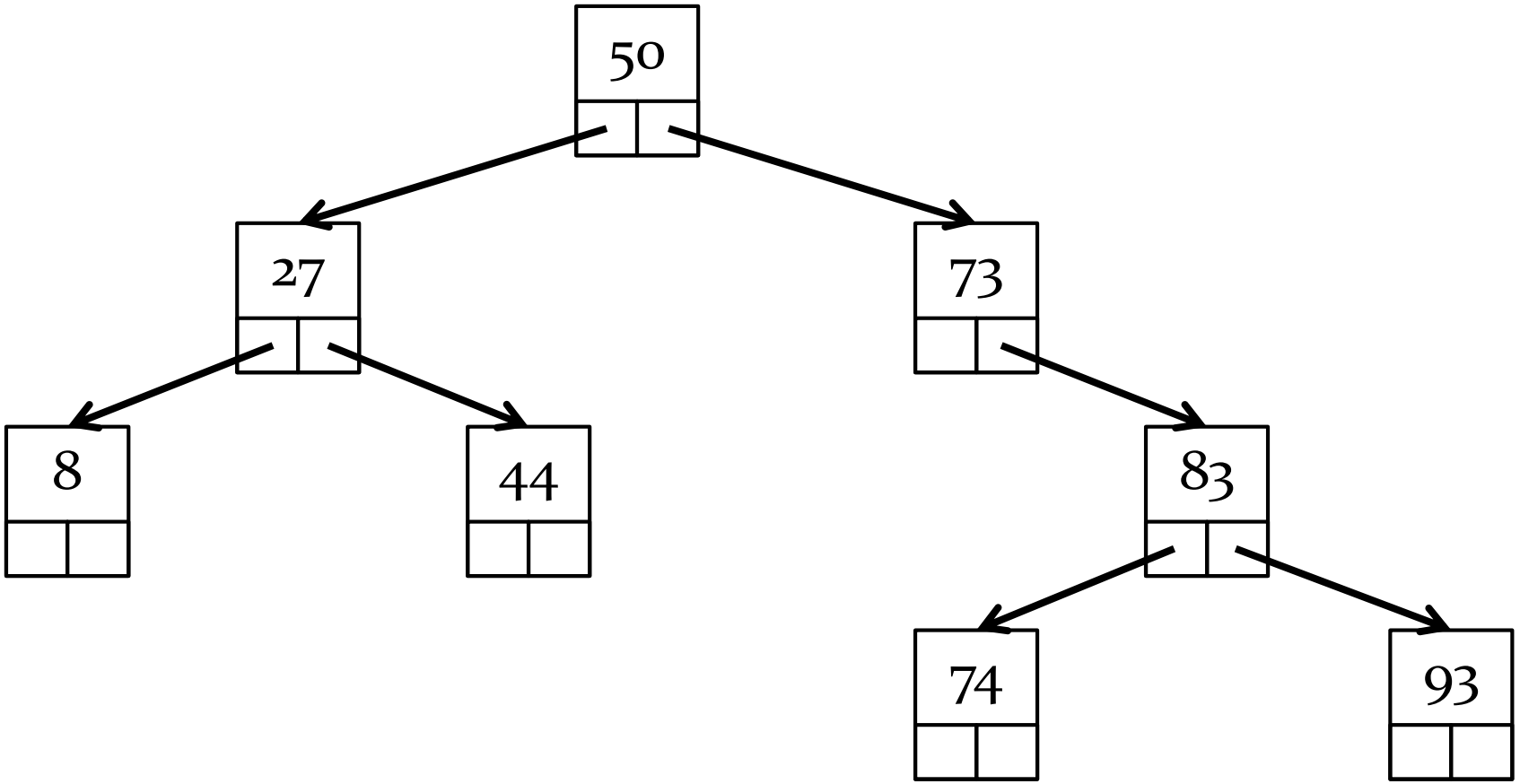


93 == 93?

Iteration or Traversal

- visiting every element of the tree can also be done recursively
- 3 possibilities based on when a node is visited

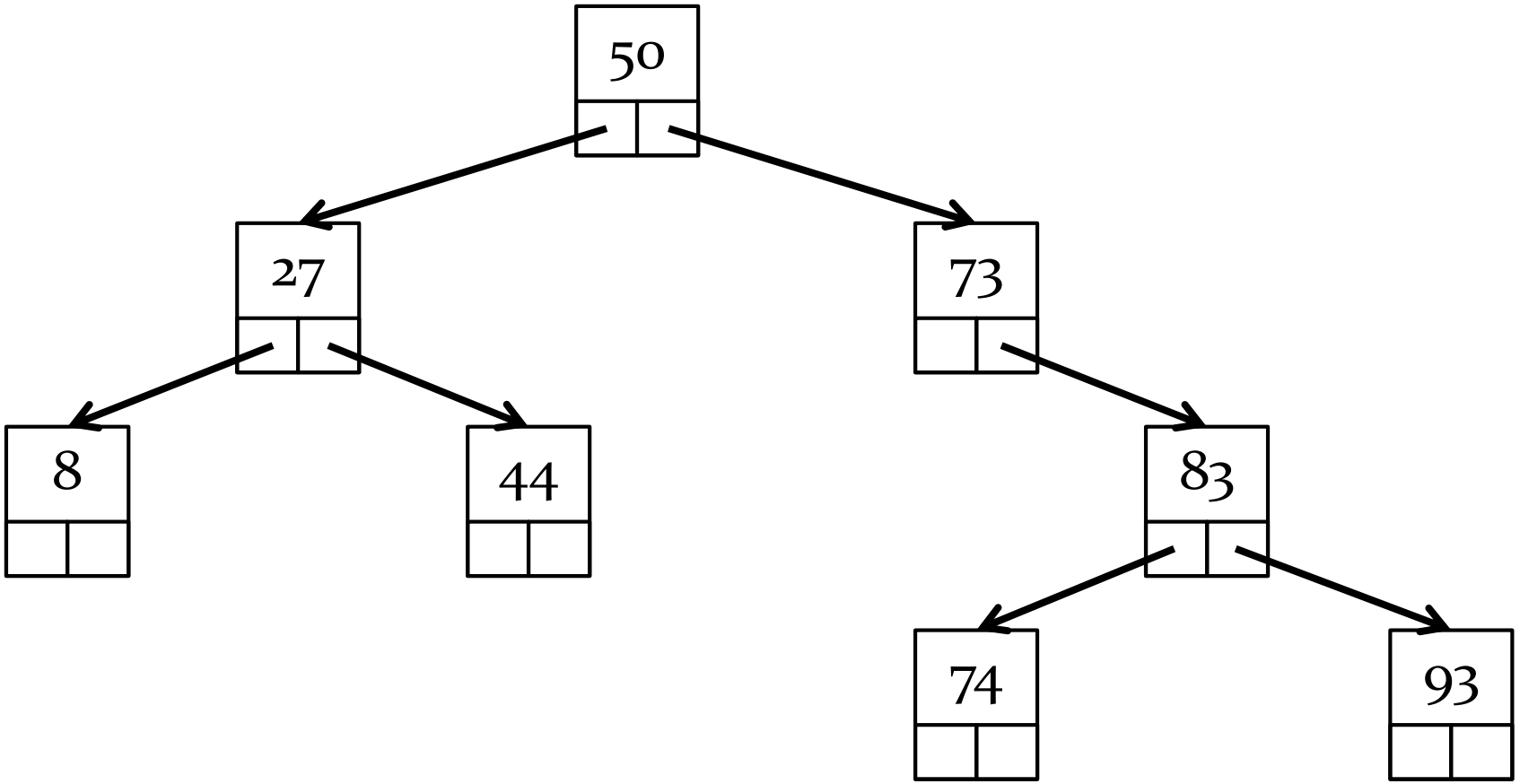
1. inorder
   - recursively traverse the left subtree,
   - then visit the node,
   - then recursively traverse the right subtree



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

Iteration or Traversal

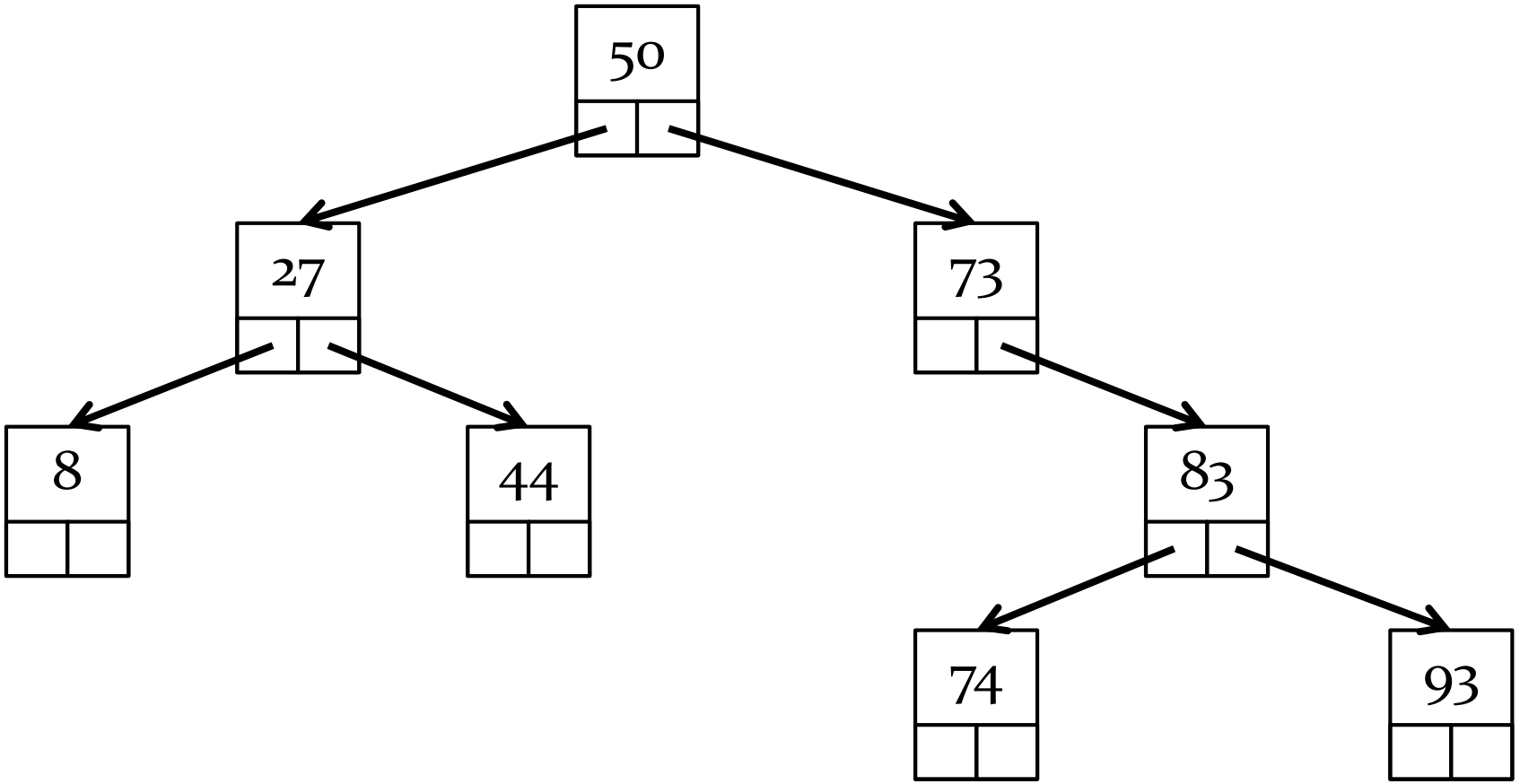
2. preorder
   - visit the node,
   - then recursively traverse the left subtree,
   - then recursively traverse the right subtree



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

Iteration or Traversal

- postorder
  - recursively traverse the left subtree,
  - then recursively traverse the right subtree,
  - then visit the node



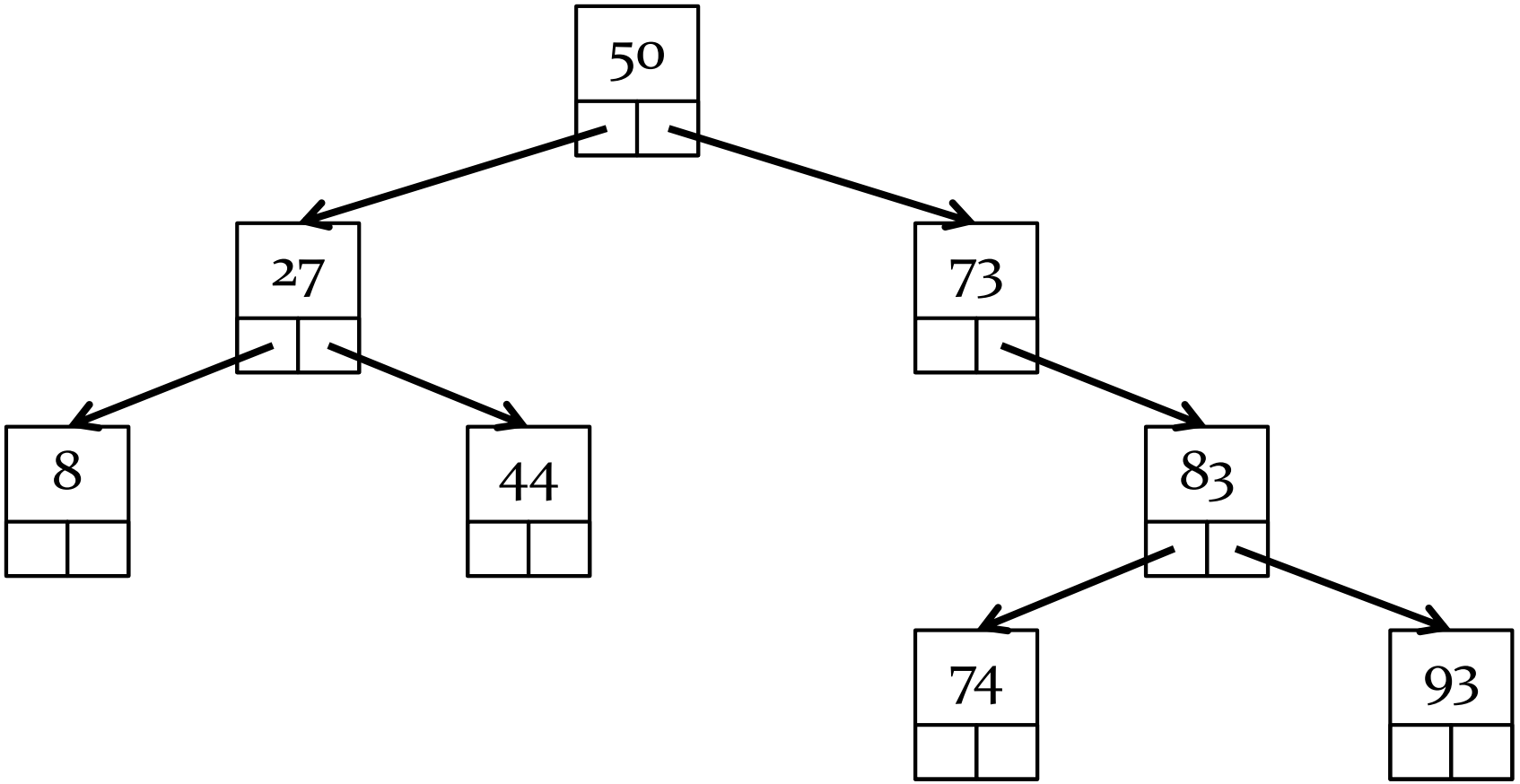
postorder: 8, 44, 27, 74, 93, 83, 73, 50

Iteration or Traversal

- what kind of traversal is **`contains`**?

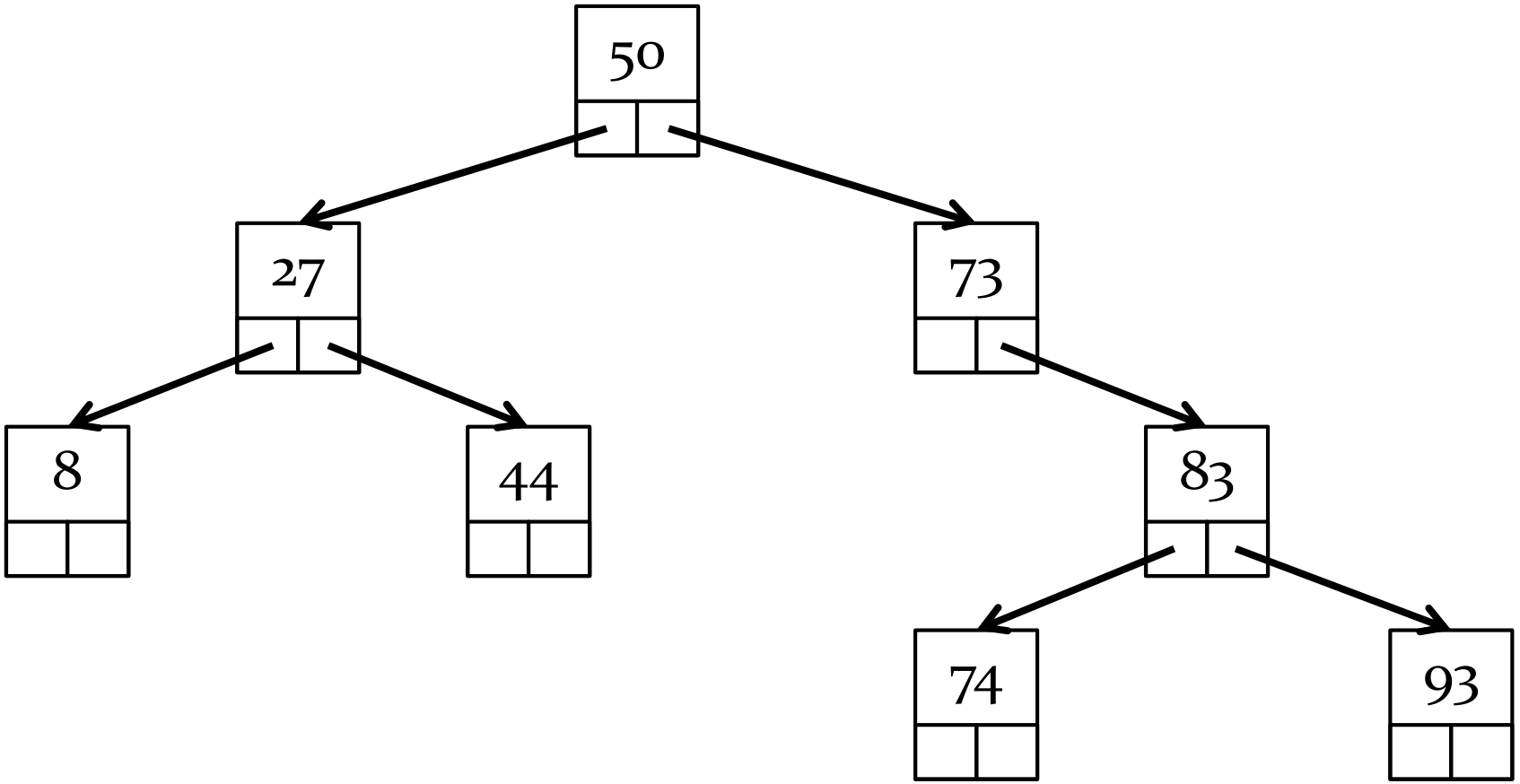
Iteration or Traversal

- the previous three tree traversals are all depth-first traversals
  - called depth first because for any node you traverse the entire left subtree before traversing the right subtree
- another possible traversal is to visit all nodes at the same level before continuing on the next lower level
  - called breadth first search



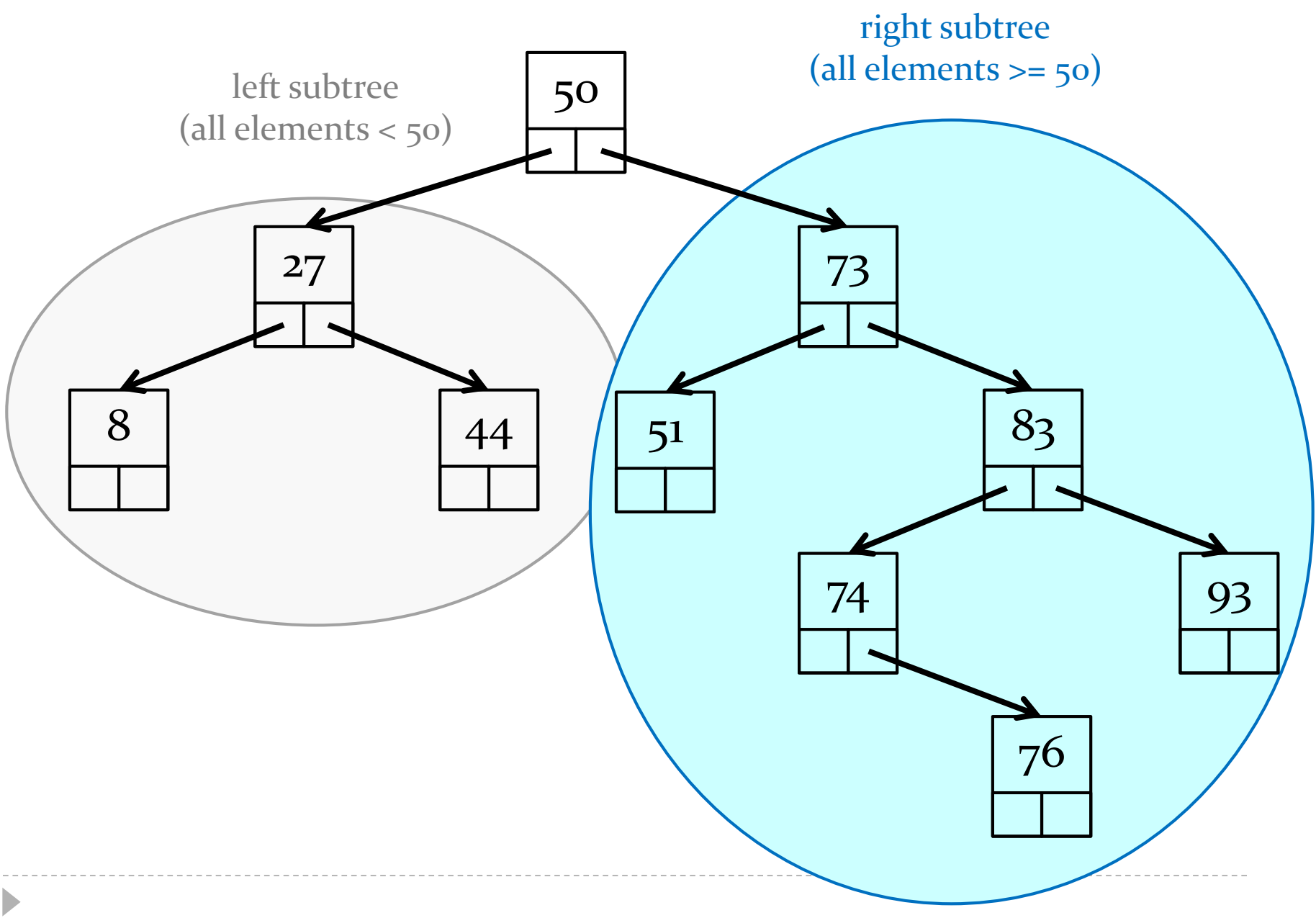
breadth first: 50, 27, 73, 8, 44, 83, 74, 93

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 or equal to 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

Binary Search Trees (BST)

- is every node of a BST the root of a BST?

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<E>> {
```

E must implement **Comparable<E>**

"for every type **E** that can be compared to itself"

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<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 {
    // call recursive static method
    BinarySearchTree.add(element, null, this.root);
  }
}
```

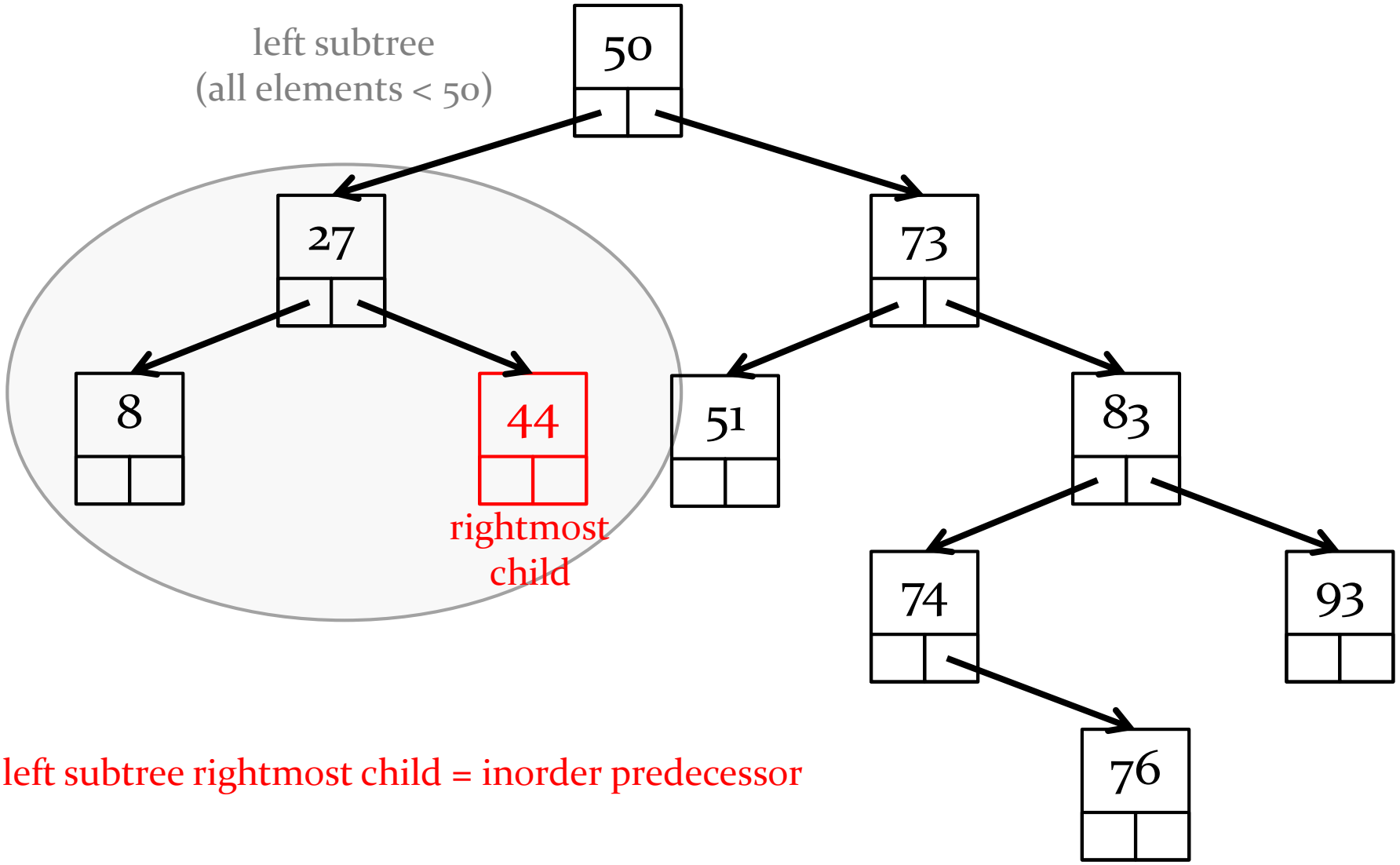
```
/**
 * Add an element to the tree with the specified root. The element is inserted into the
 * tree in a position that preserves the definition of a binary search tree.
 *
 * @param element      the element to add to the subtree
 * @param root         the root of the subtree
 */
private static <E extends Comparable<E>>
void add(E element, Node<E> root) {
  if (element.compareTo(root.data) < 0) {          // element belongs in the left subtree
    
  } else {                                         // element belongs in the right subtree
    
  }
}
```

```java
/**
 * Add an element to the tree with the specified root. 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 root         the root of the subtree
 */
private static <E extends Comparable<E>>
void add(E element, Node<E> root) {
  if (element.compareTo(root.data) < 0) {          // element belongs in the left subtree
    if (root.left == null) {                       // is there no left subtree?
      root.left = new Node<E>(element);            // add the element as the new left child
    
    
    
  } else {                                         // element belongs in the right subtree
    if (root.right == null) {                      // is there no right subtree?
      root.right = new Node<E>(element);           // add the element as the new right child
    
    
    
  }
}
```

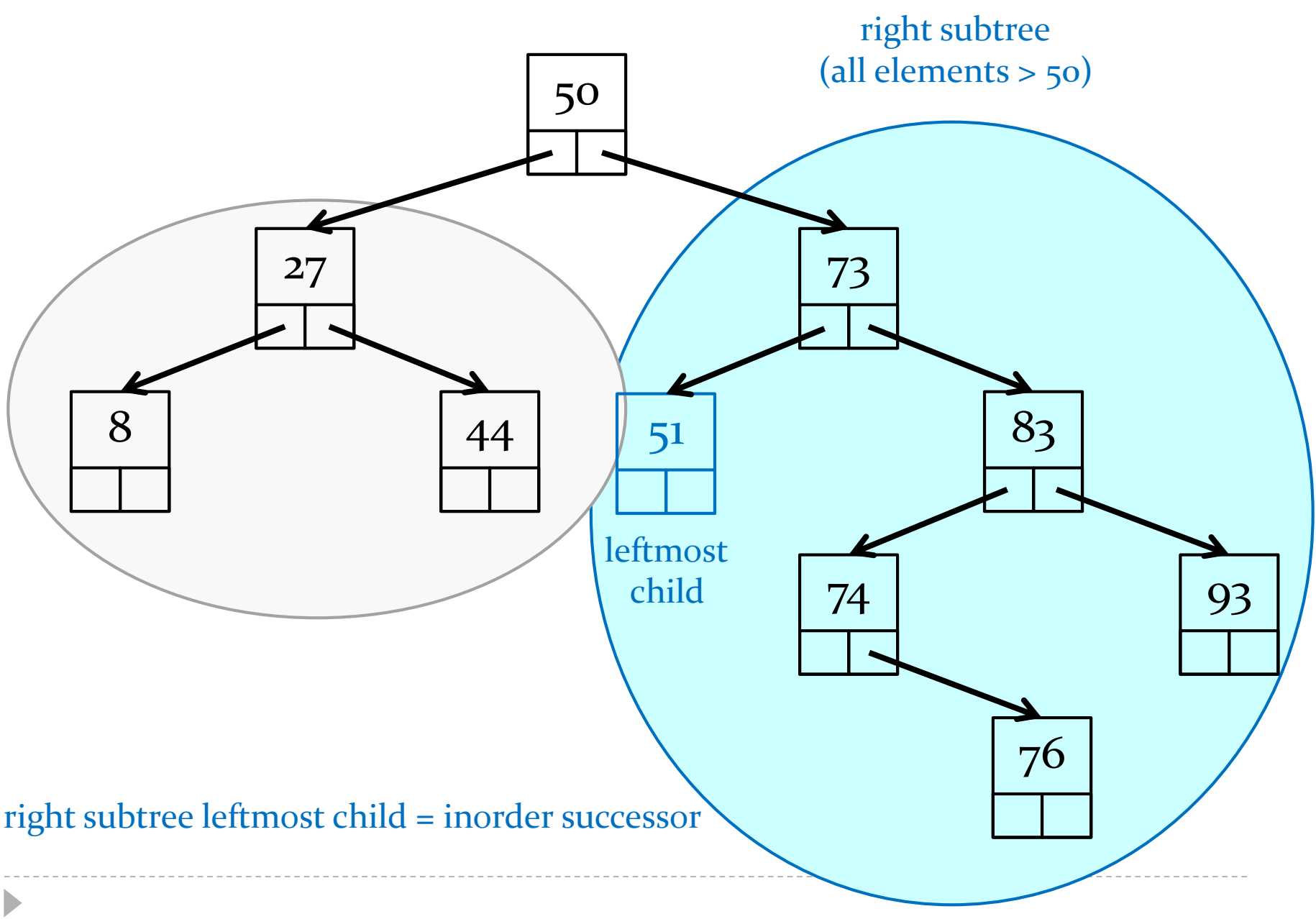
```java
/**
 * Add an element to the tree with the specified root. The element is inserted into the
 * tree in a position that preserves the definition of a binary search tree.
 *
 * @param element      the element to add to the subtree
 * @param root         the root of the subtree
 */
private static <E extends Comparable<E>>
void add(E element, Node<E> root) {
  if (element.compareTo(root.data) < 0) {          // element belongs in the left subtree
    if (root.left == null) {                       // is there no left subtree?
      root.left = new Node<E>(element);            // add the element as the new left child
    } else {
      BinarySearchTree.add(element, root.left);    // recursively add to the left subtree
    }
  } else {                                         // element belongs in the right subtree
    if (root.right == null) {                      // is there no right subtree?
      root.right = new Node<E>(element);           // add the element as the new right child
    } else {
      BinarySearchTree.add(element, root.right);   // recursively add to the right subtree
    }
  }
}
```

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



left subtree rightmost child = inorder predecessor



right subtree 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 root
 *            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> root) {
    base case?

    recursive case?
}
```

```java
/**
 * Find the node in a subtree that has the largest data element.
 *
 * @param root
 *            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> root) {
    base case?

    recursive case?
}
```

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

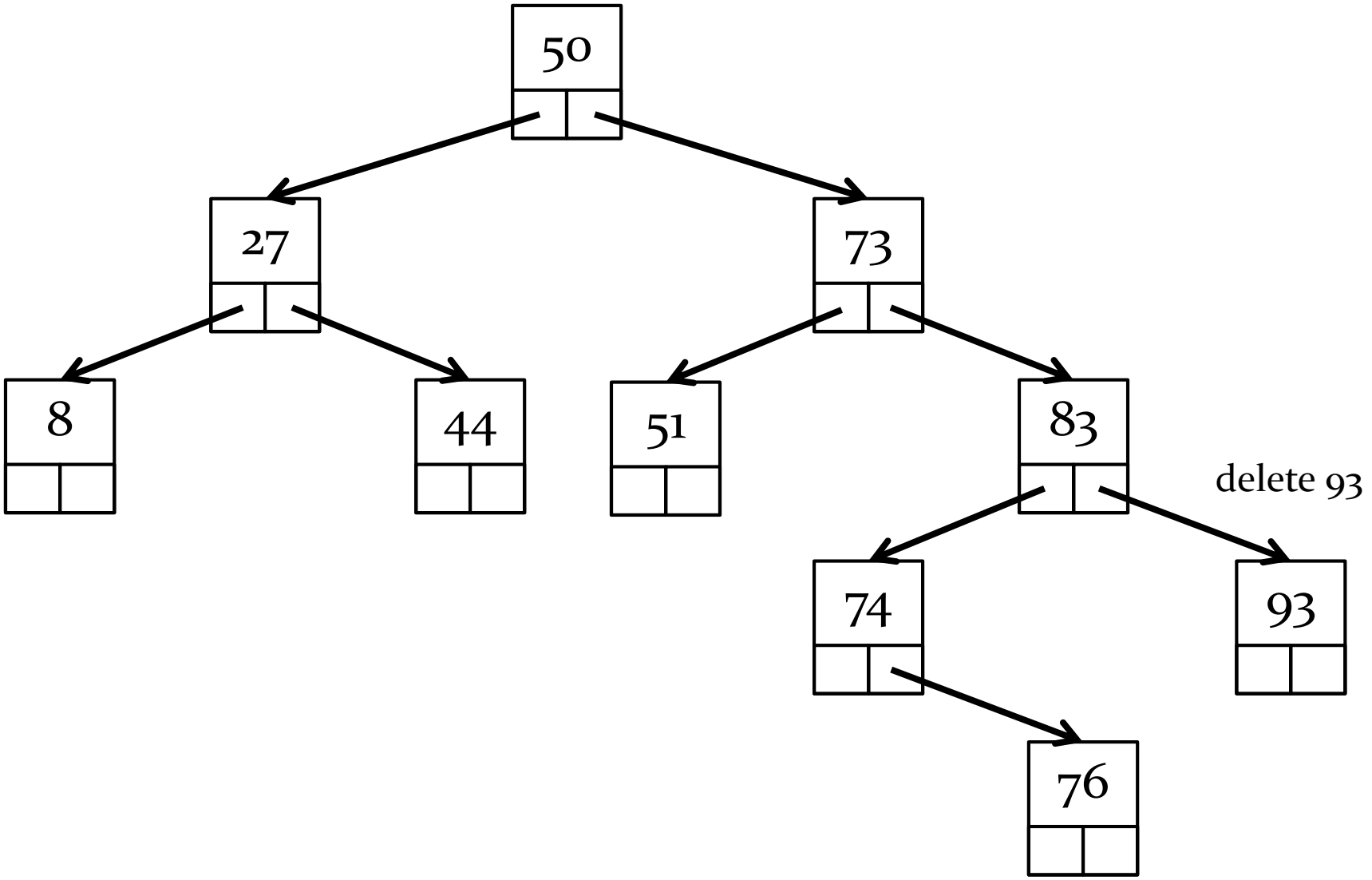
Deletion from a BST

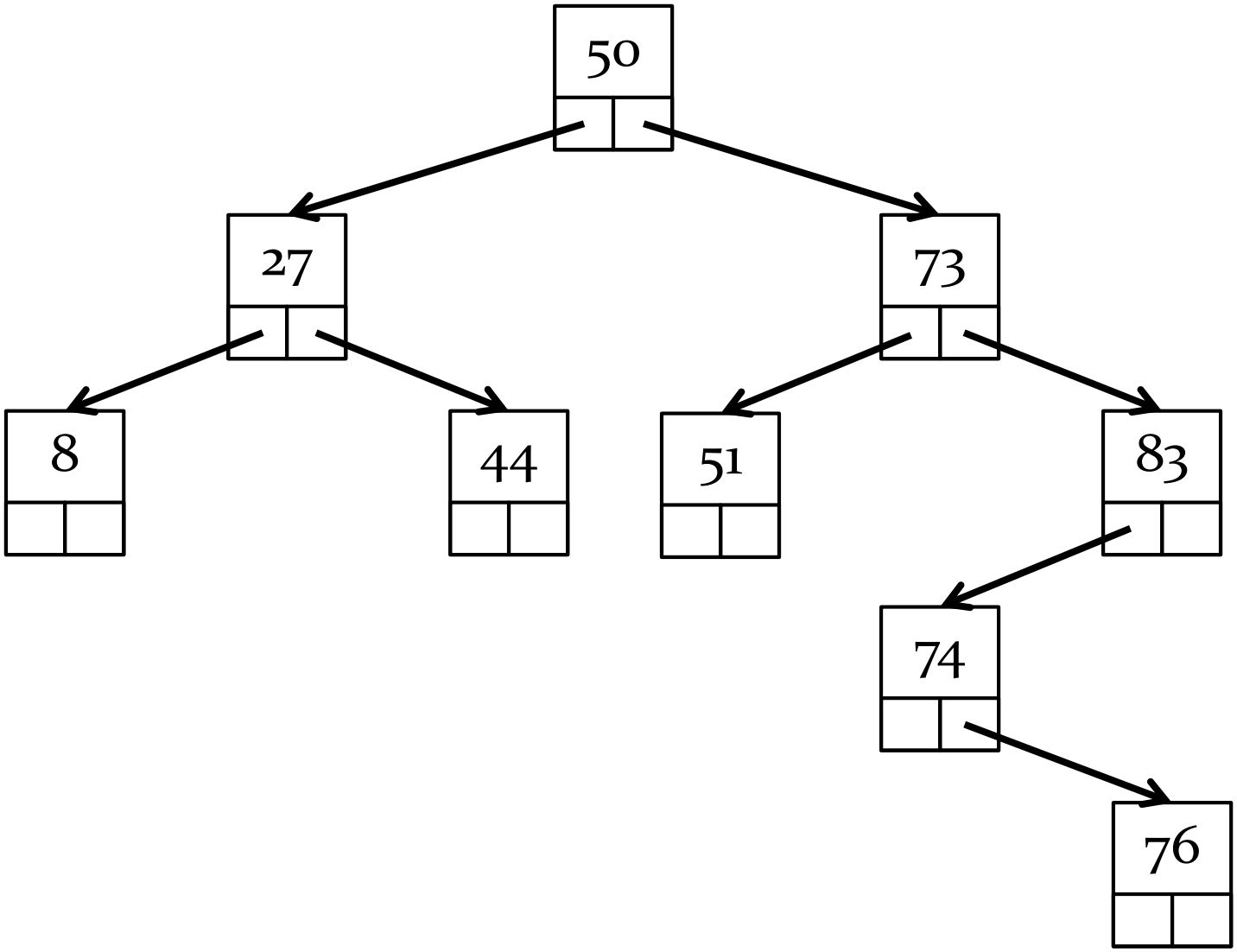
- to delete a node in a BST there are 3 cases to consider:
  1. deleting a leaf node
  2. deleting a node with one child
  3. deleting a node with two children

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

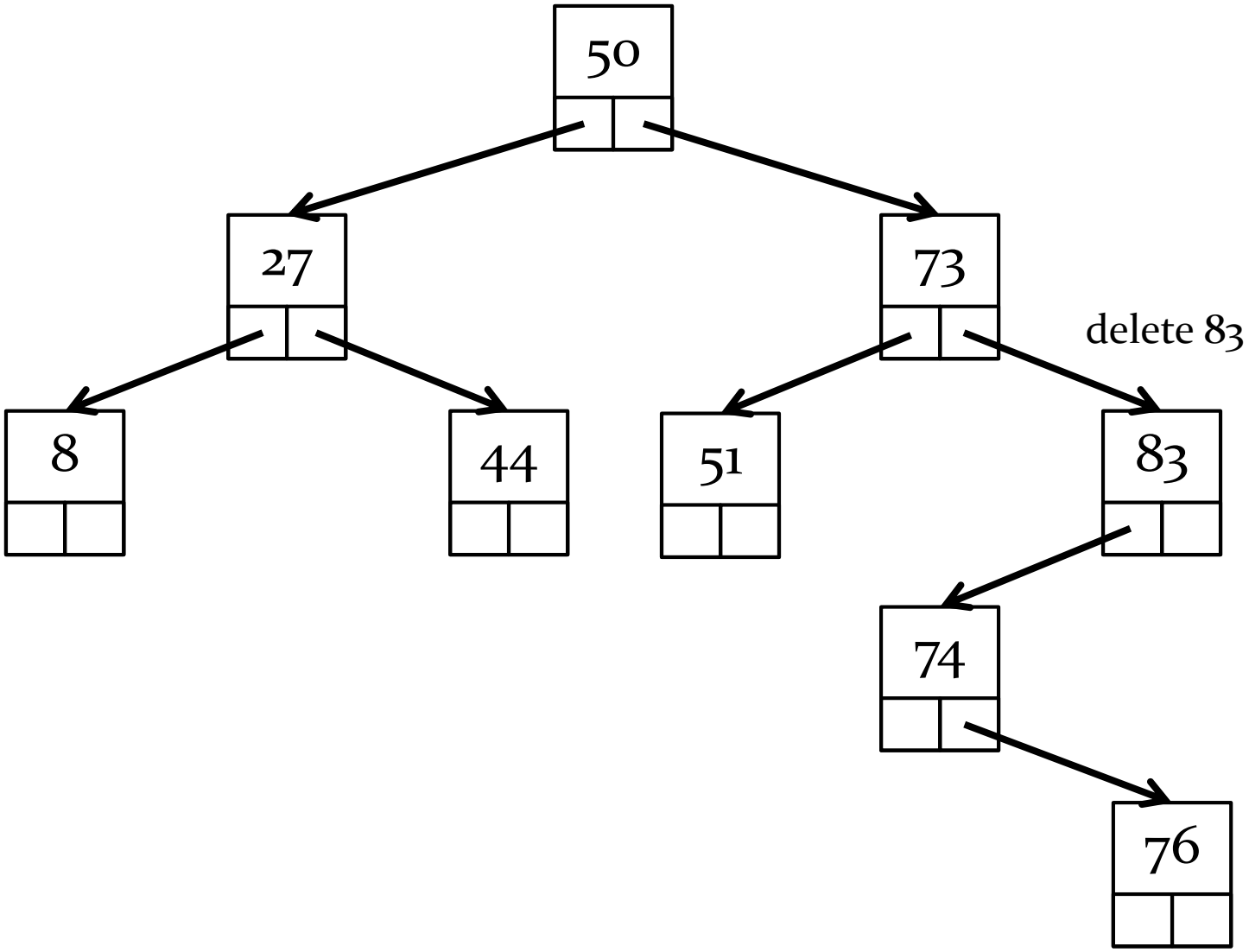

50
27
73
8
44
51
83
delete 93
74
93
76

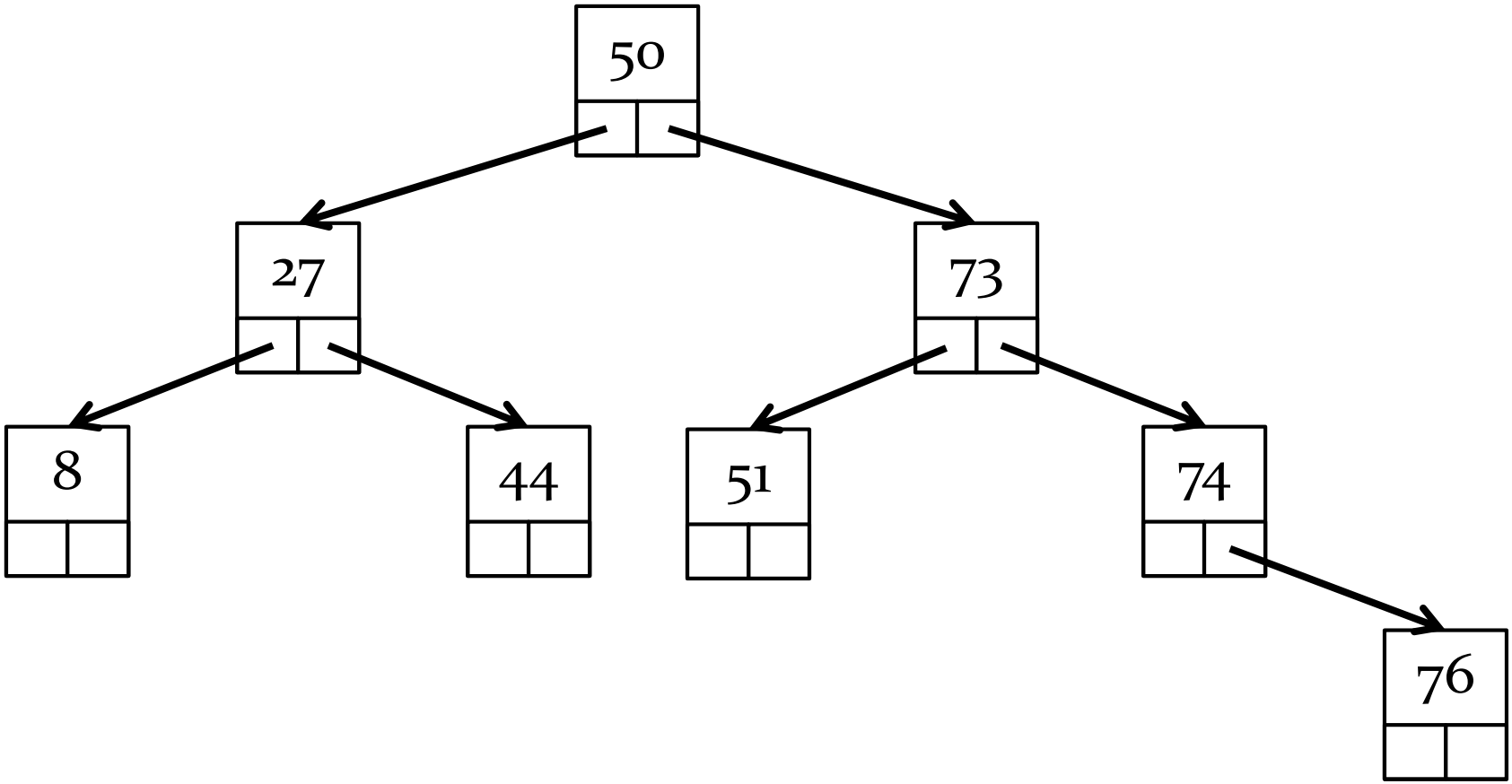

50
27
73
8
44
51
83
74
76

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

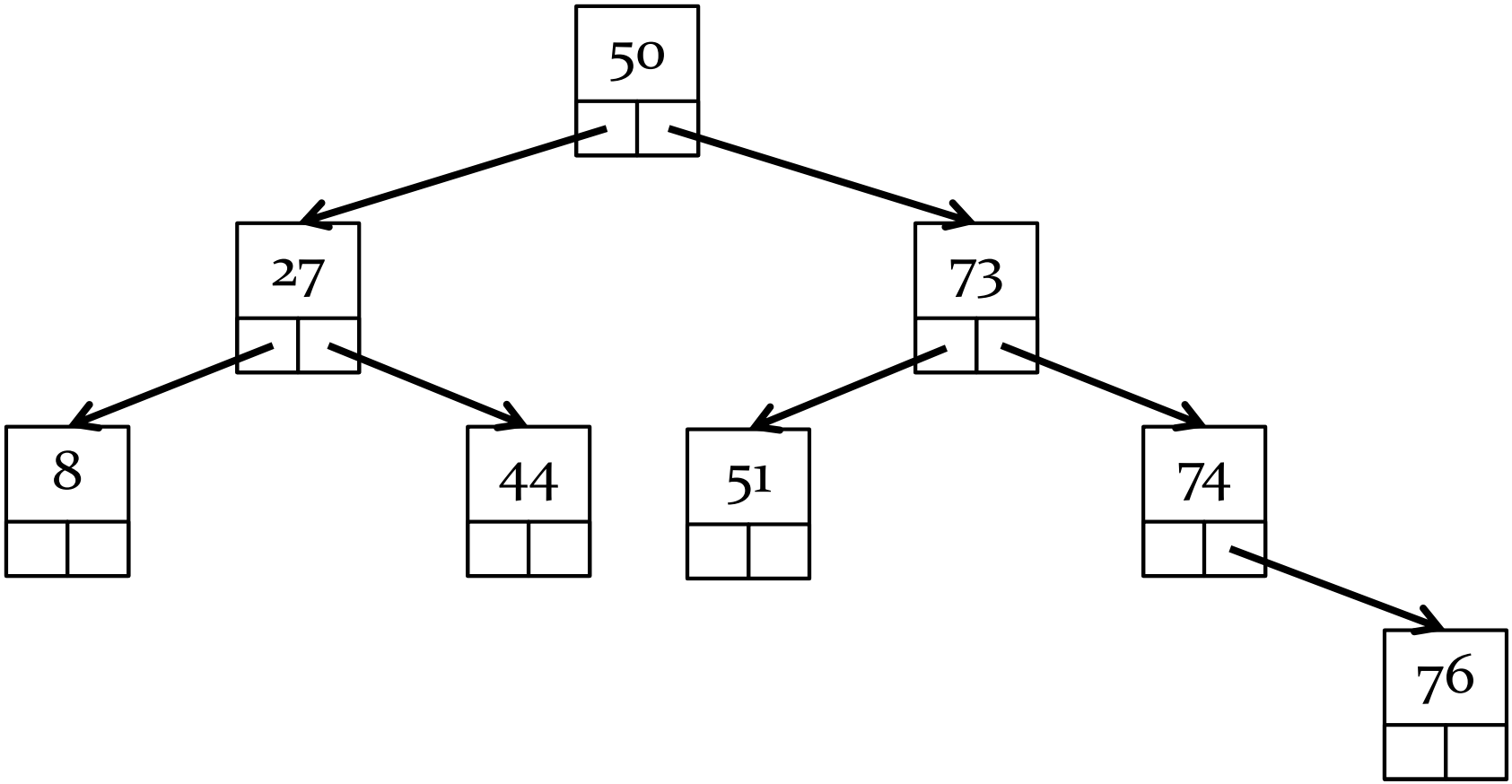
- deleting a node with two children is a little trickier
  - can you see how to do it?

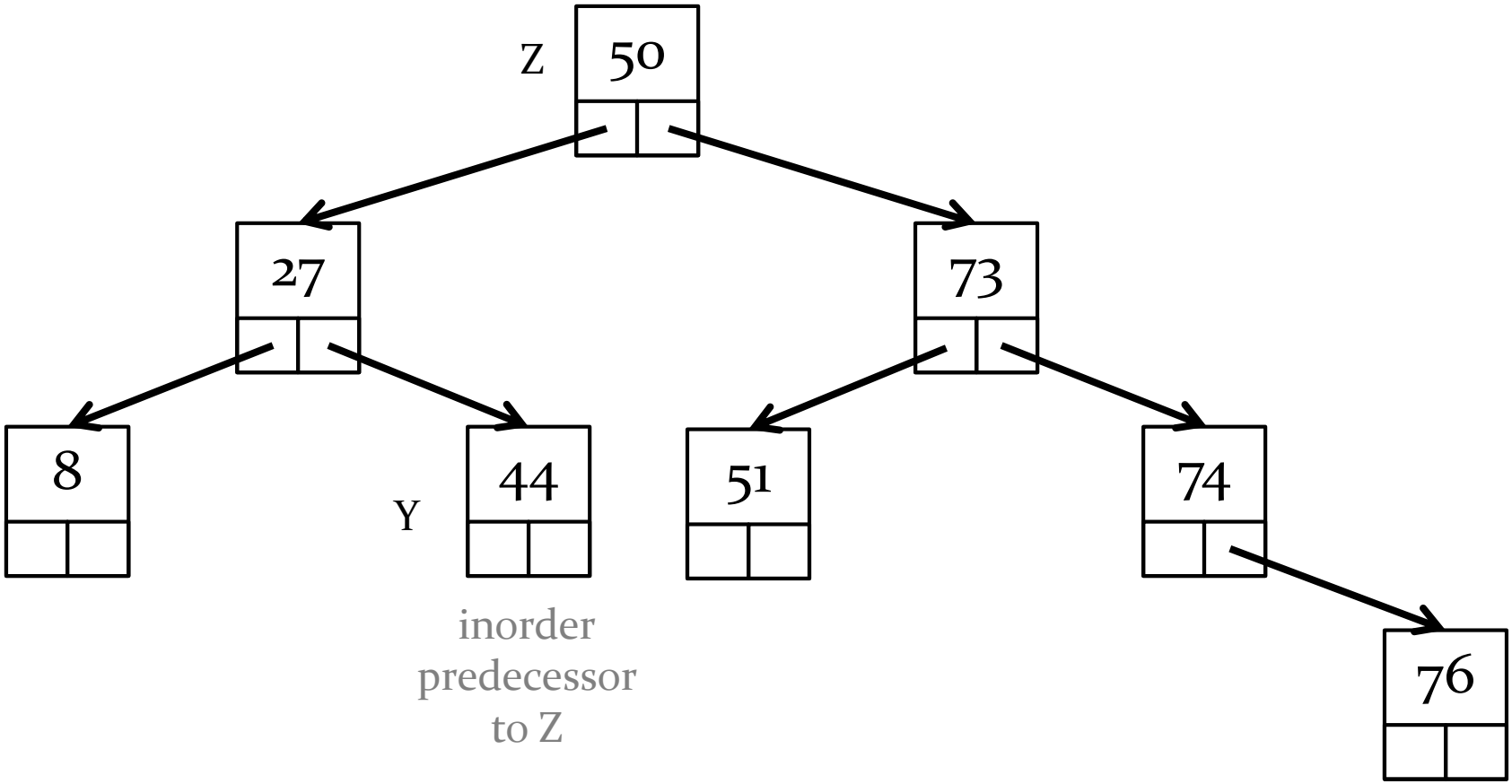
Deleting a Node with Two Children

- 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

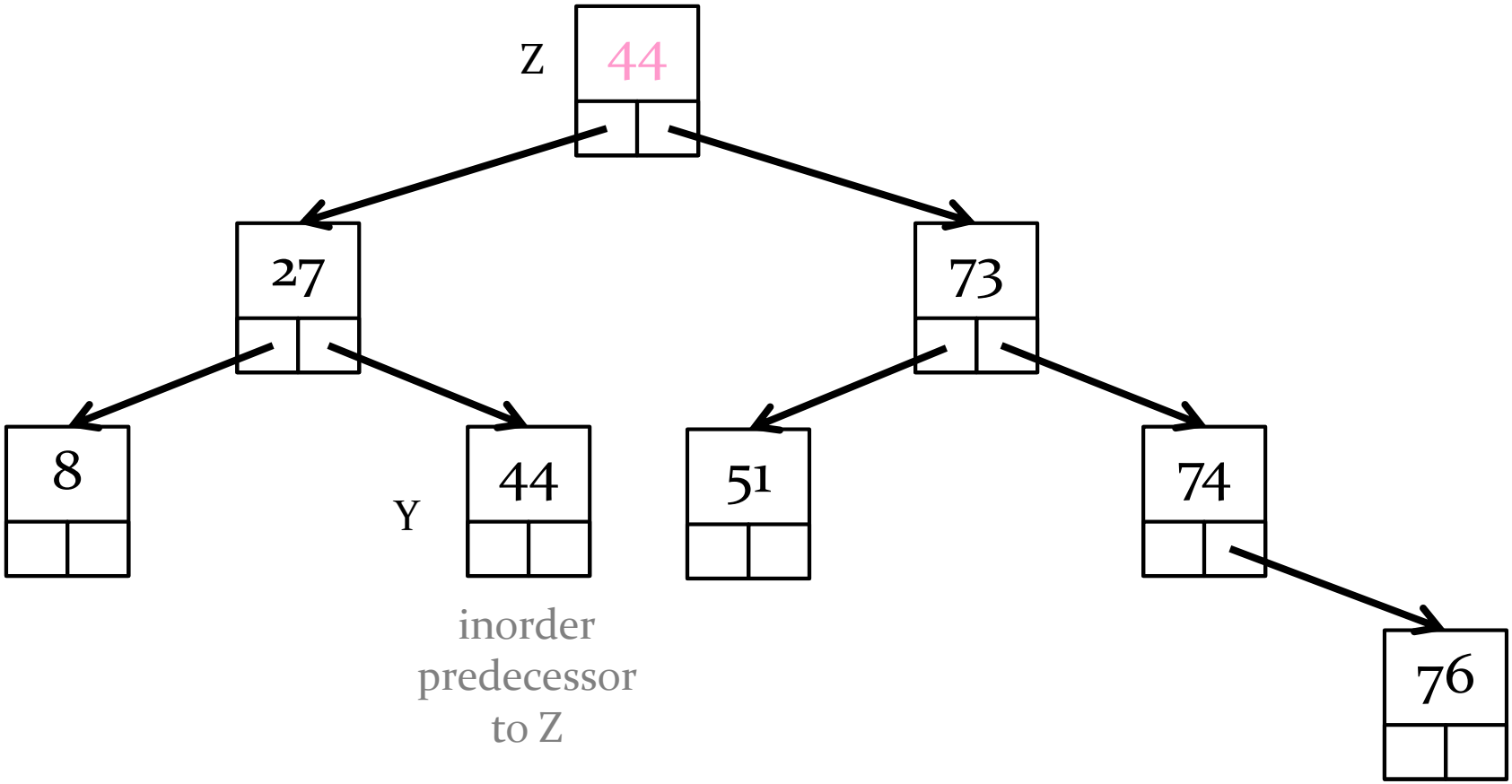
- e.g., delete 50

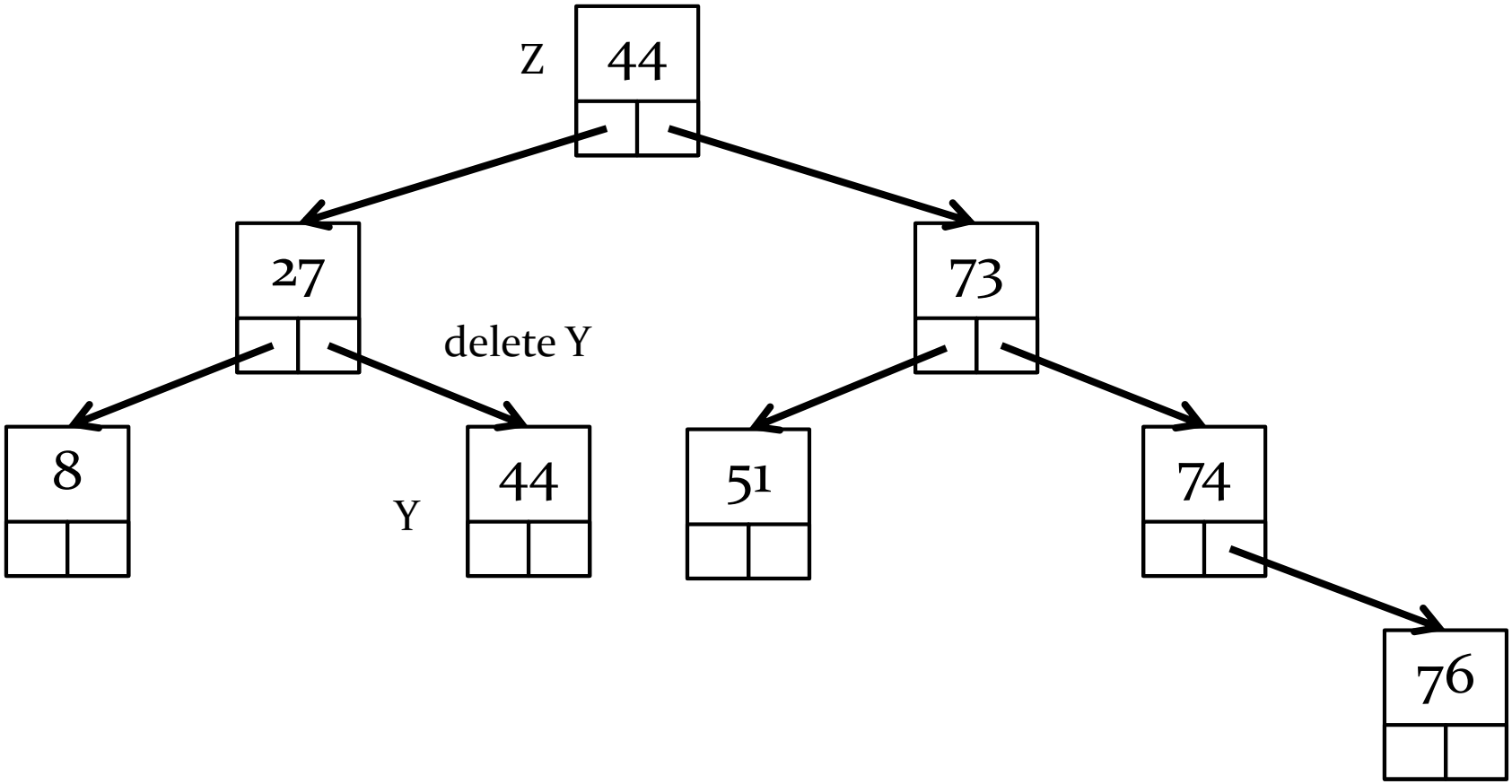
delete 50 using inorder predecessor

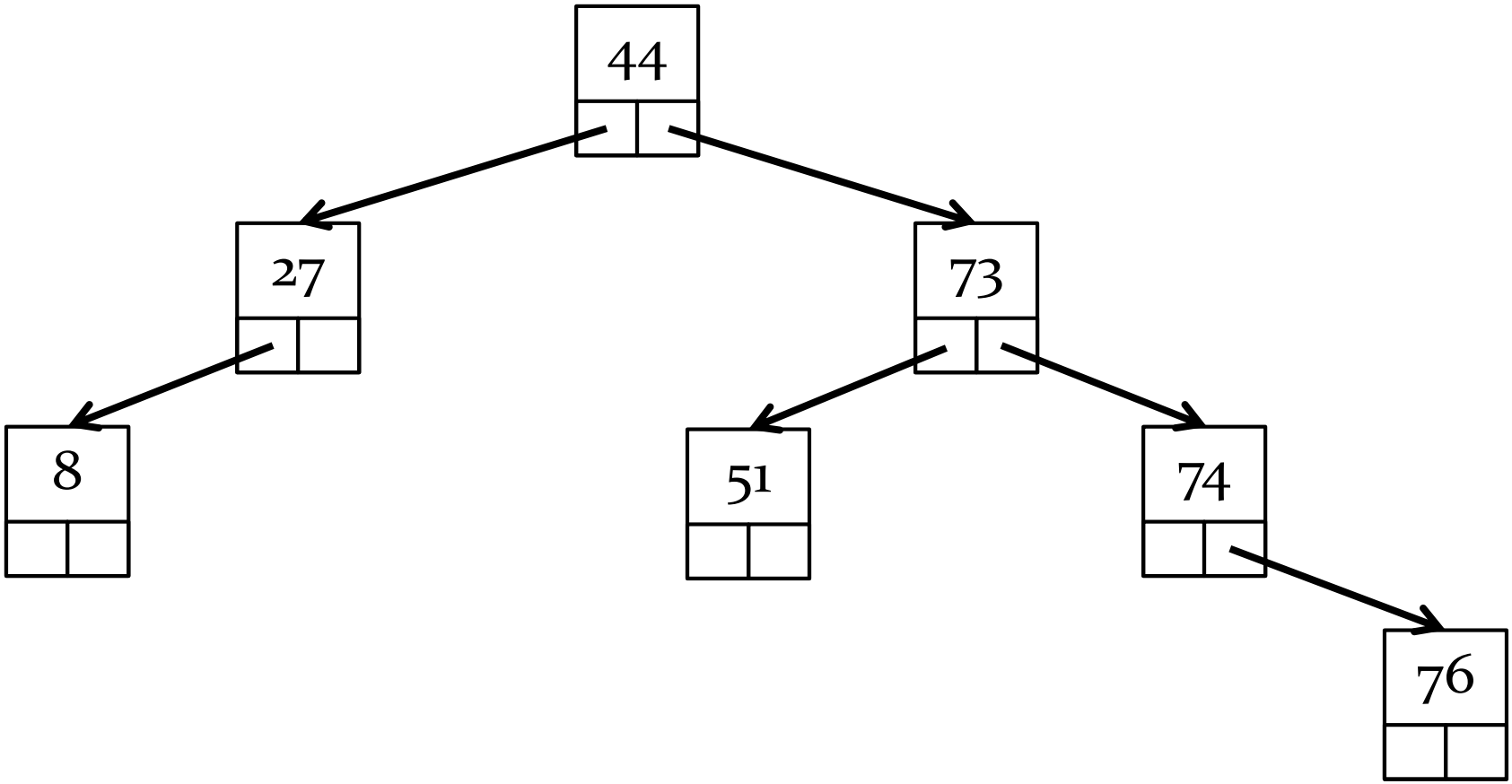



Z
50
27
73
8
Y
44
51
74
76
inorder
predecessor
to Z

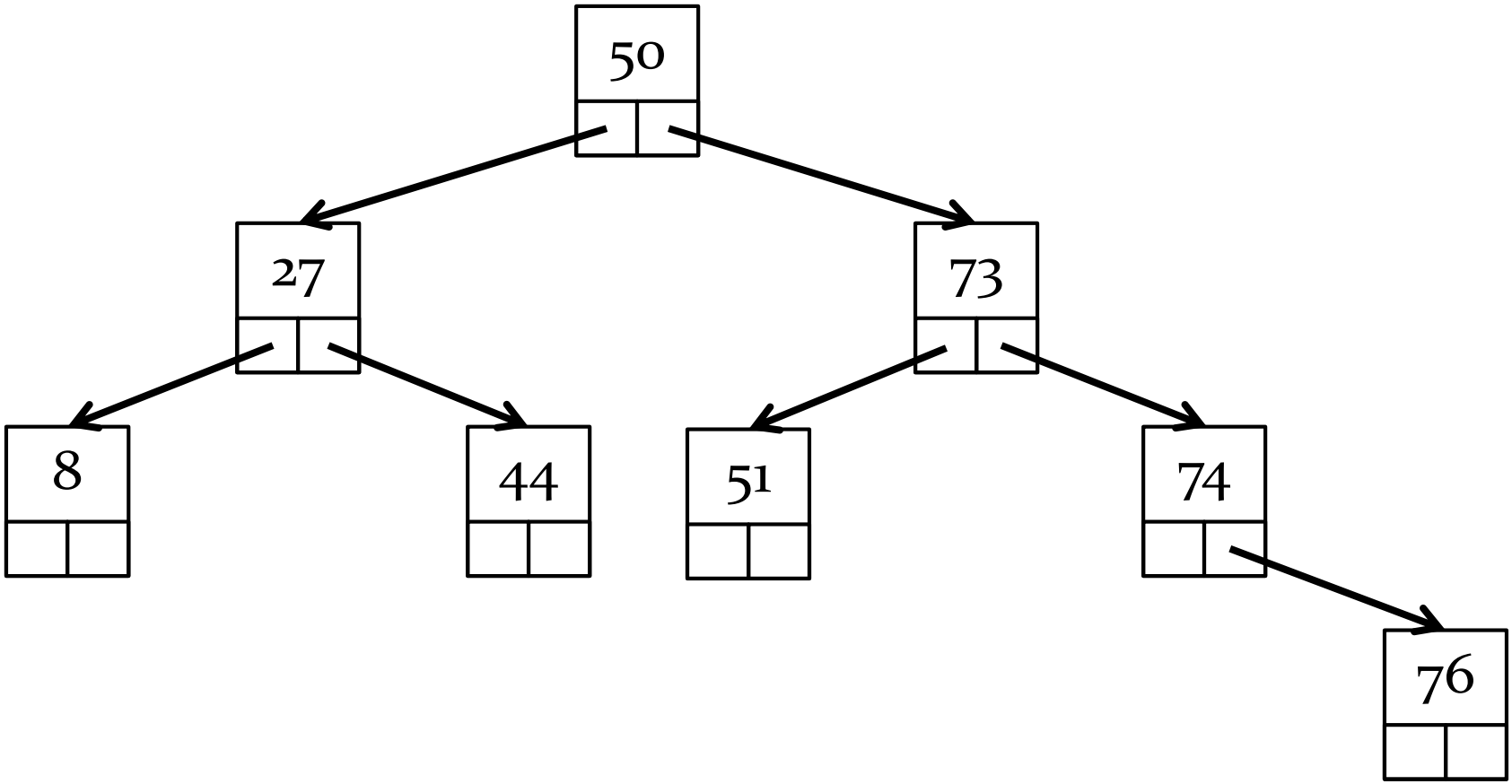
copy Y data to Z data

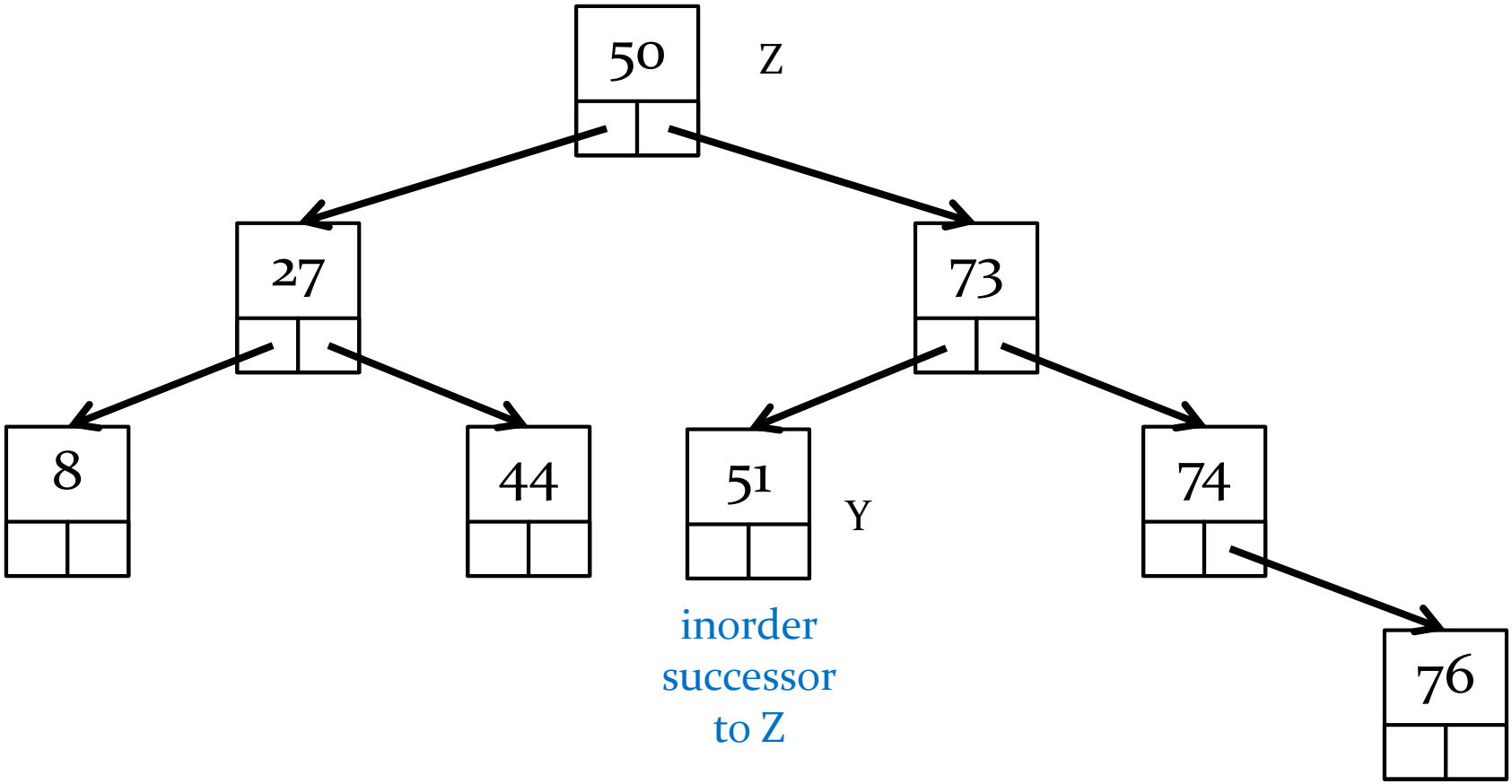


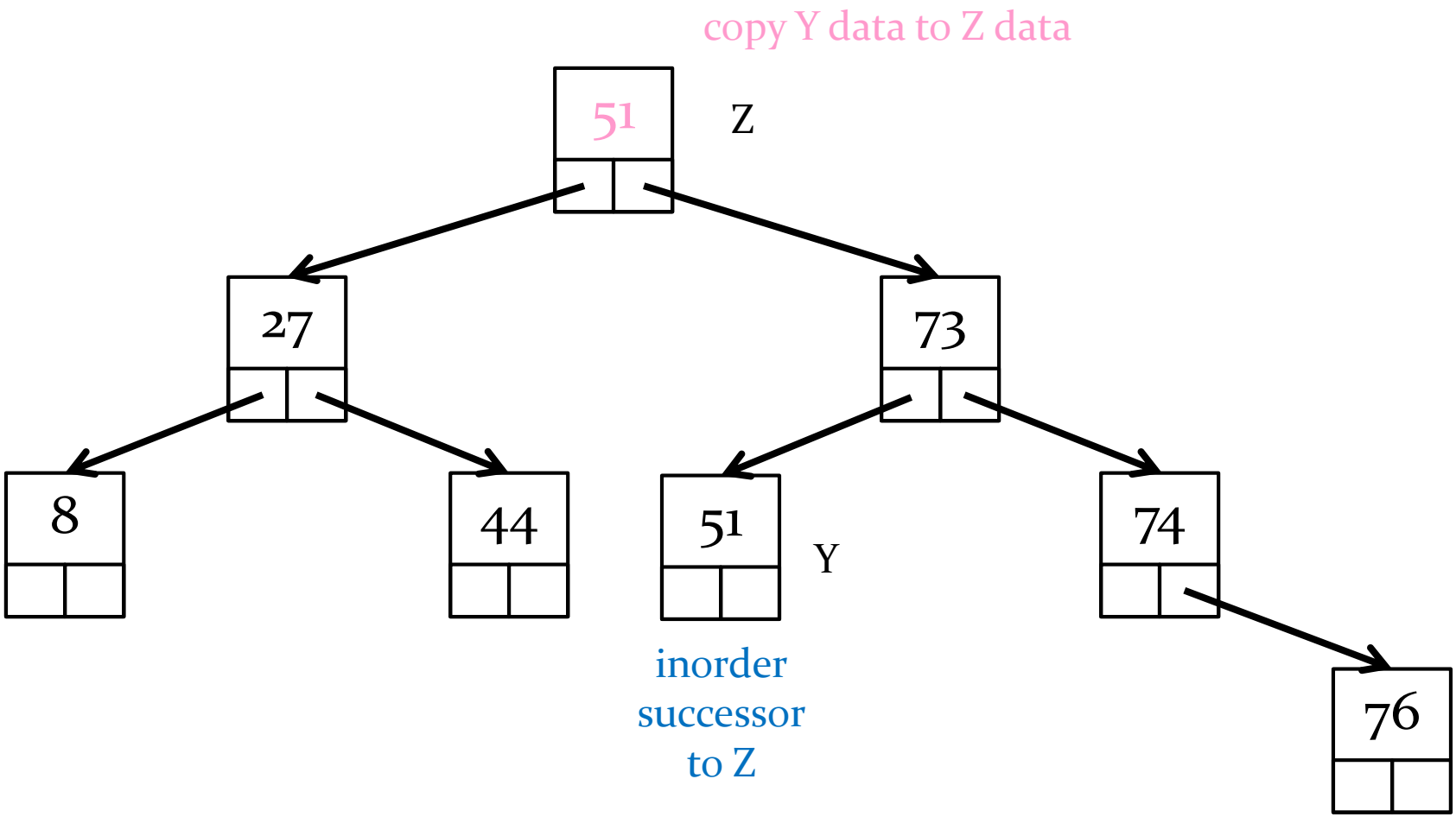

Z
44
27
delete Y
73
8
Y
44
51
74
76

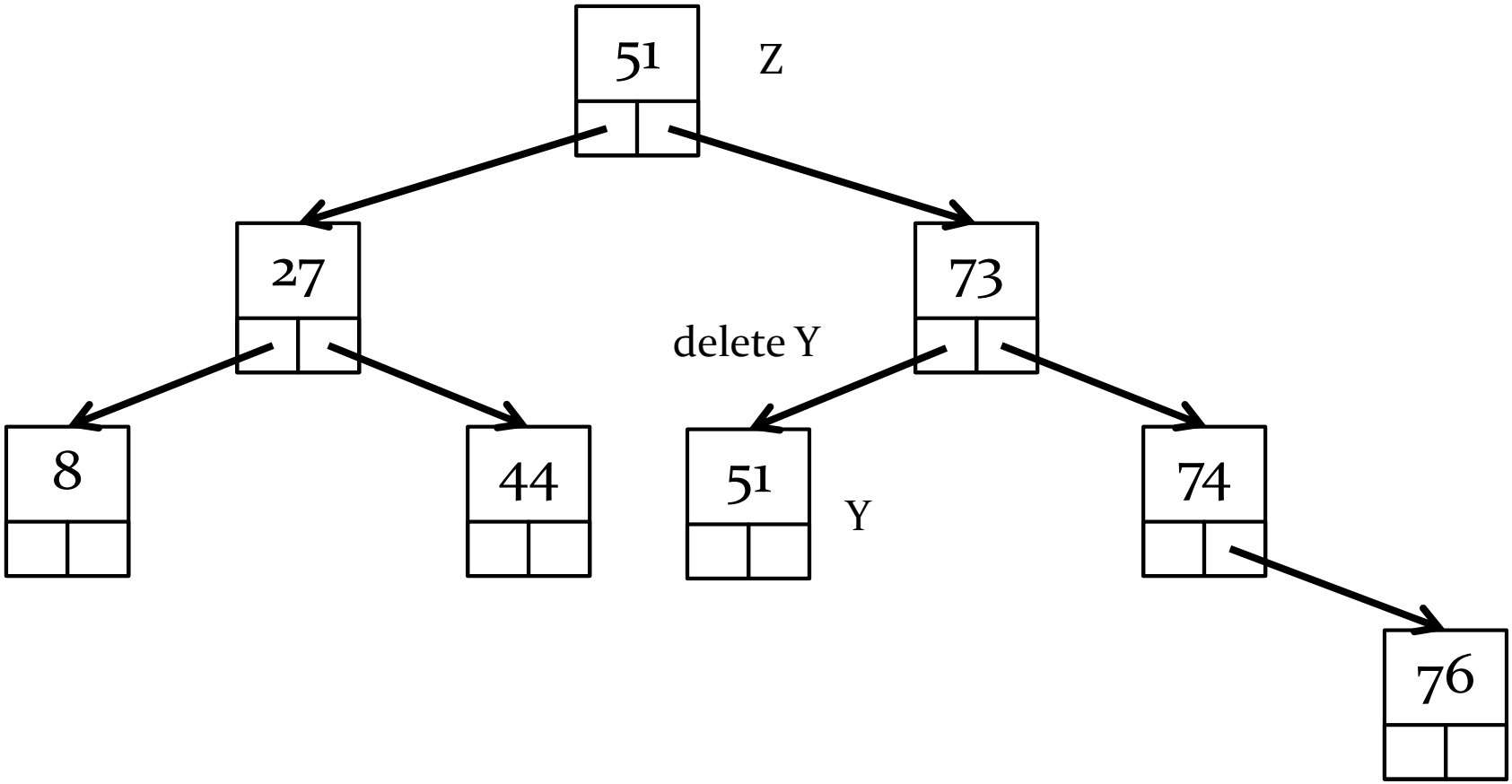

44
27
73
8
51
74
76

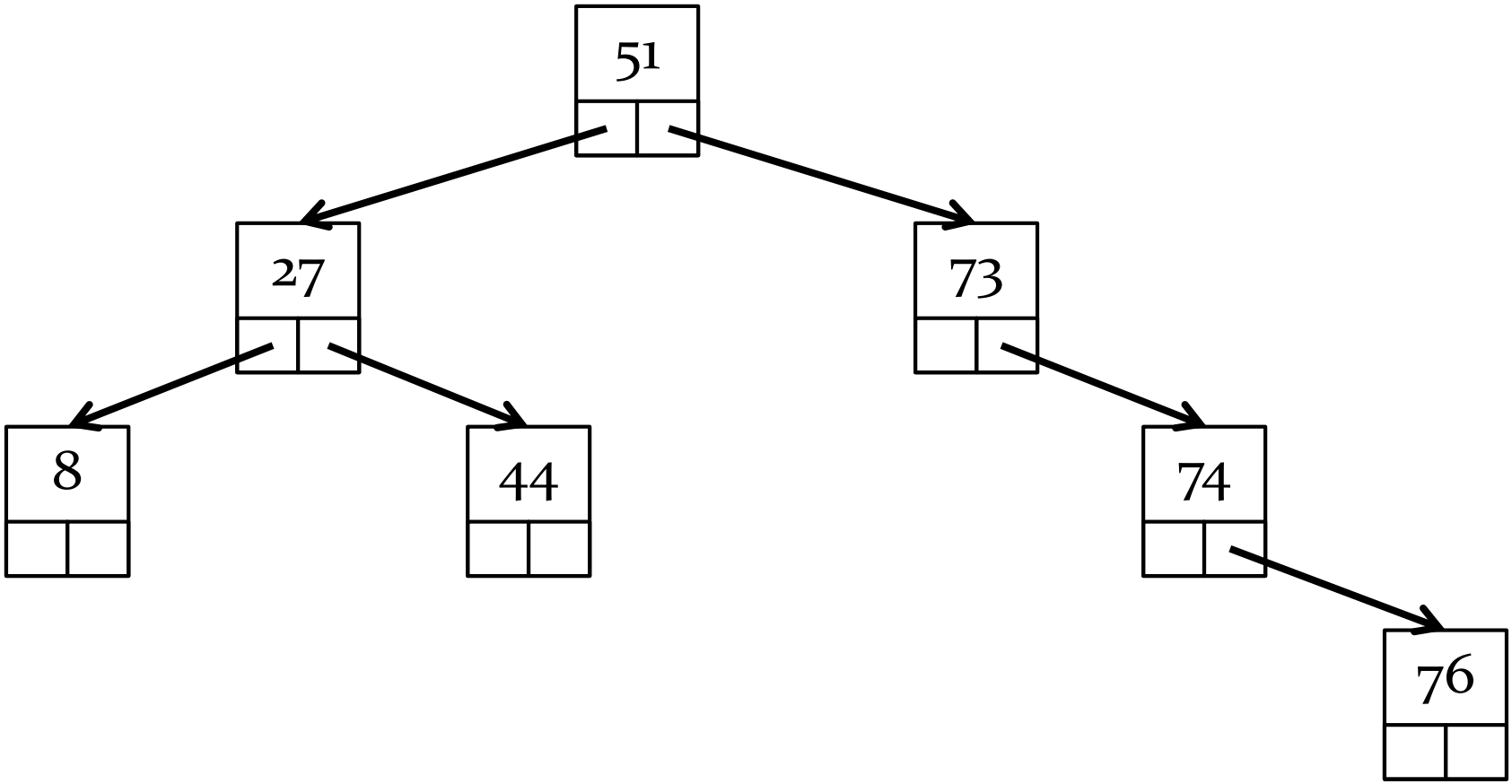
delete 50 using inorder successor




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