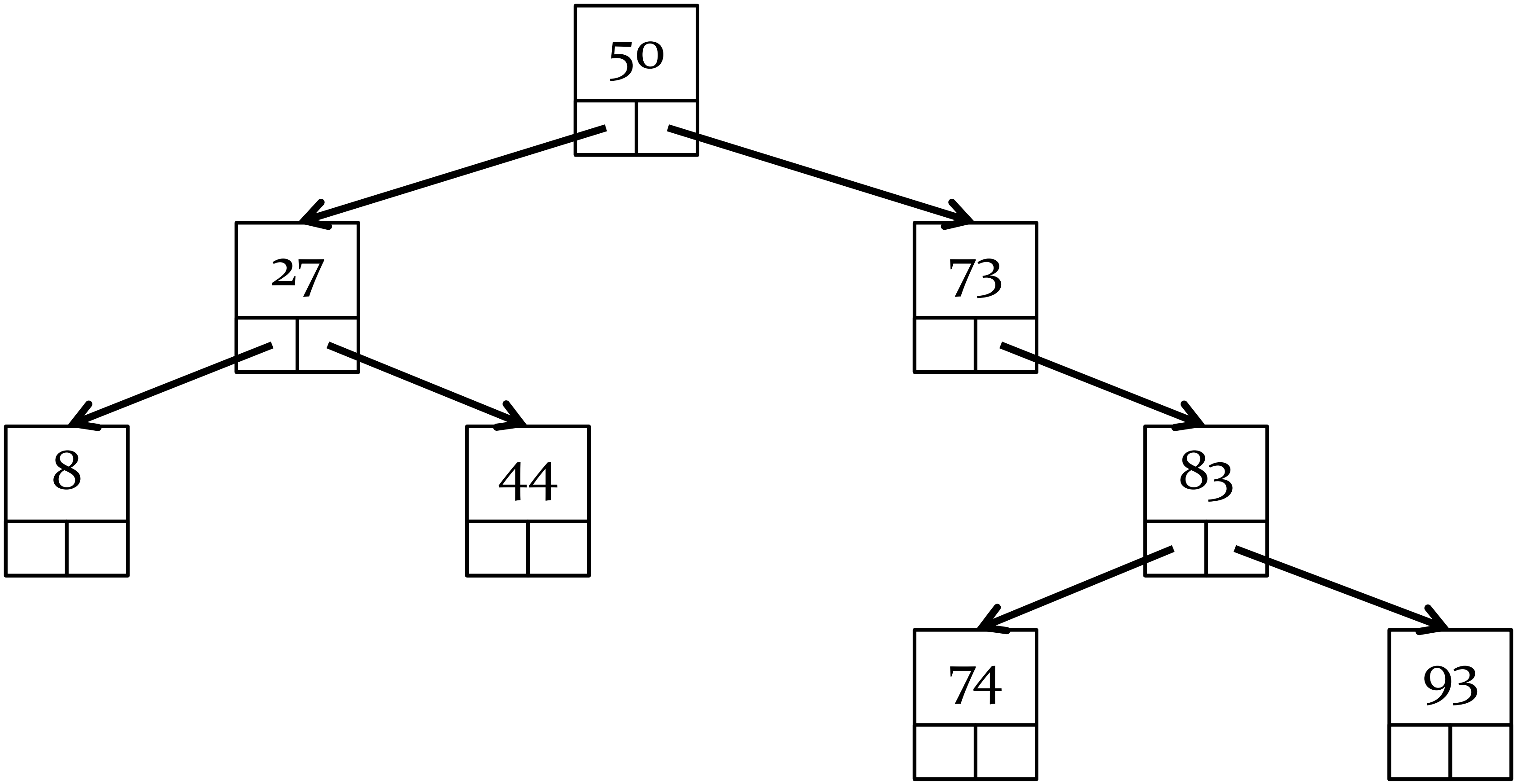
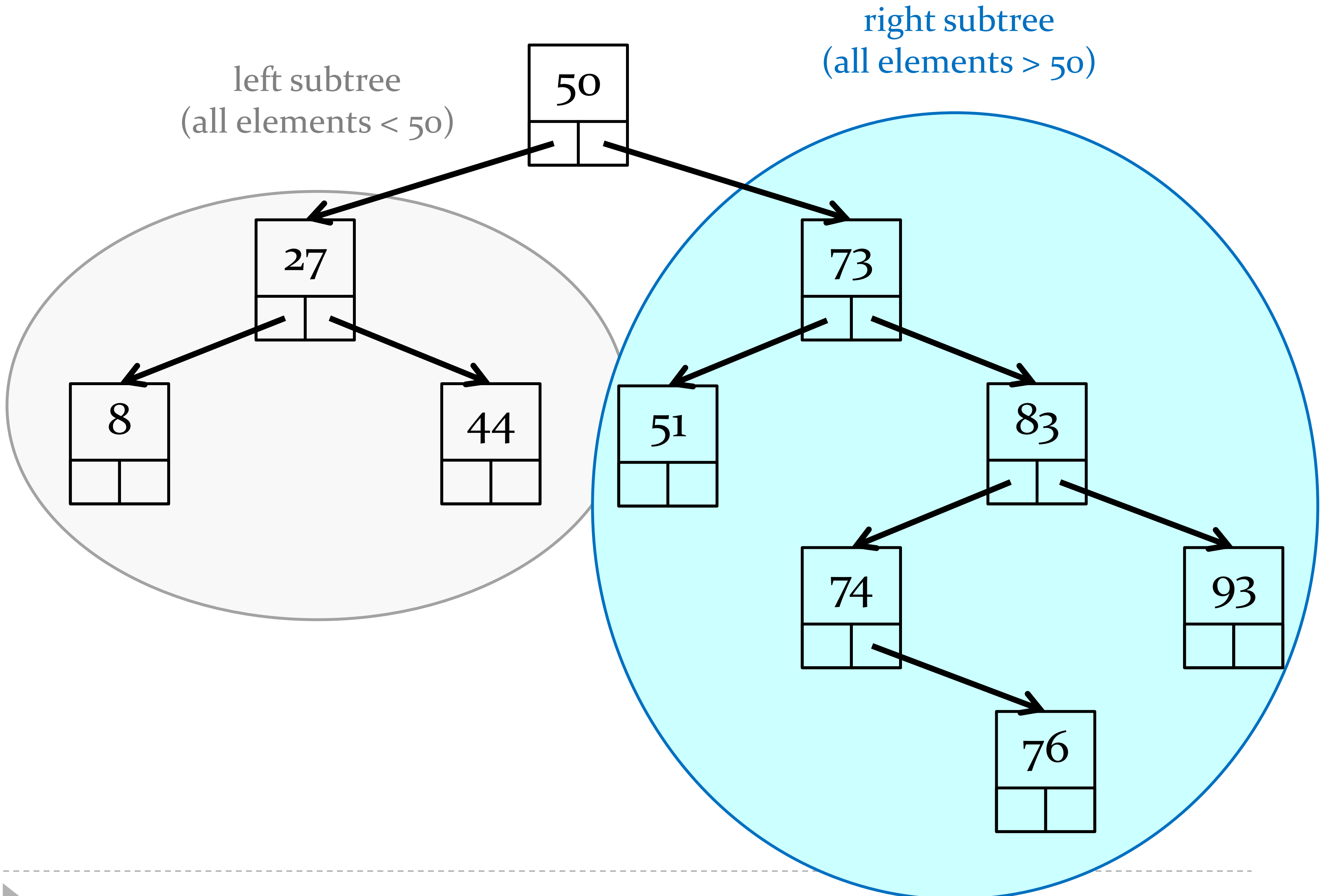
Binary Search Trees


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


right subtree
(all elements > 50)
left 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

```java
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

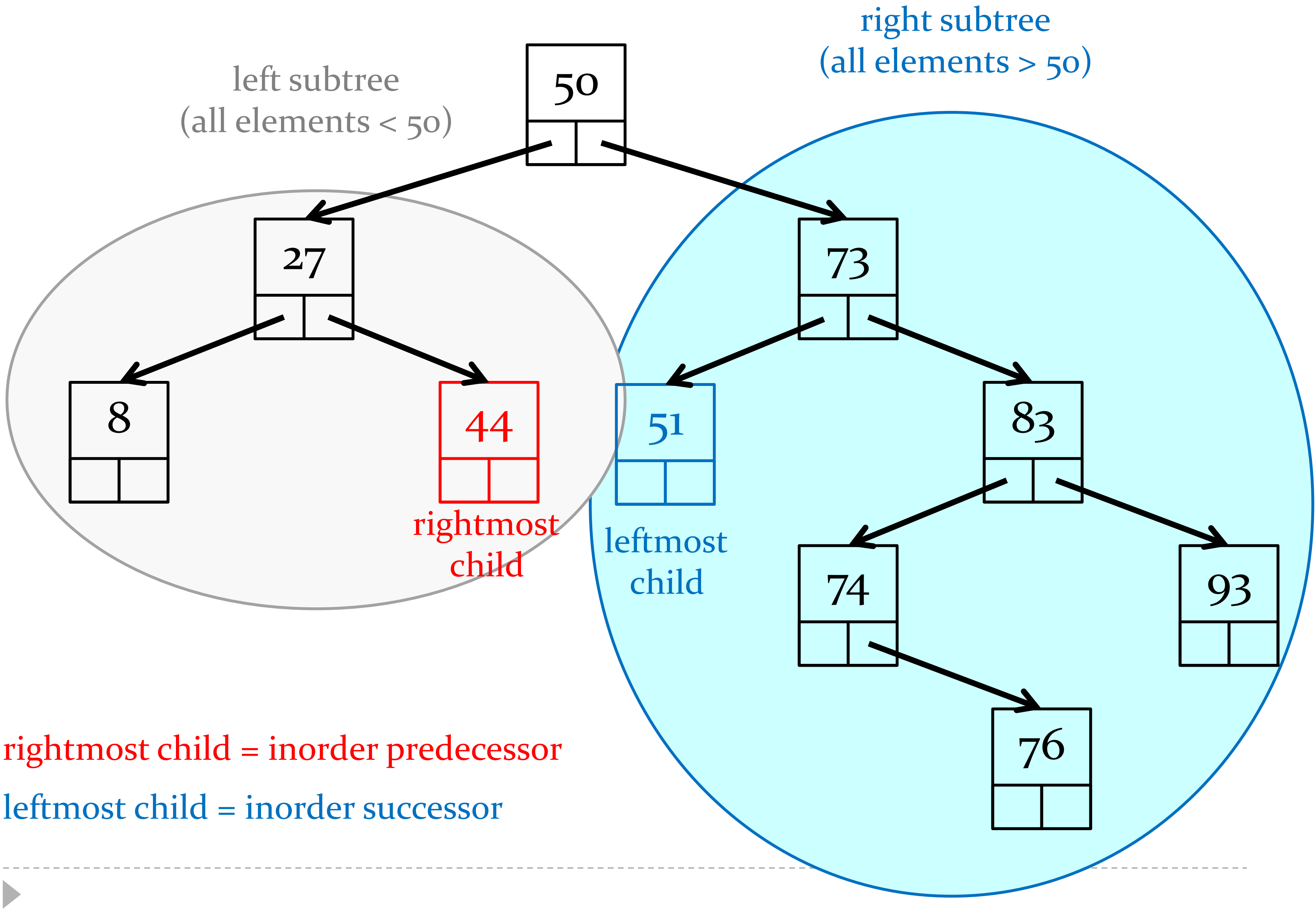
- the definition for a BST tells you everything that you need to know to add an element
- in a binary search tree:
  1. all nodes in the left subtree have data elements that are less than the data element of the root node
  2. all nodes in the right subtree have data elements that are greater than the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

```java
/**
 * 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);
}
```

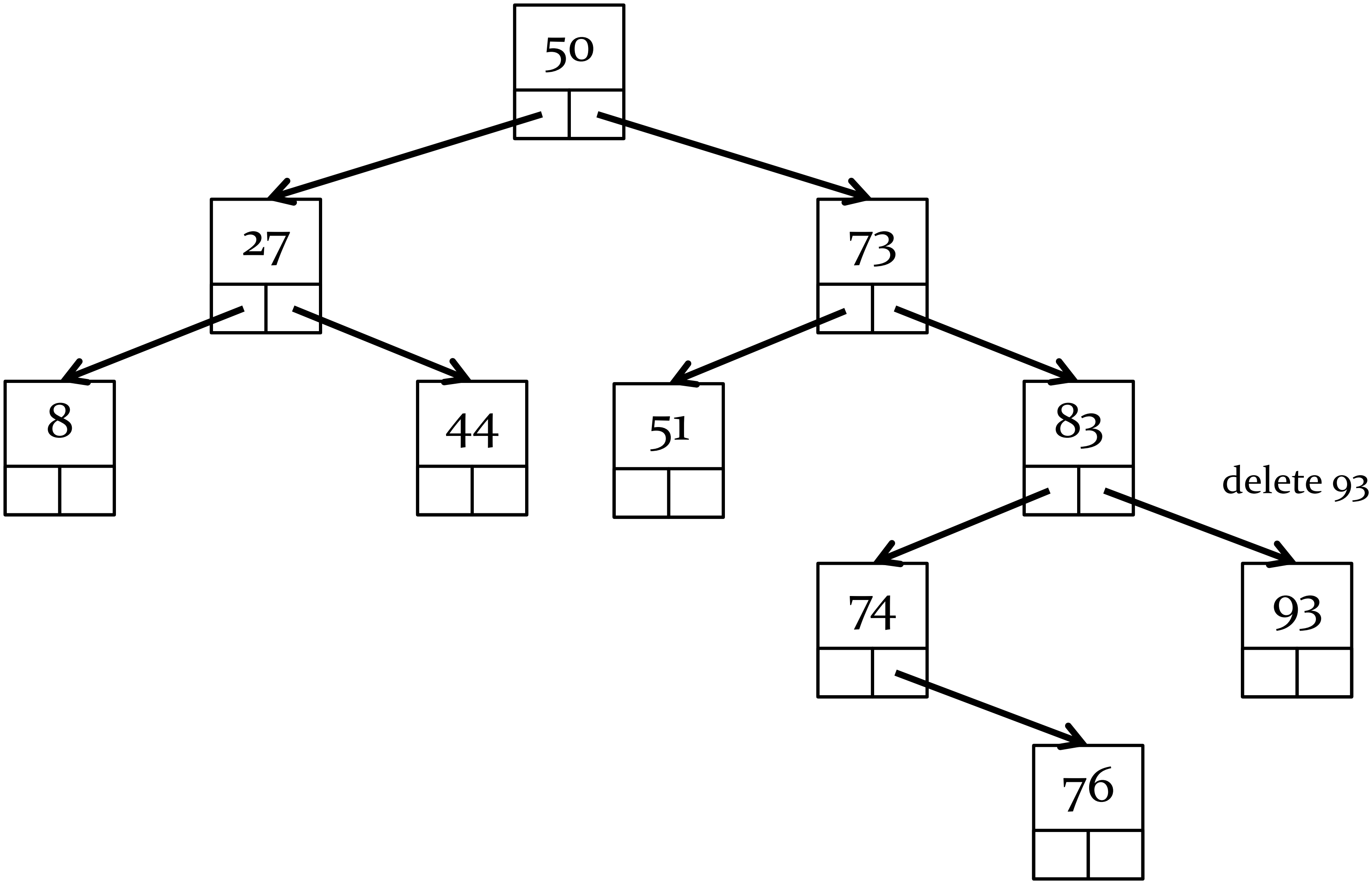
```java
/**
 * Find the node in a subtree that is the successor to the root of the
 * subtree. If the successor node exists, then it is the node that has the
 * smallest data element in the right subtree of <code>subtreeRoot</code>.
 *
 * @param subtreeRoot
 *          the root of the subtree
 * @return 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);
}
```

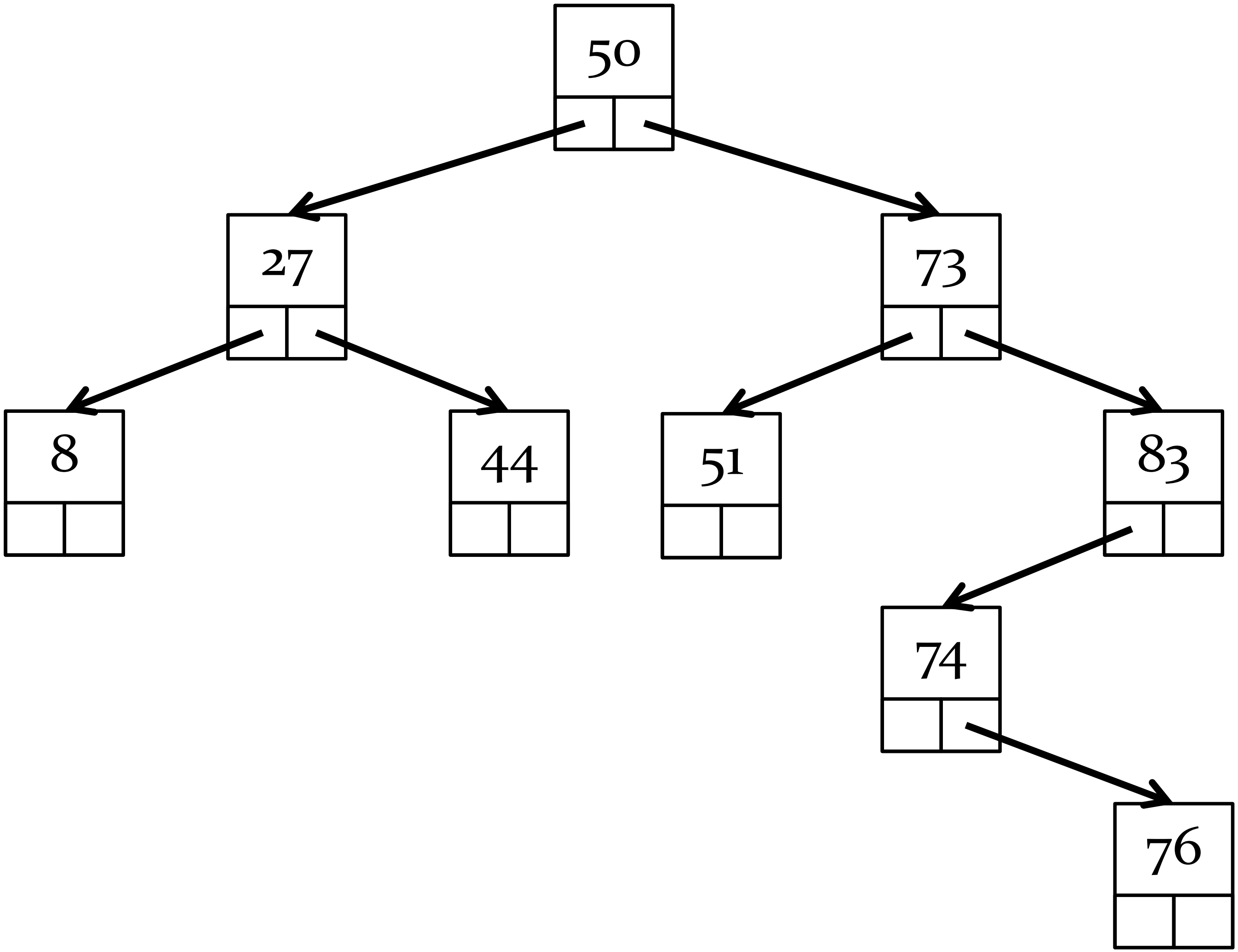
Deletion from a BST

- 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

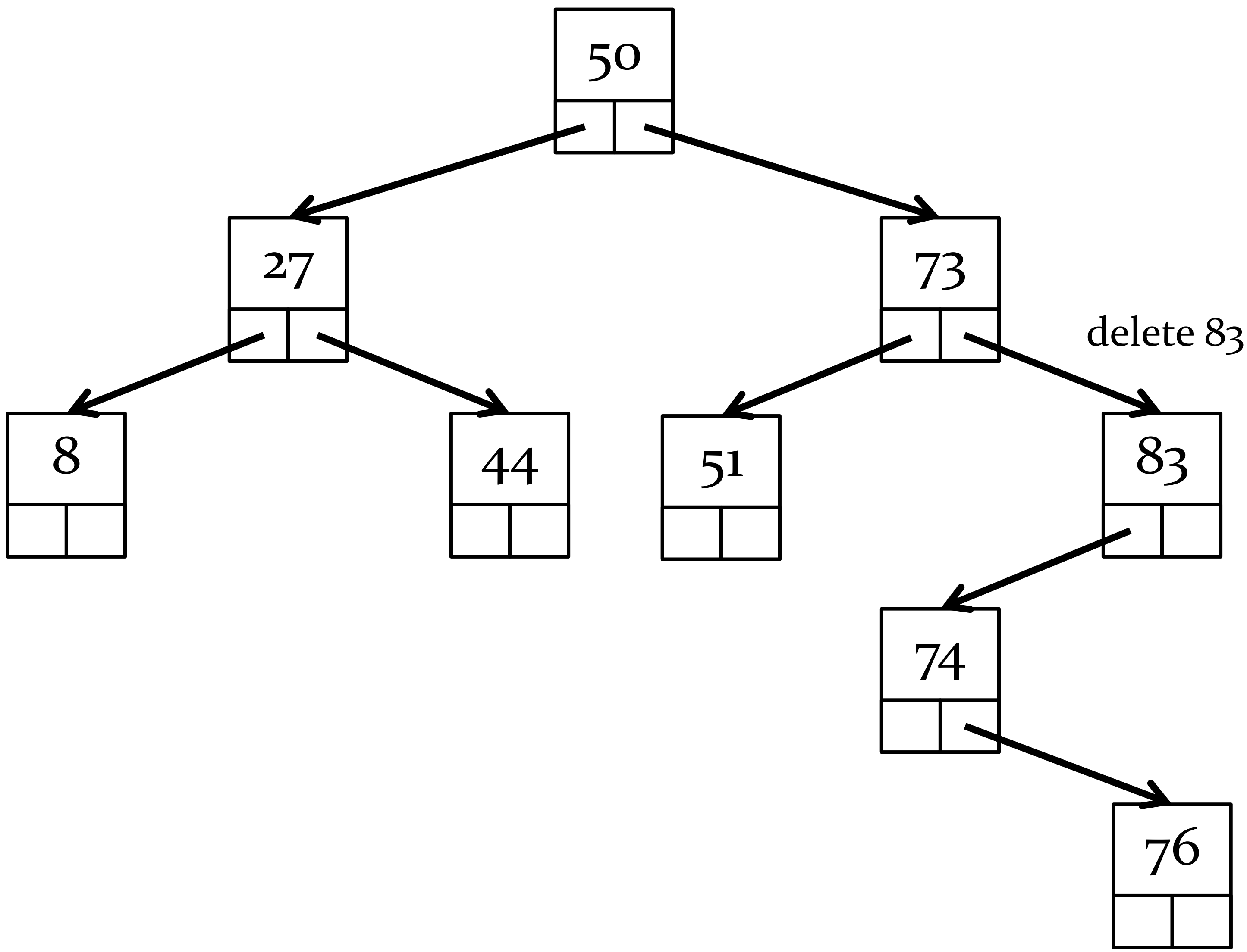

50
27
73
8
44
51
83
delete 93
74
93
76

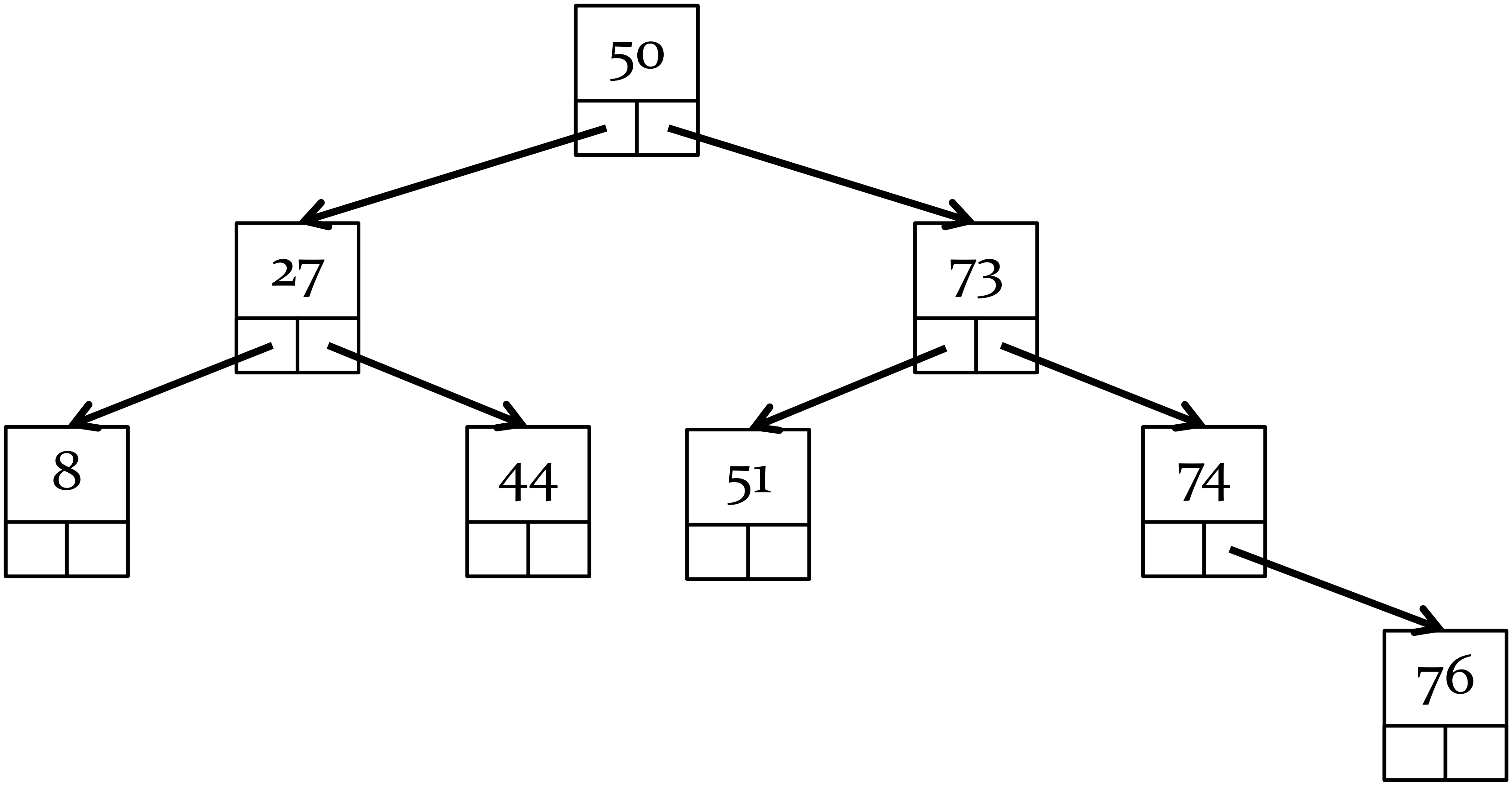

50
27
73
8
44
51
83
74
76

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

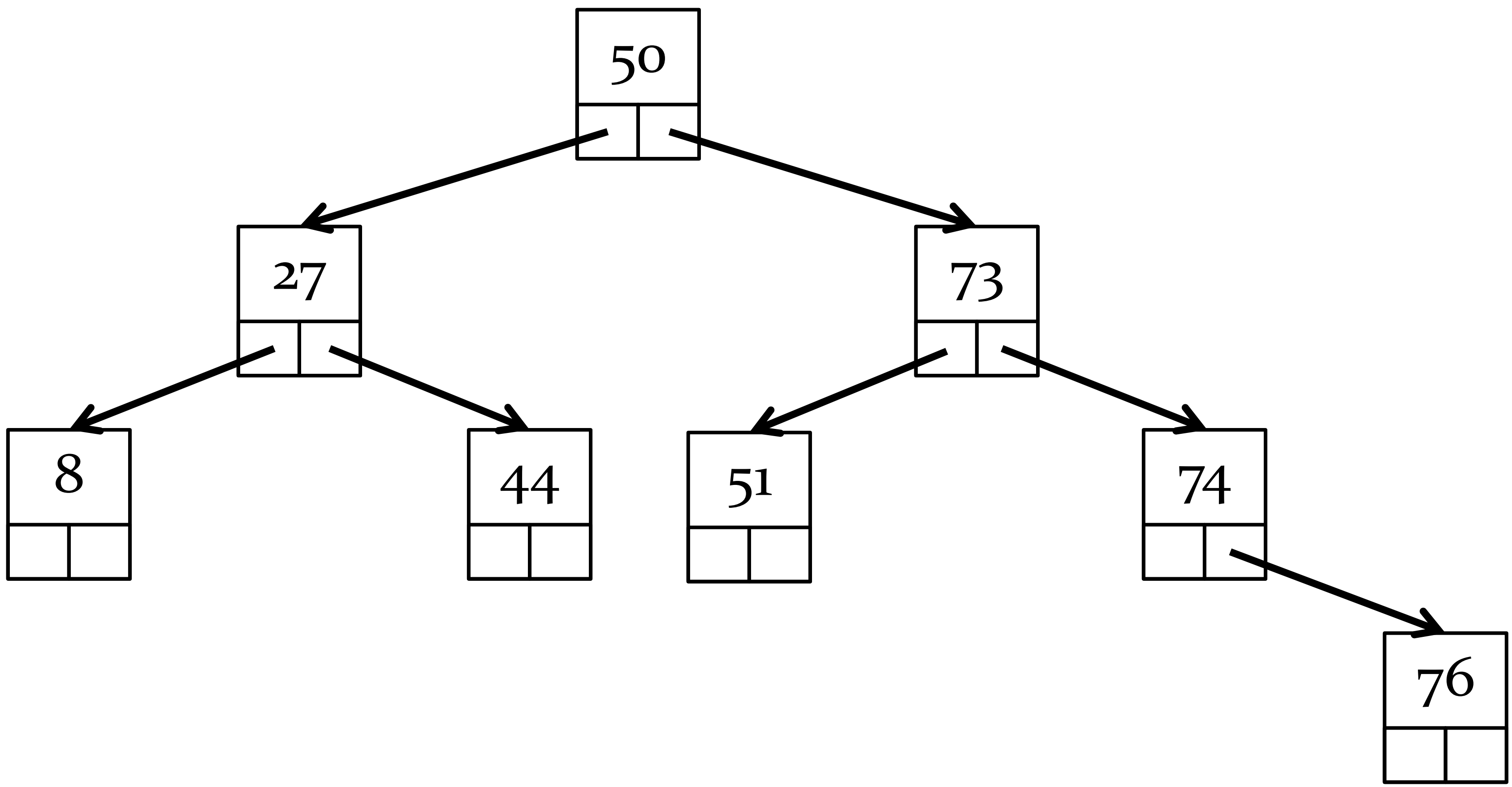
- deleting a node with two children is a little trickier
  - can you see how to do it?

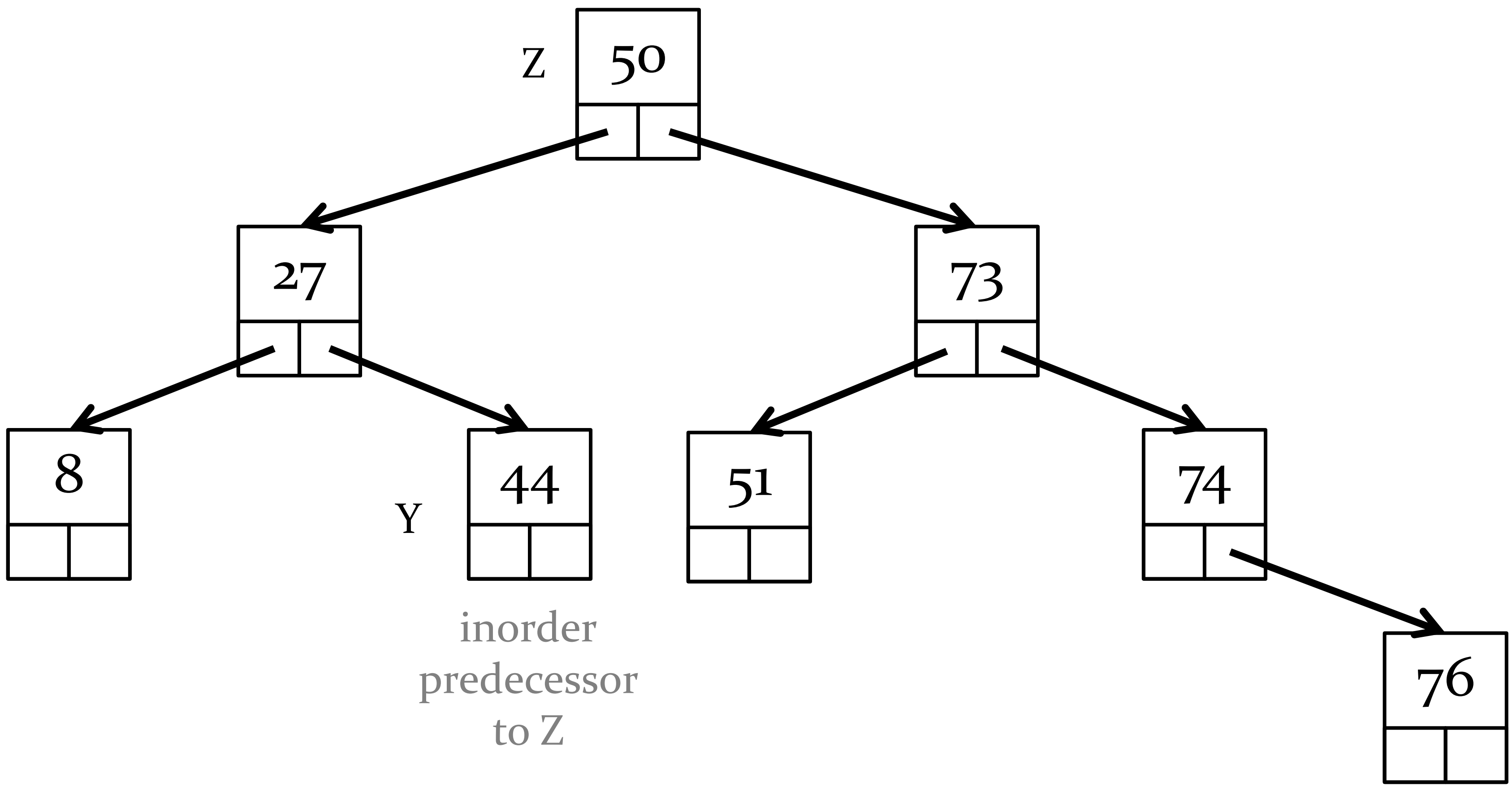
Deleting a Node with Two Children

- 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

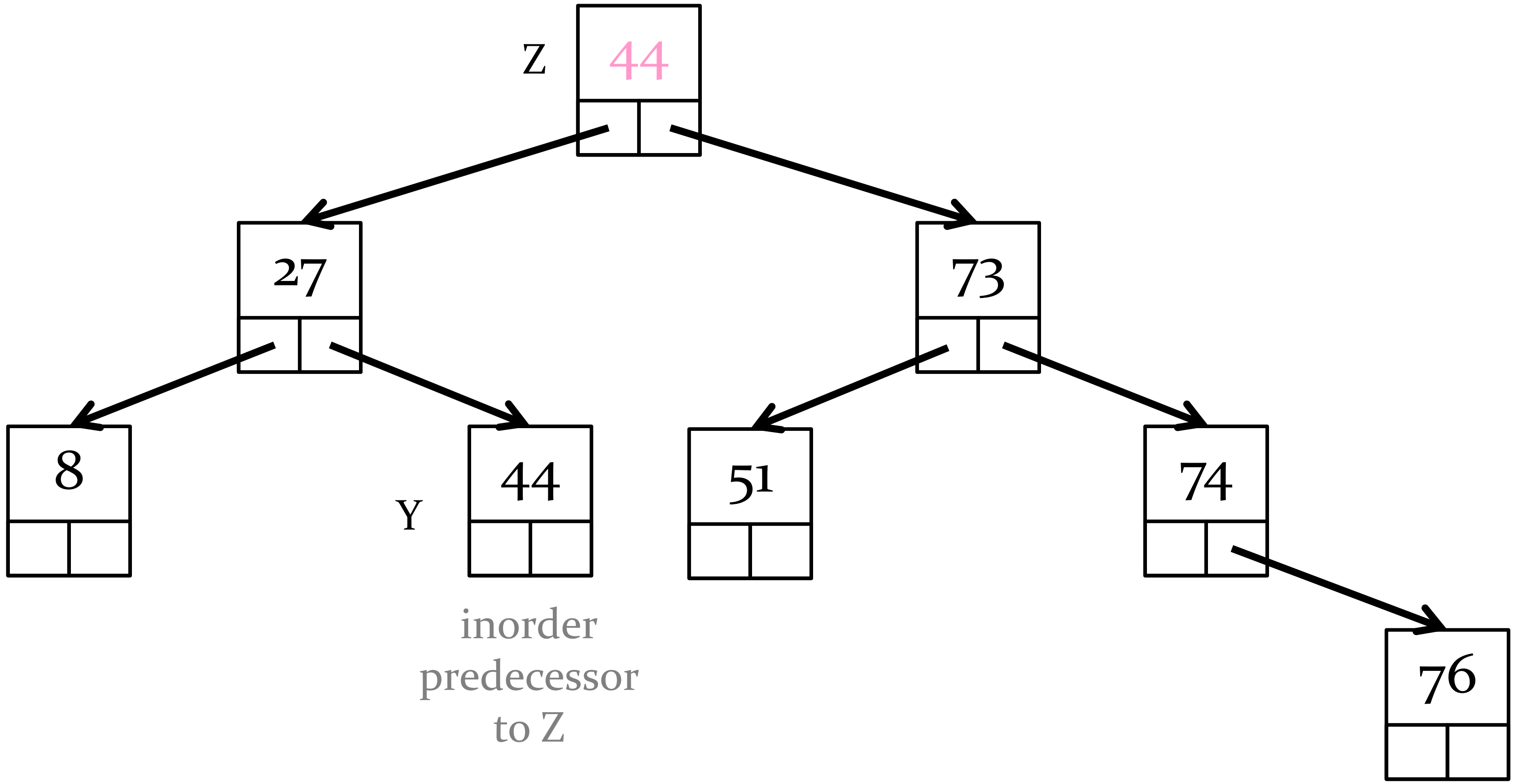
- e.g., delete 50

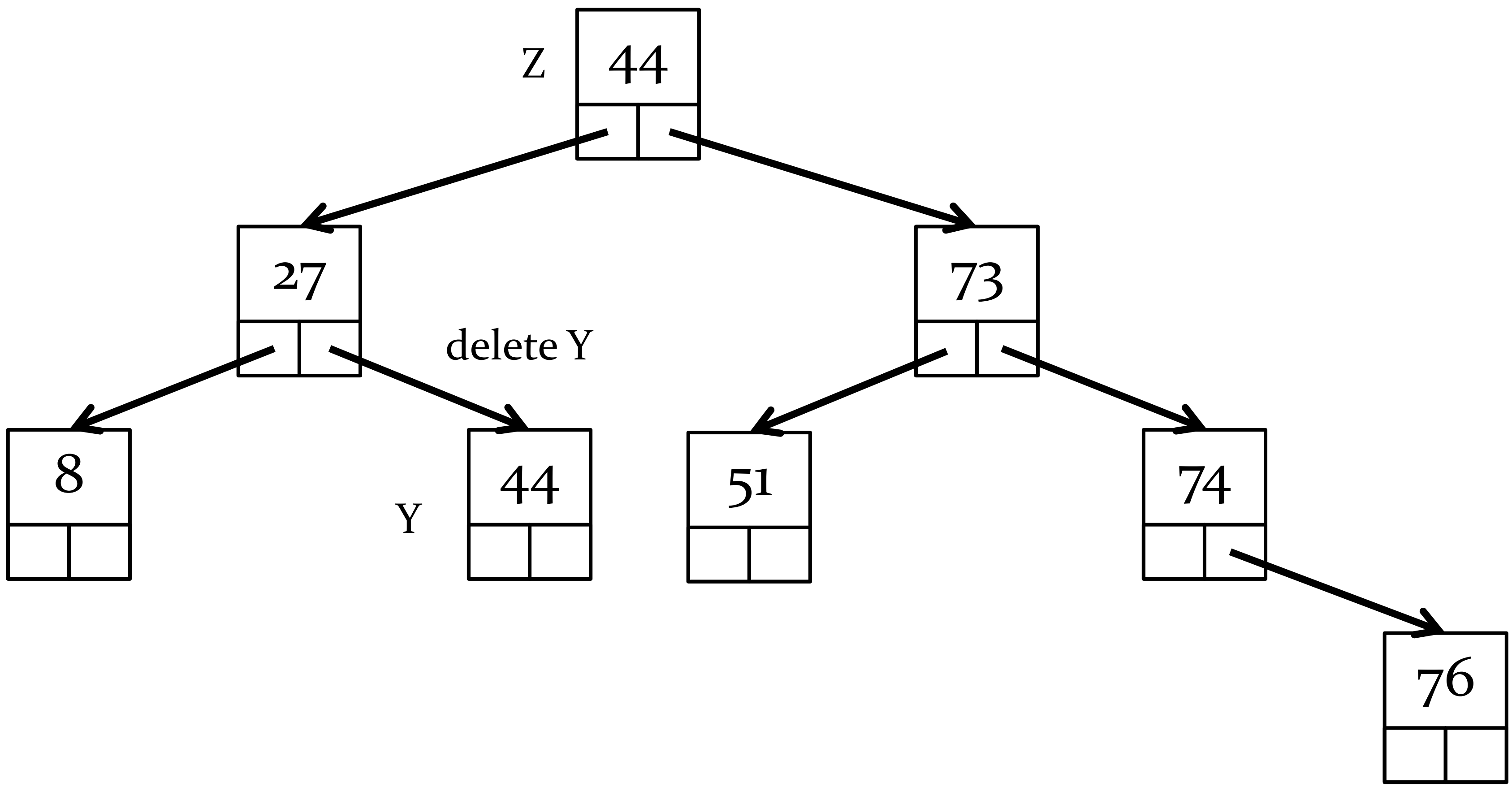
delete 50 using inorder predecessor



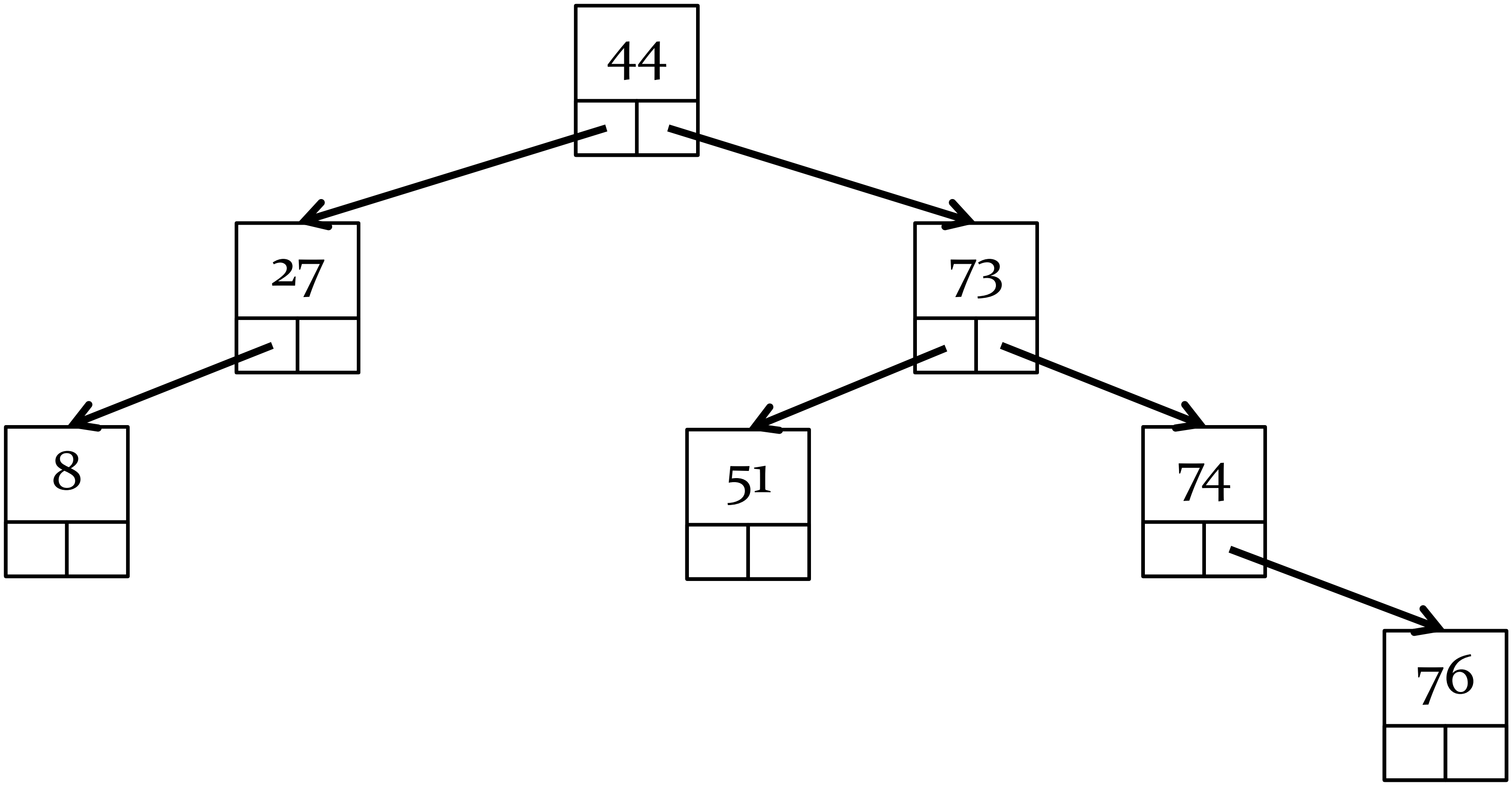

Z
50
27
73
8
Y
44
51
74
76
inorder
predecessor
to Z

copy Y data to Z data

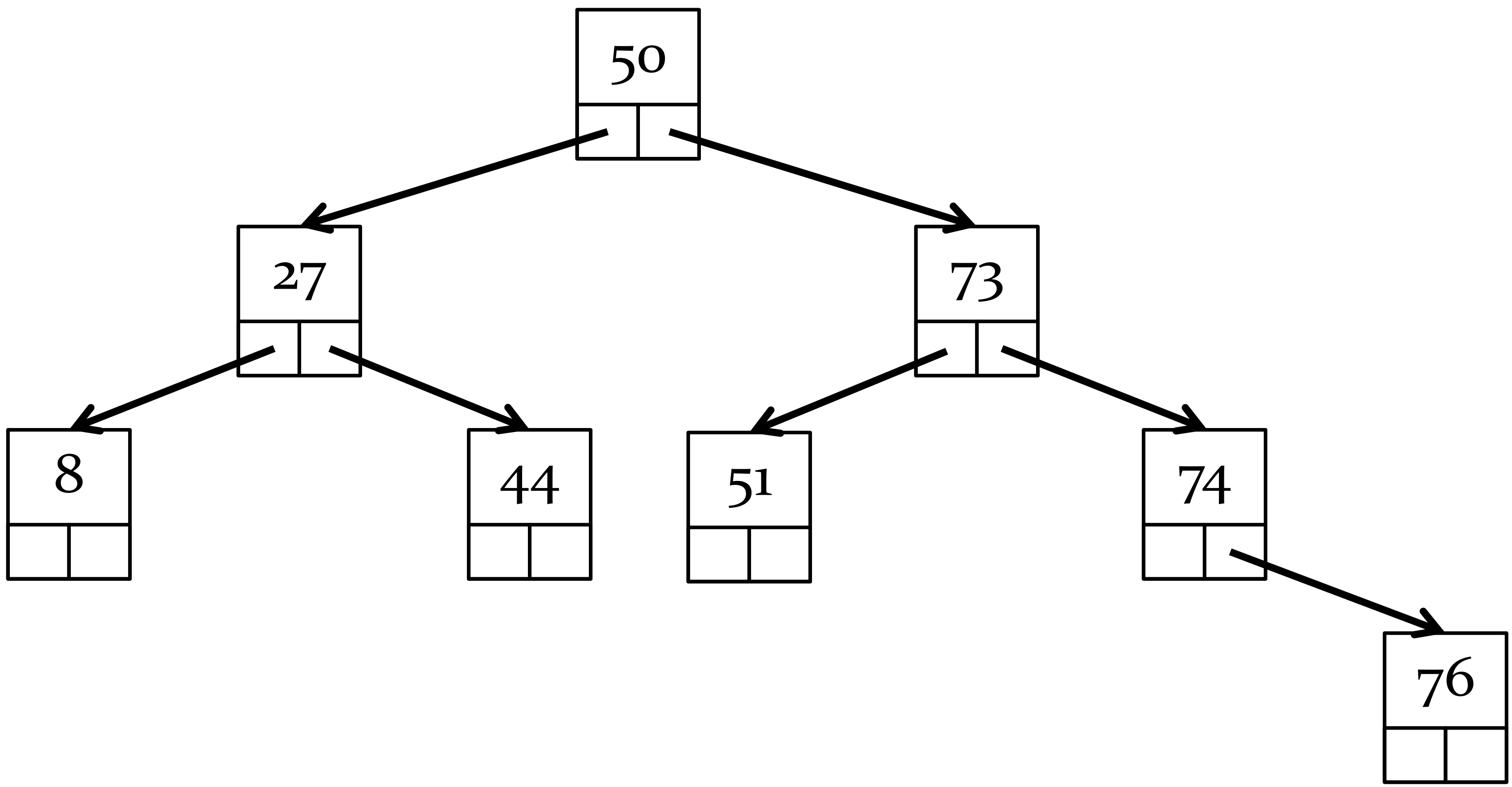


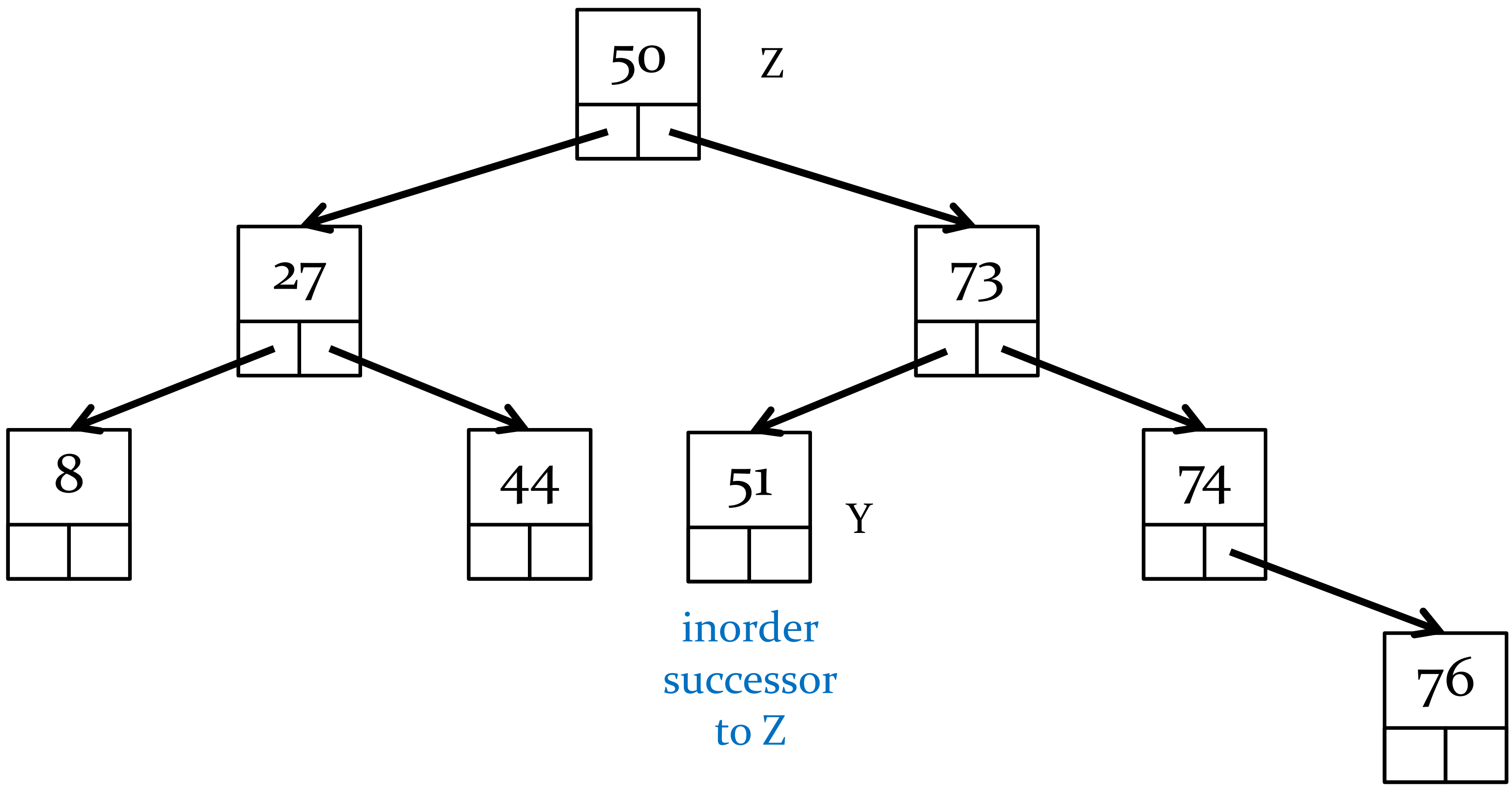


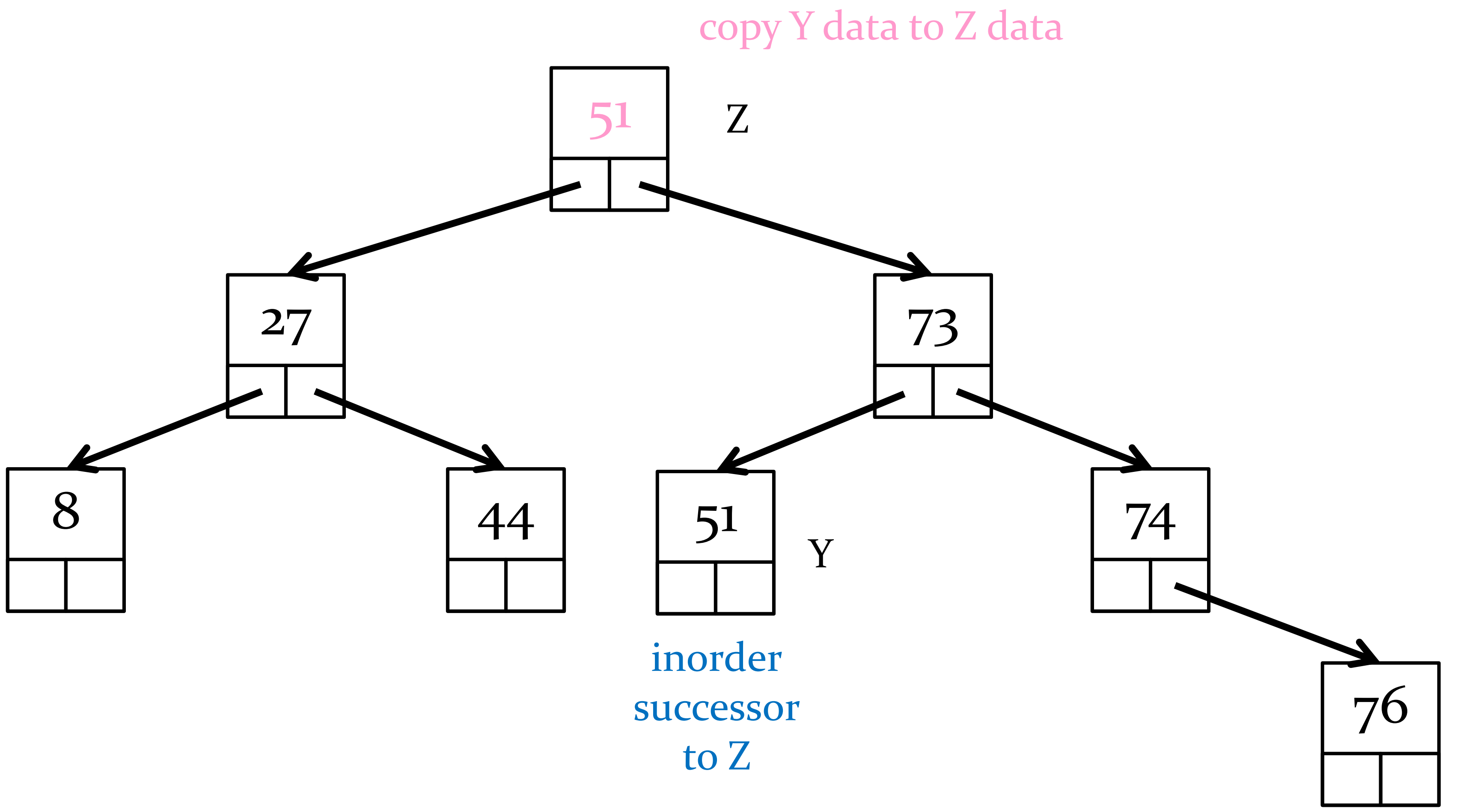
Z
44
27
73
delete Y
8
Y
44
51
74
76


44
27
73
8
51
74
76

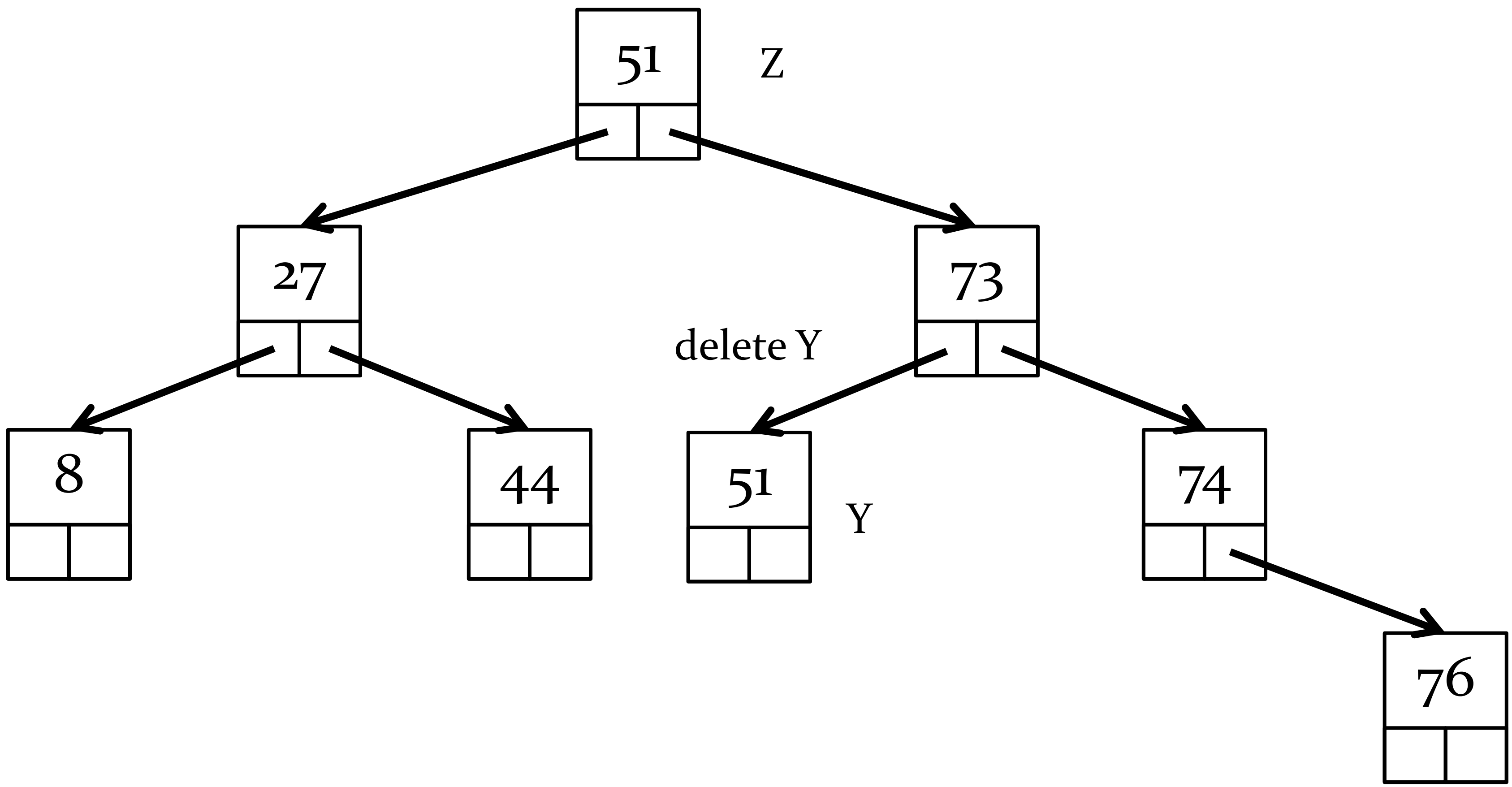
delete 50 using inorder successor

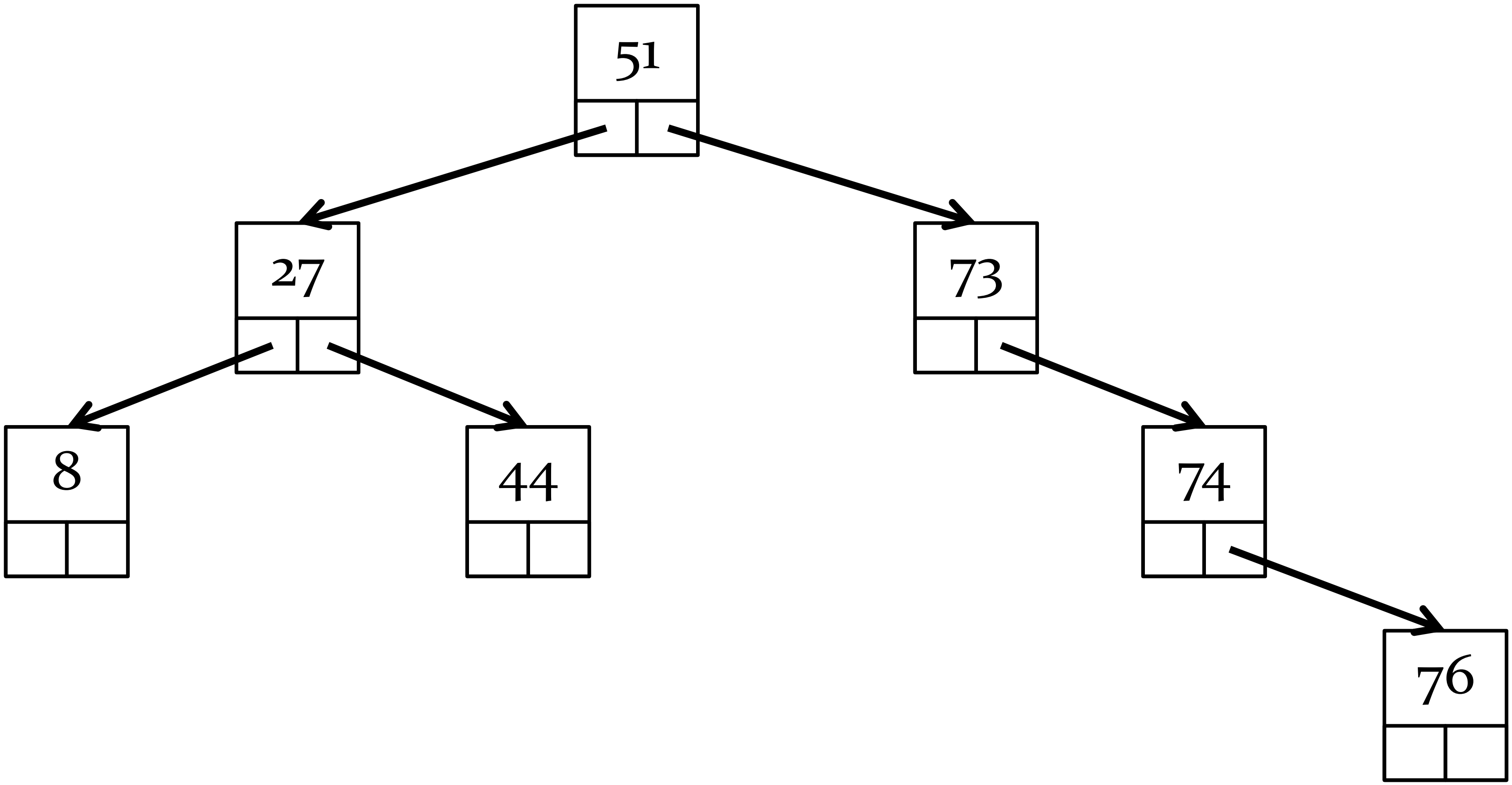



50
Z
27
73
8
44
51
Y
74
76
inorder
successor
to Z



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