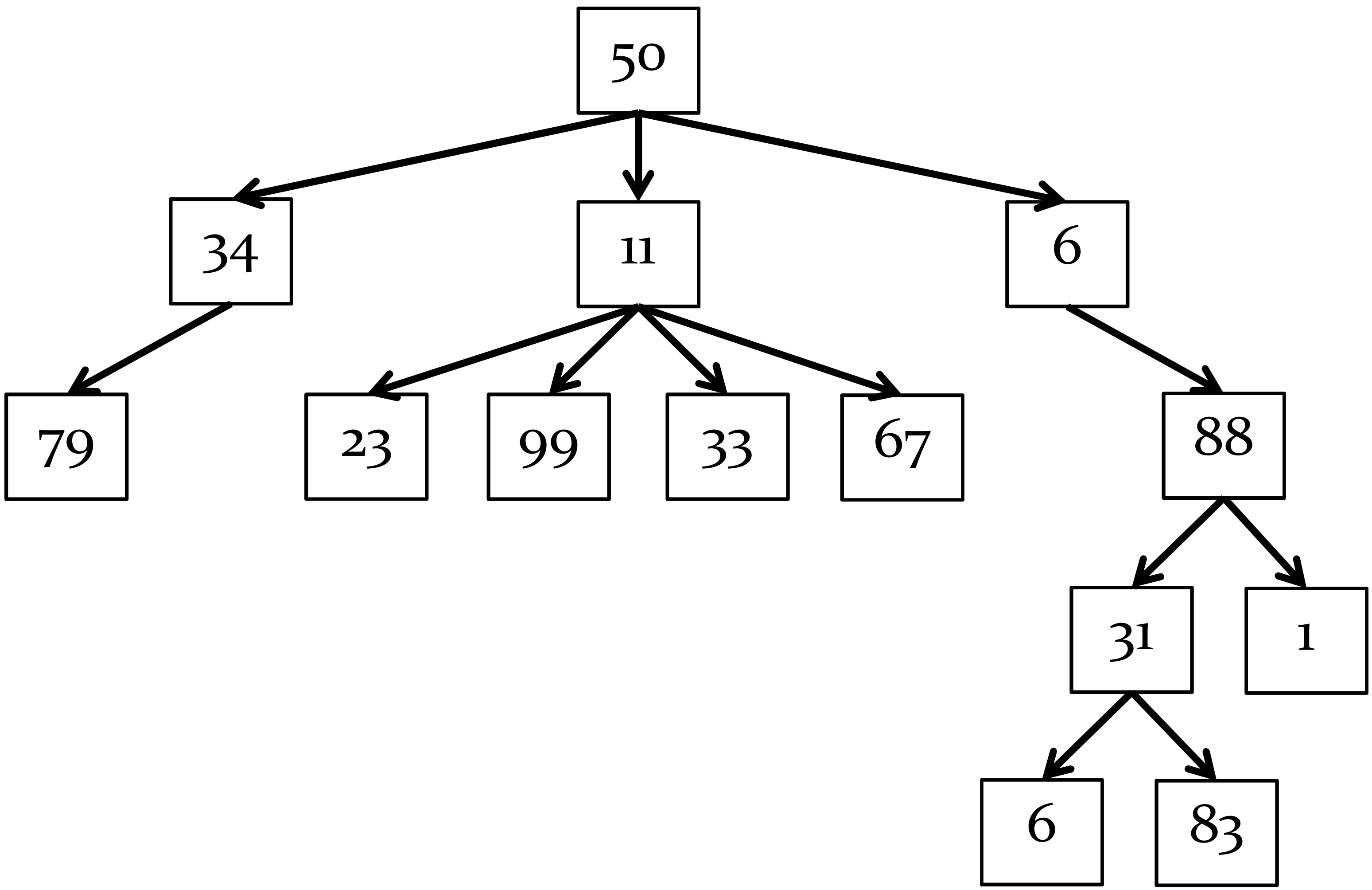
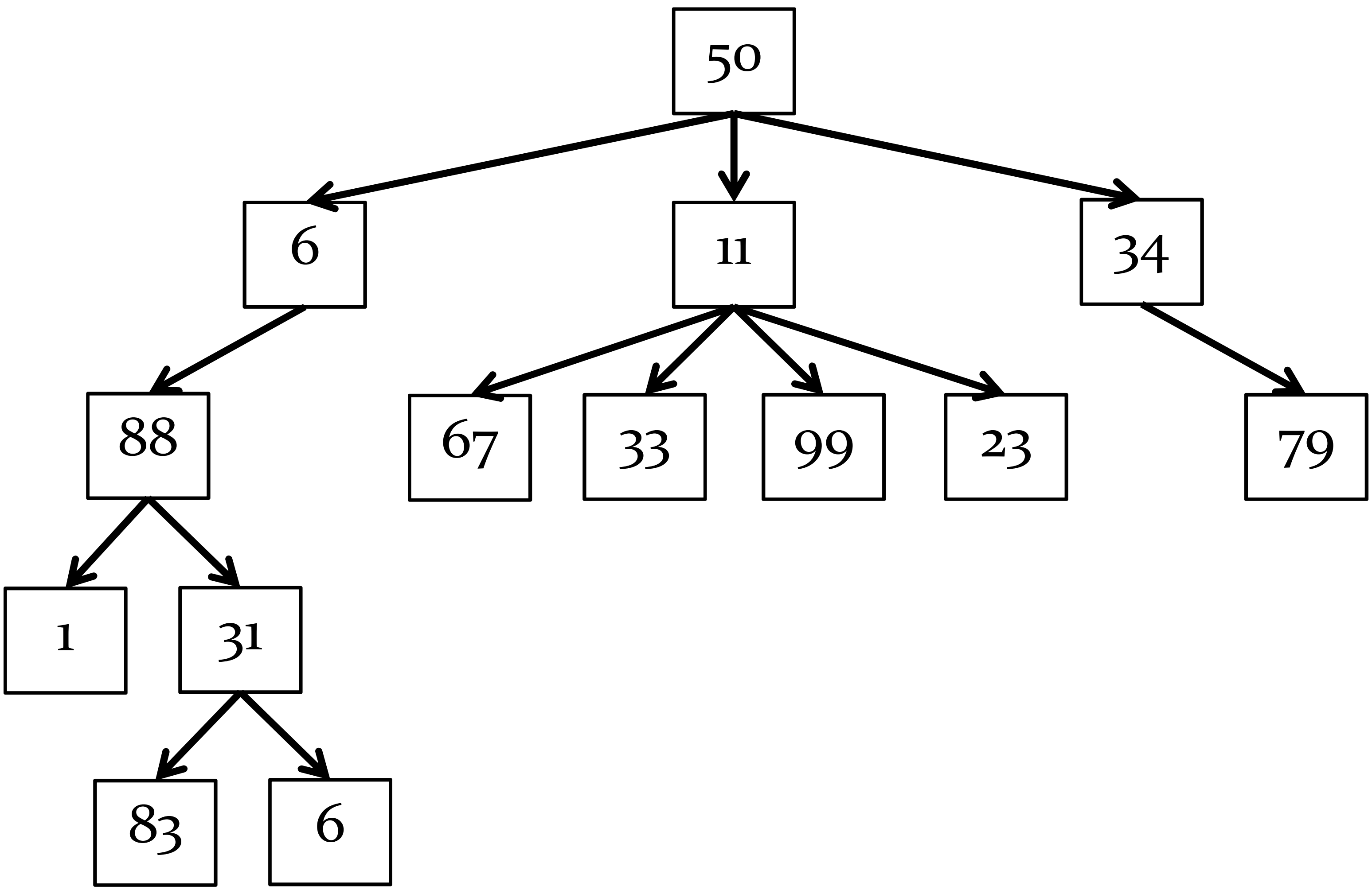
Recursive Objects (Part 4)

Trees

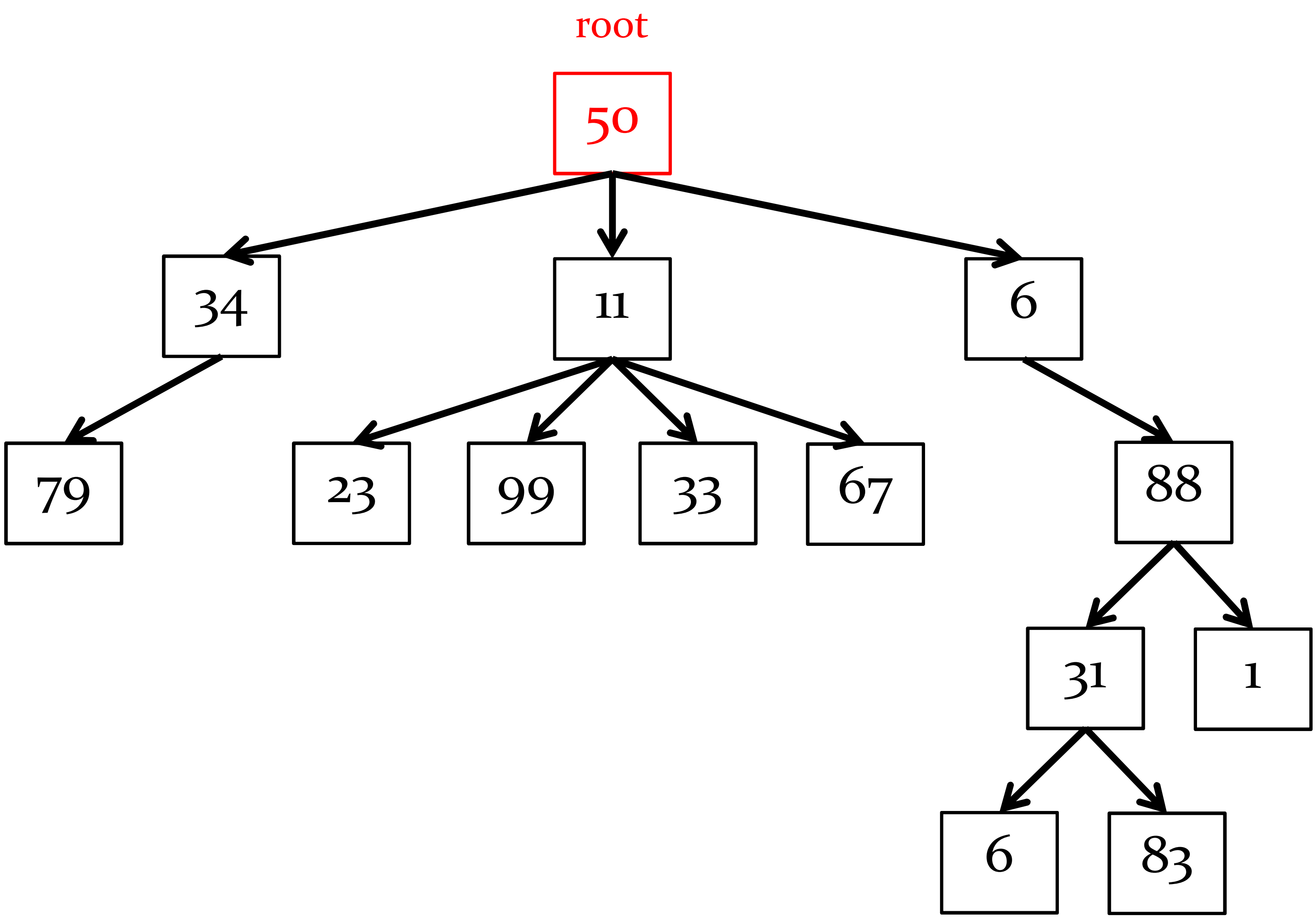
- a tree is a data structure made up of nodes
  - each node stores data
  - each node has links to zero or more nodes in the next level of the tree
    - children of the node
  - each node has exactly one parent node
    - except for the root node


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

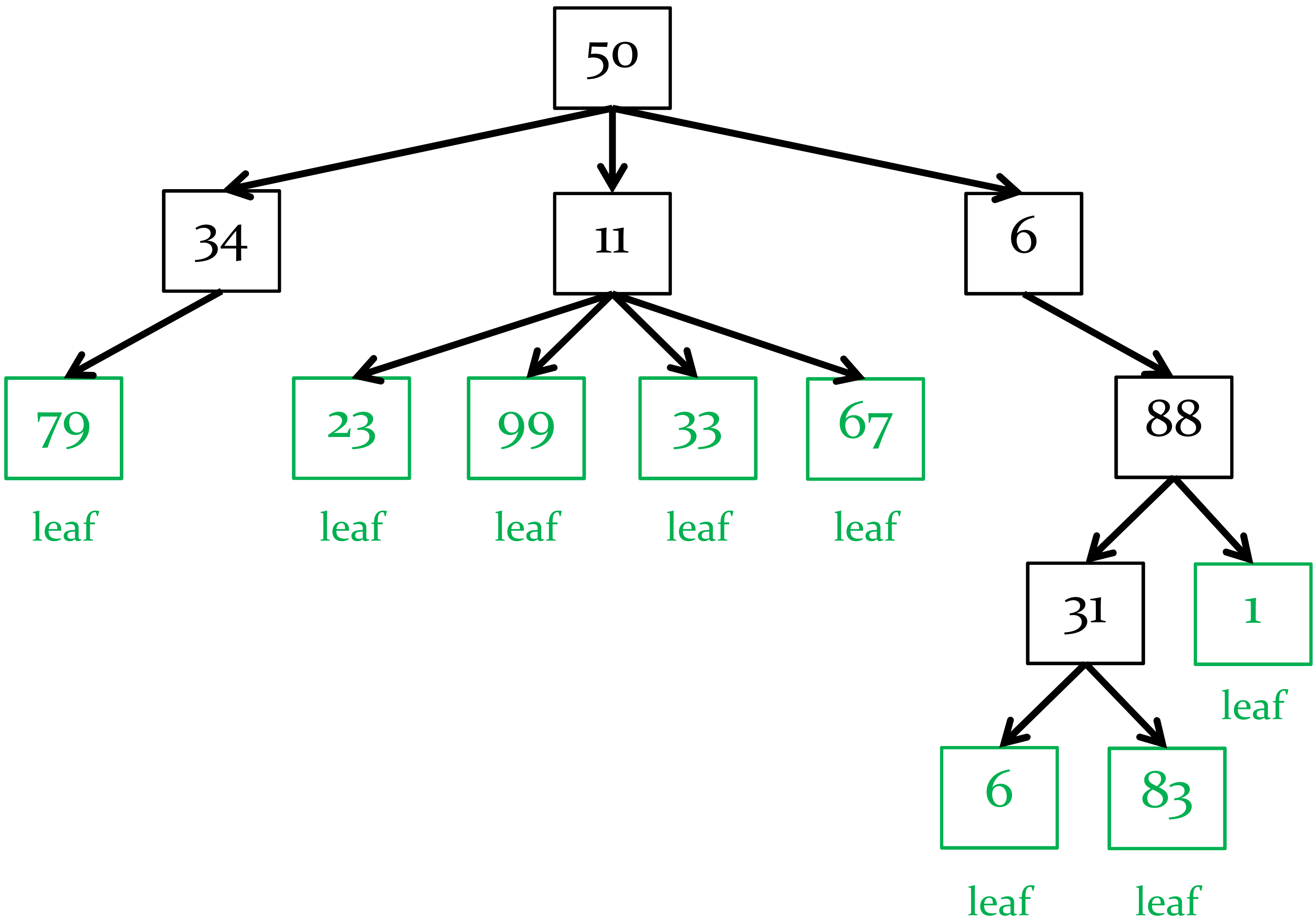
Trees

- 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

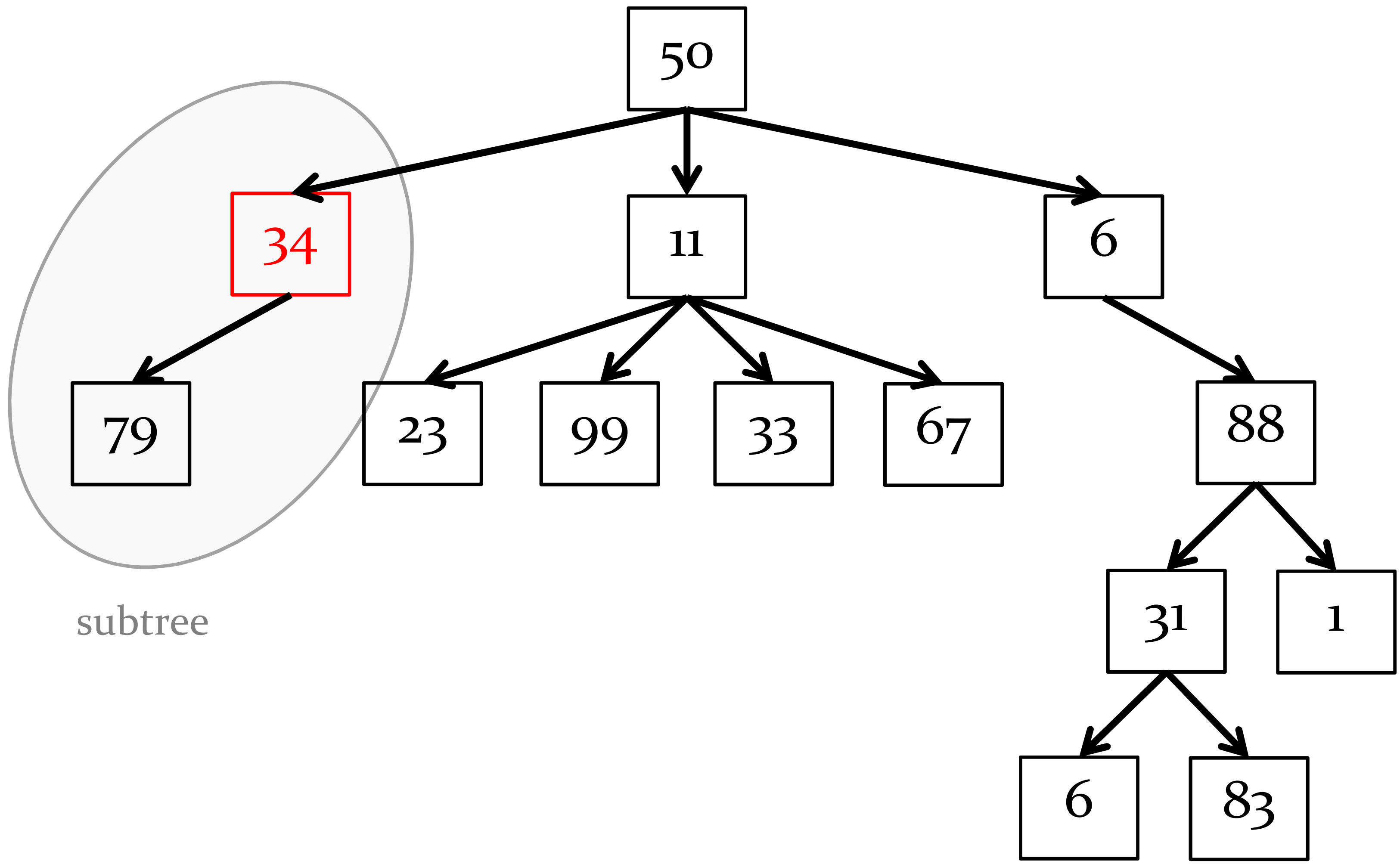
Trees

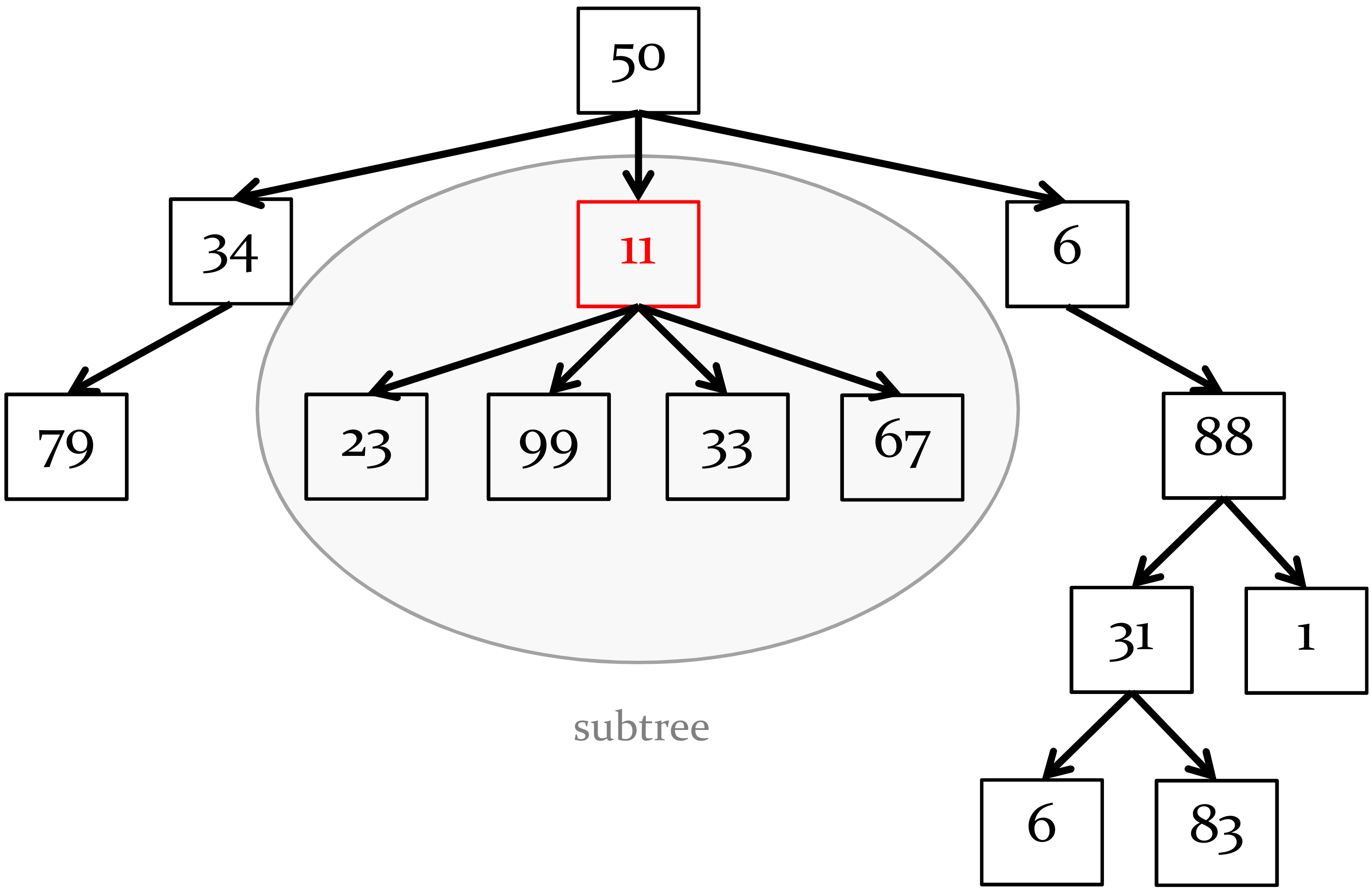
- a node without any children is called a leaf

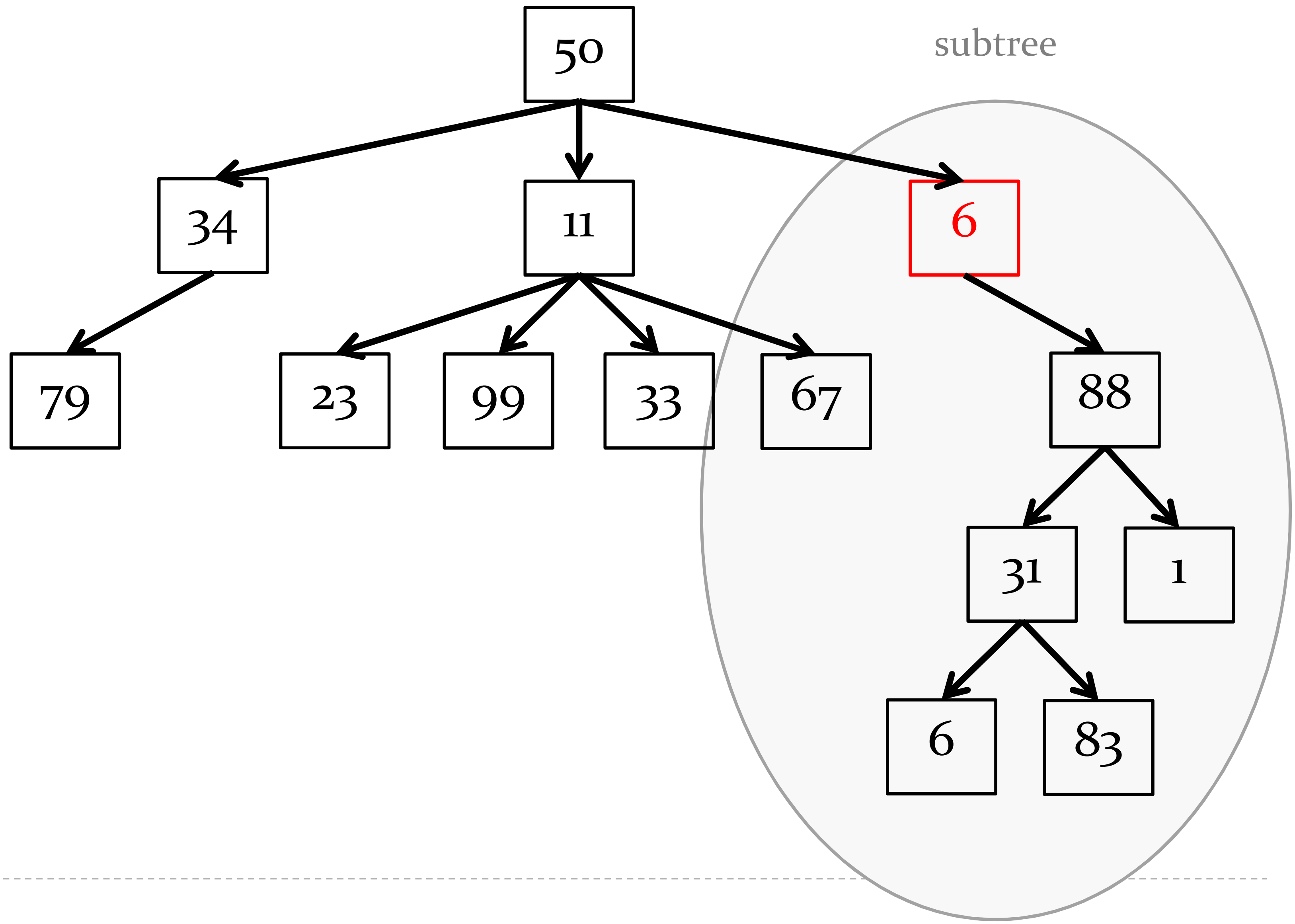

50
34
11
6
79
23
99
33
67
88
leaf
leaf
leaf
leaf
leaf
31
1
leaf
6
83
leaf
leaf

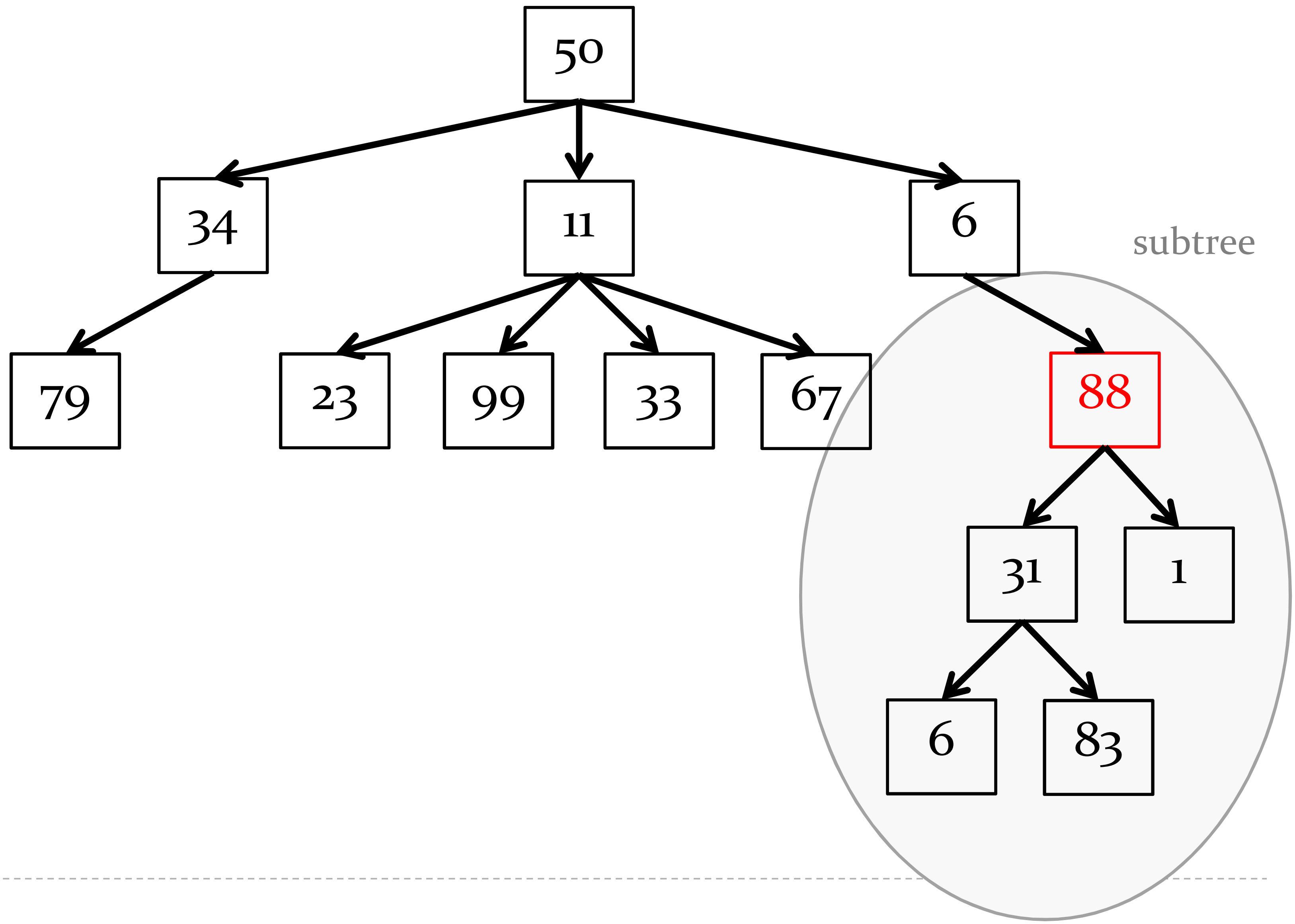
Trees

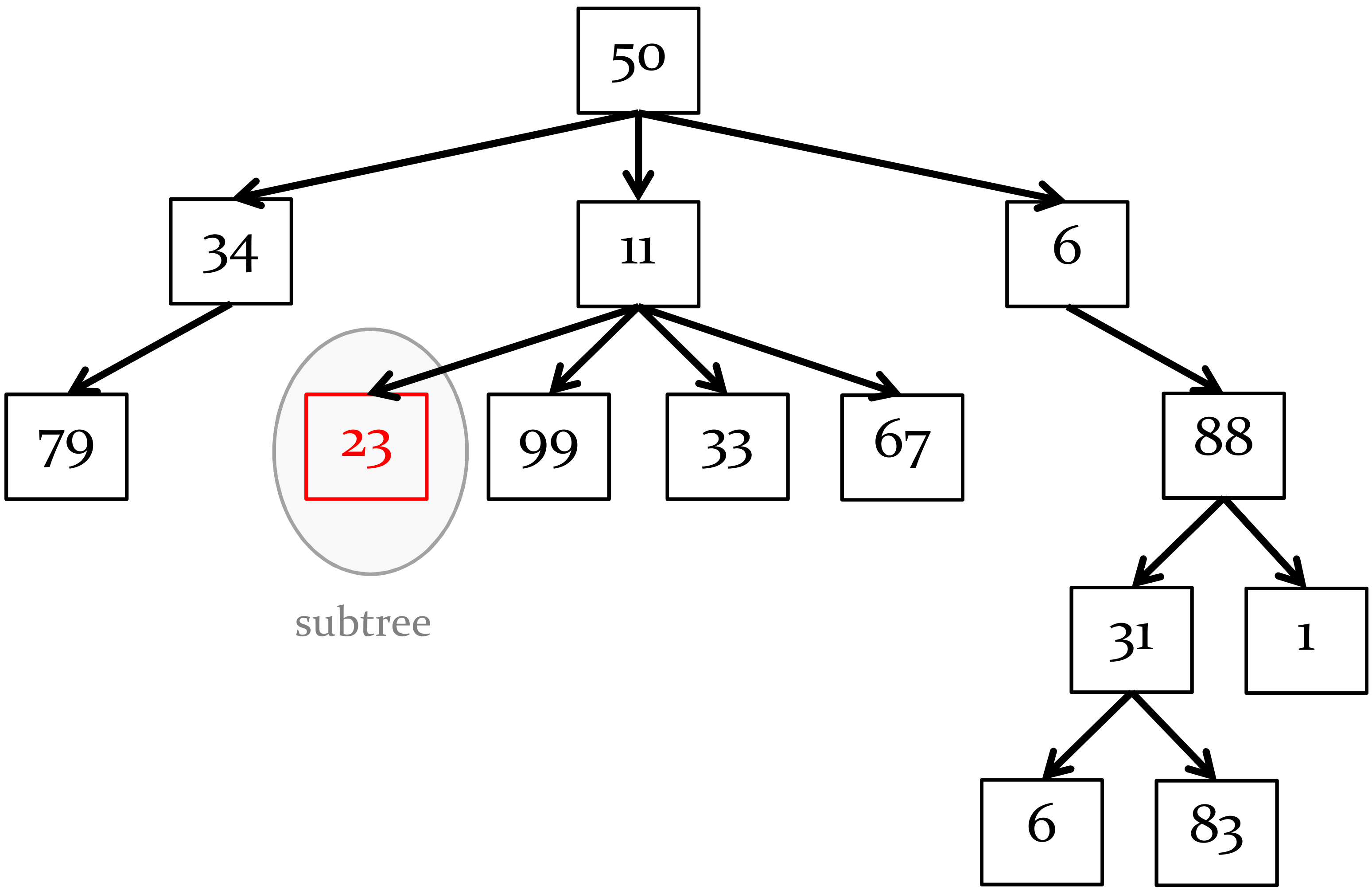
- the recursive structure of a tree means that every node is the root of a tree


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

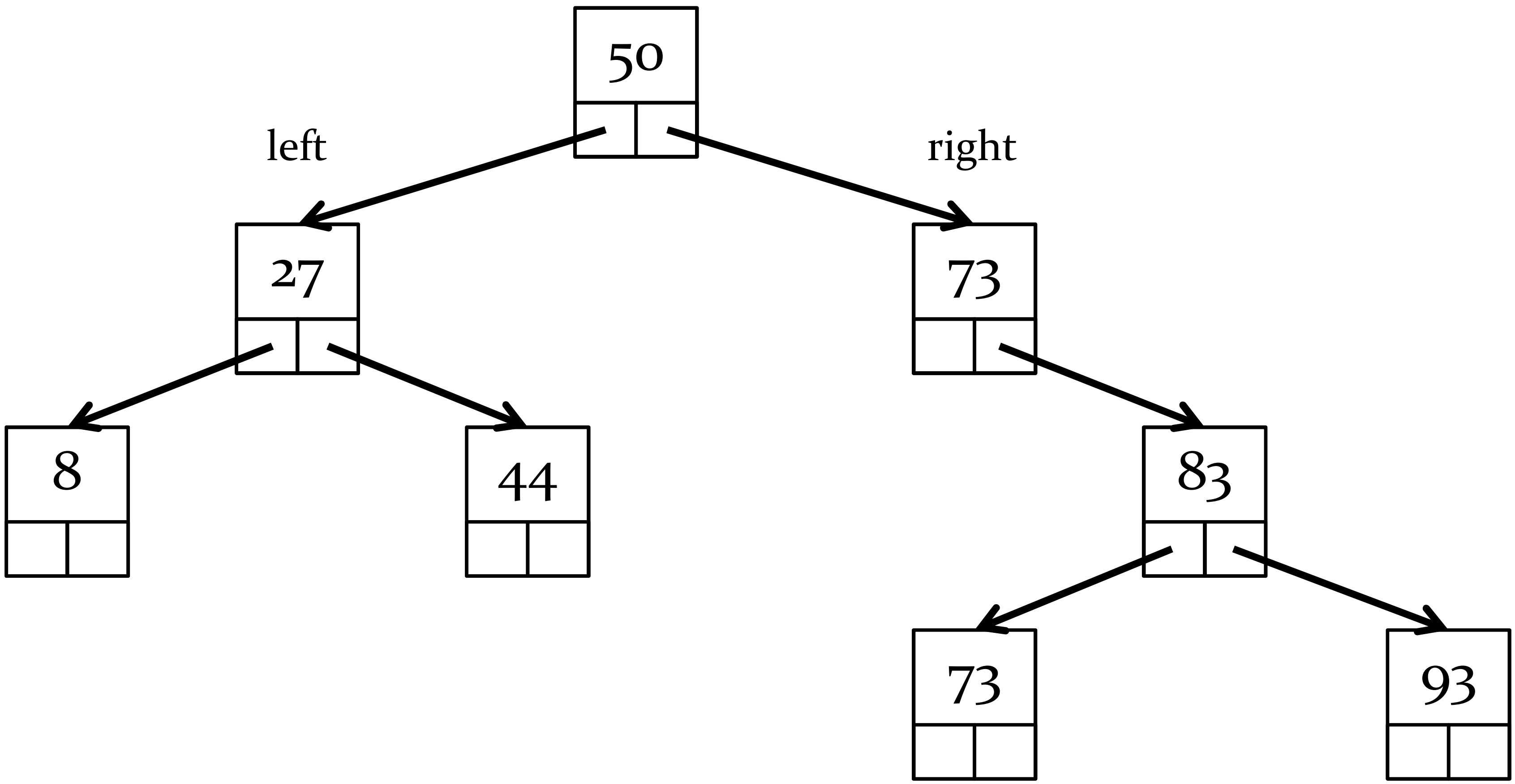

subtree
50
34
11
6
79
23
99
33
67
88
31
1
6
83

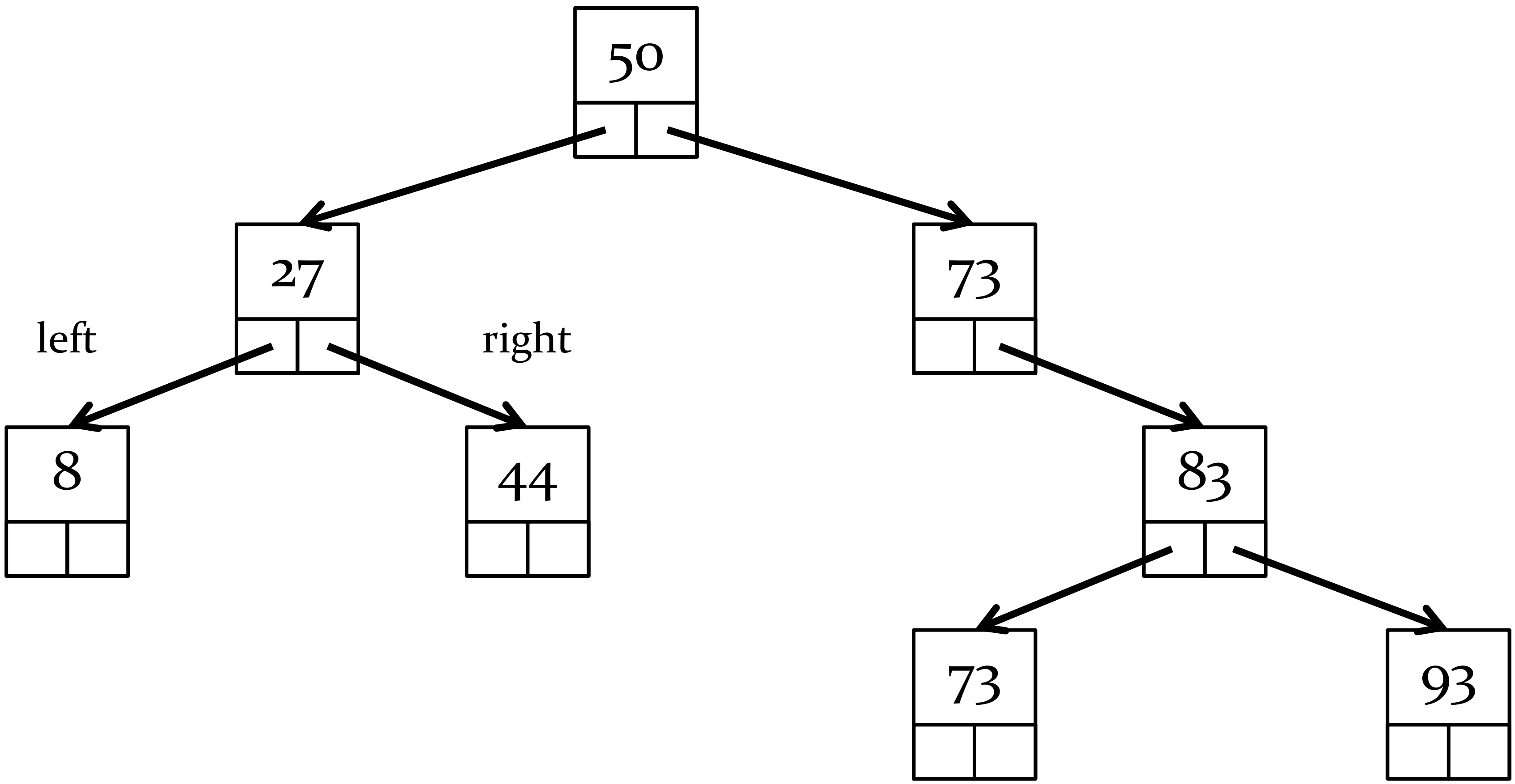

50
34
11
6
subtree
79
23
99
33
67
88
31
1
6
83

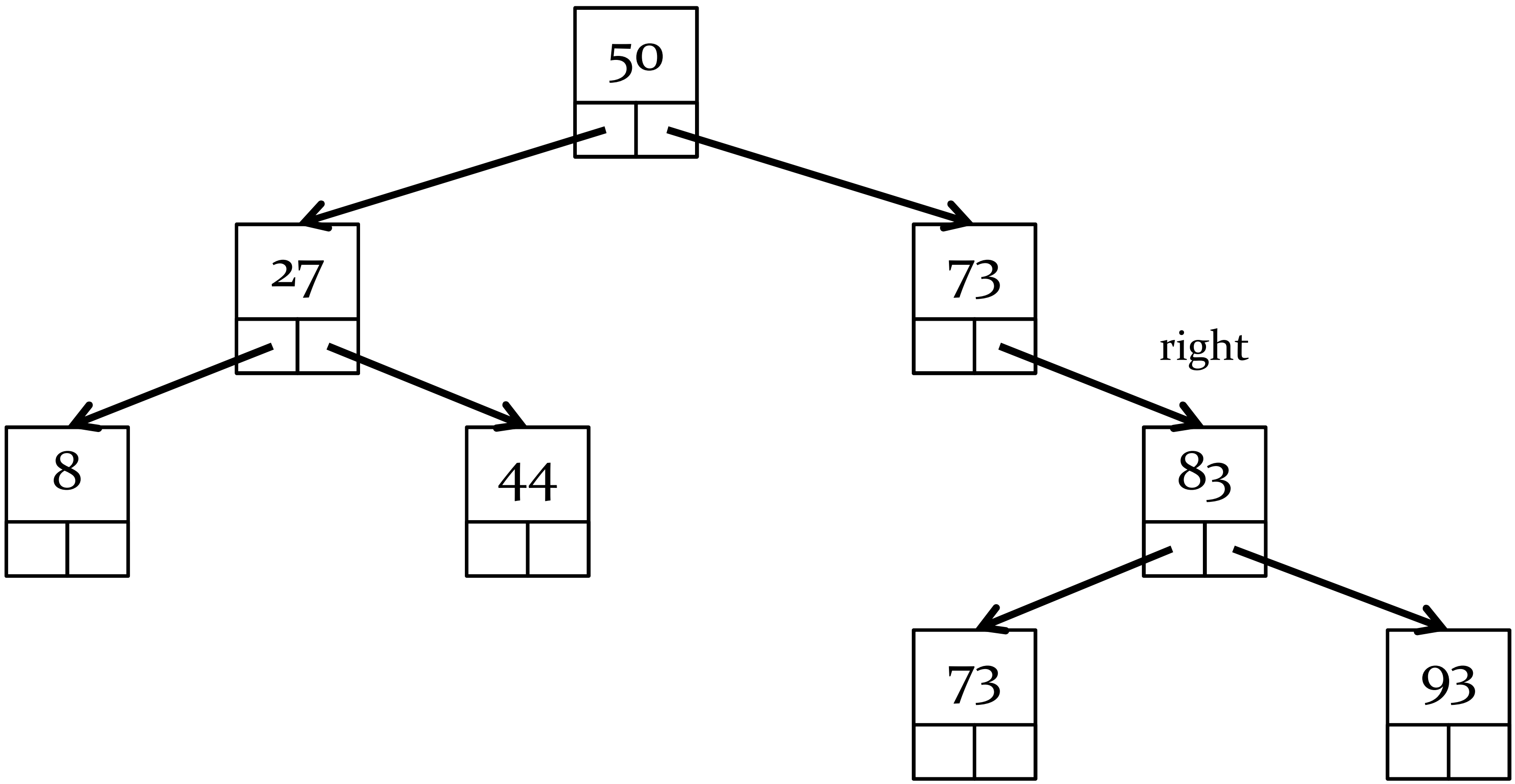

50
34
11
6
79
23
99
33
67
88
31
1
6
83
subtree

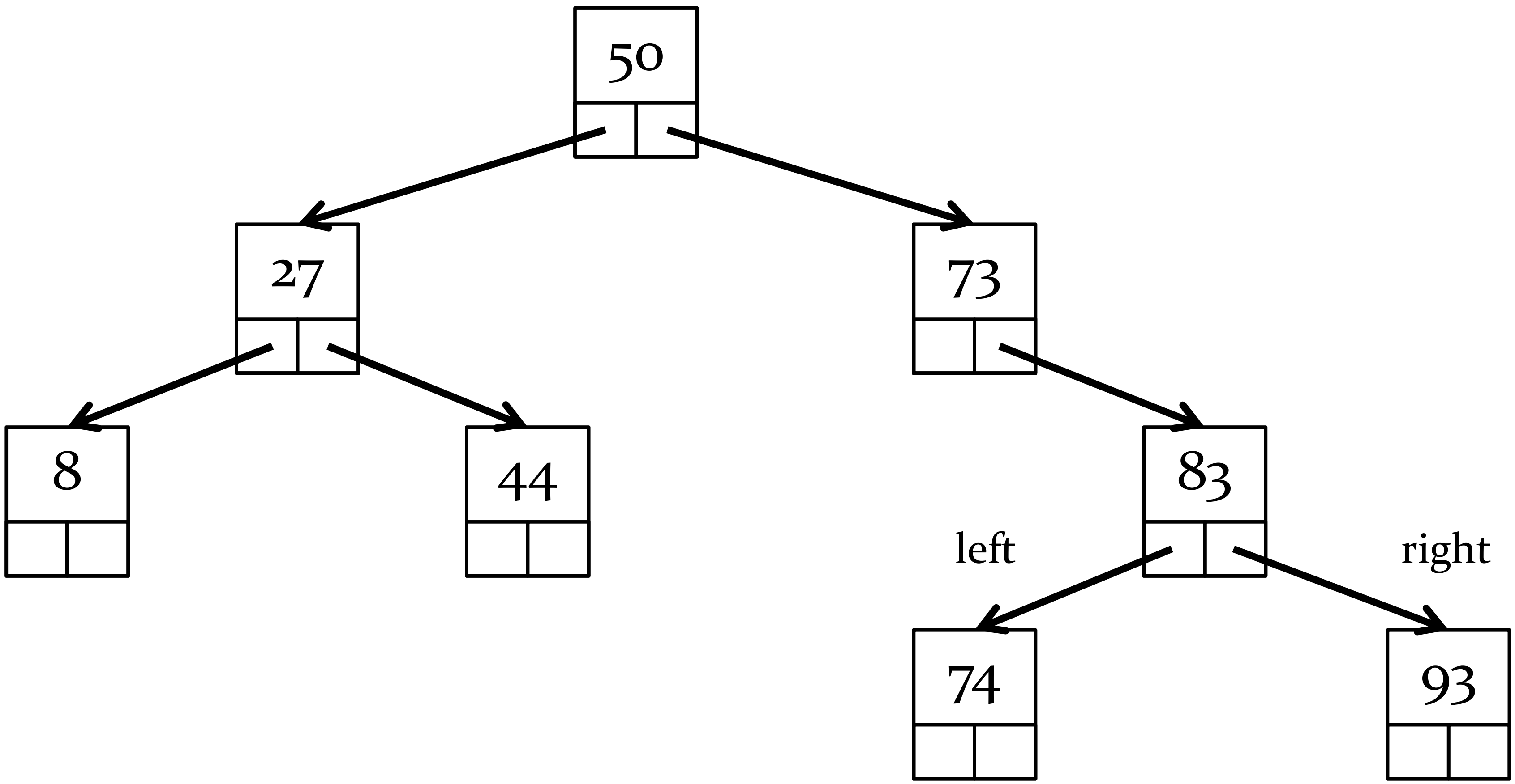
Binary Tree

- 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

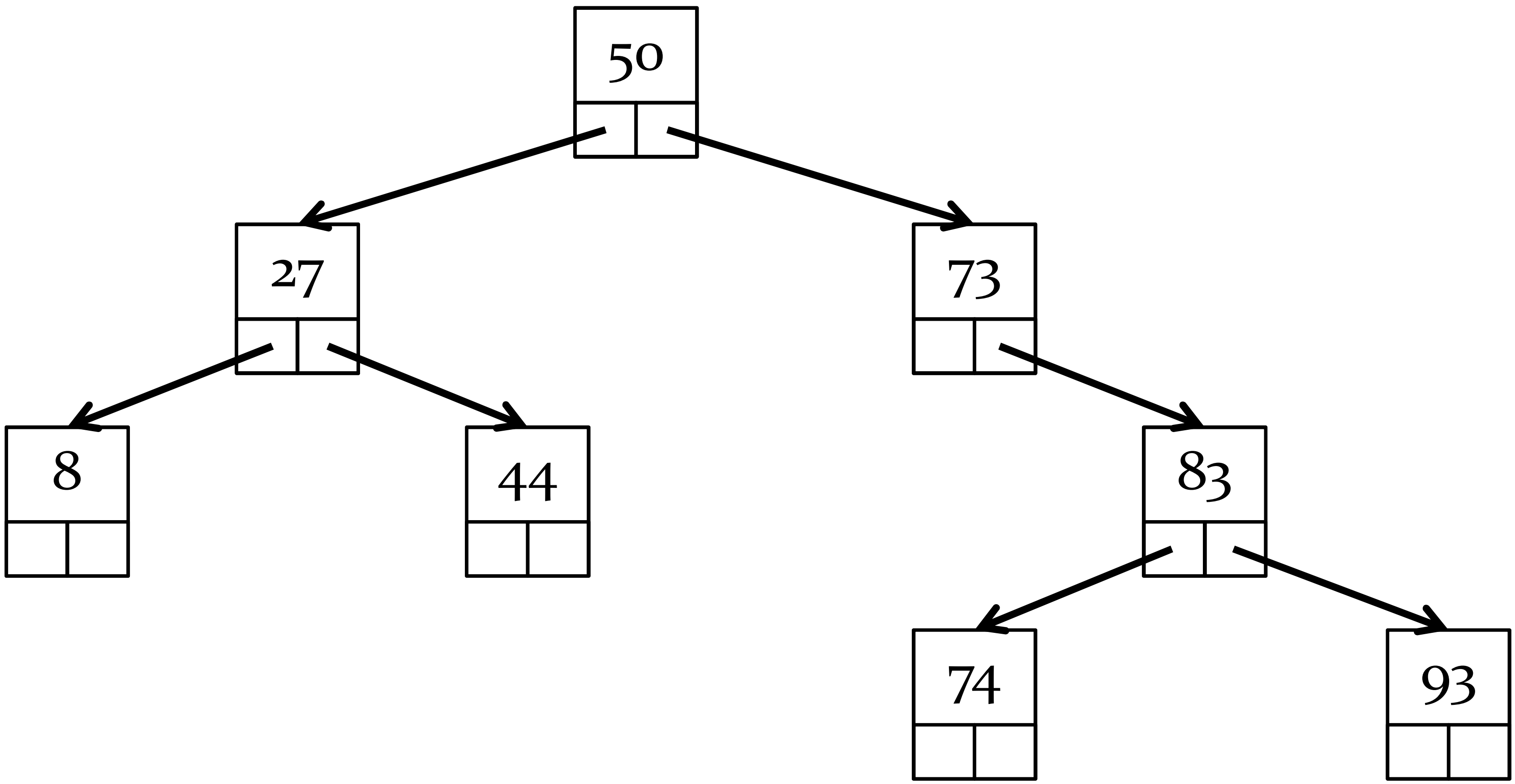
Binary Tree Algorithms

- 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

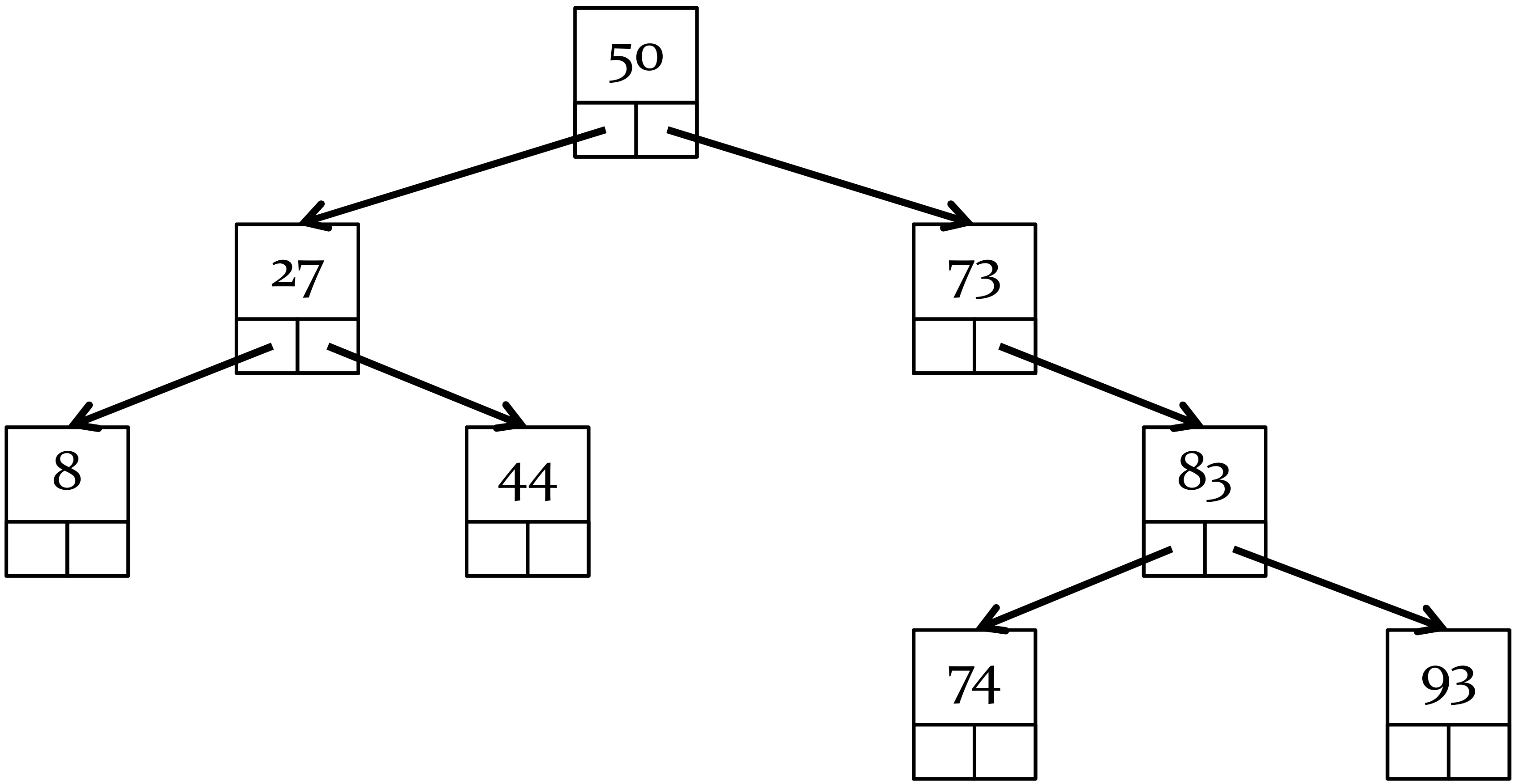
```java
public static <E> boolean contains(E element, Node<E> node) {
  if (node == null) {
    return false;
  }
  if (element.equals(node.data)) {
    return true;
  }
  boolean inLeftTree = contains(element, node.left);
  if (inLeftTree) {
    return true;
  }
  boolean inRightTree = contains(element,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

Iteration

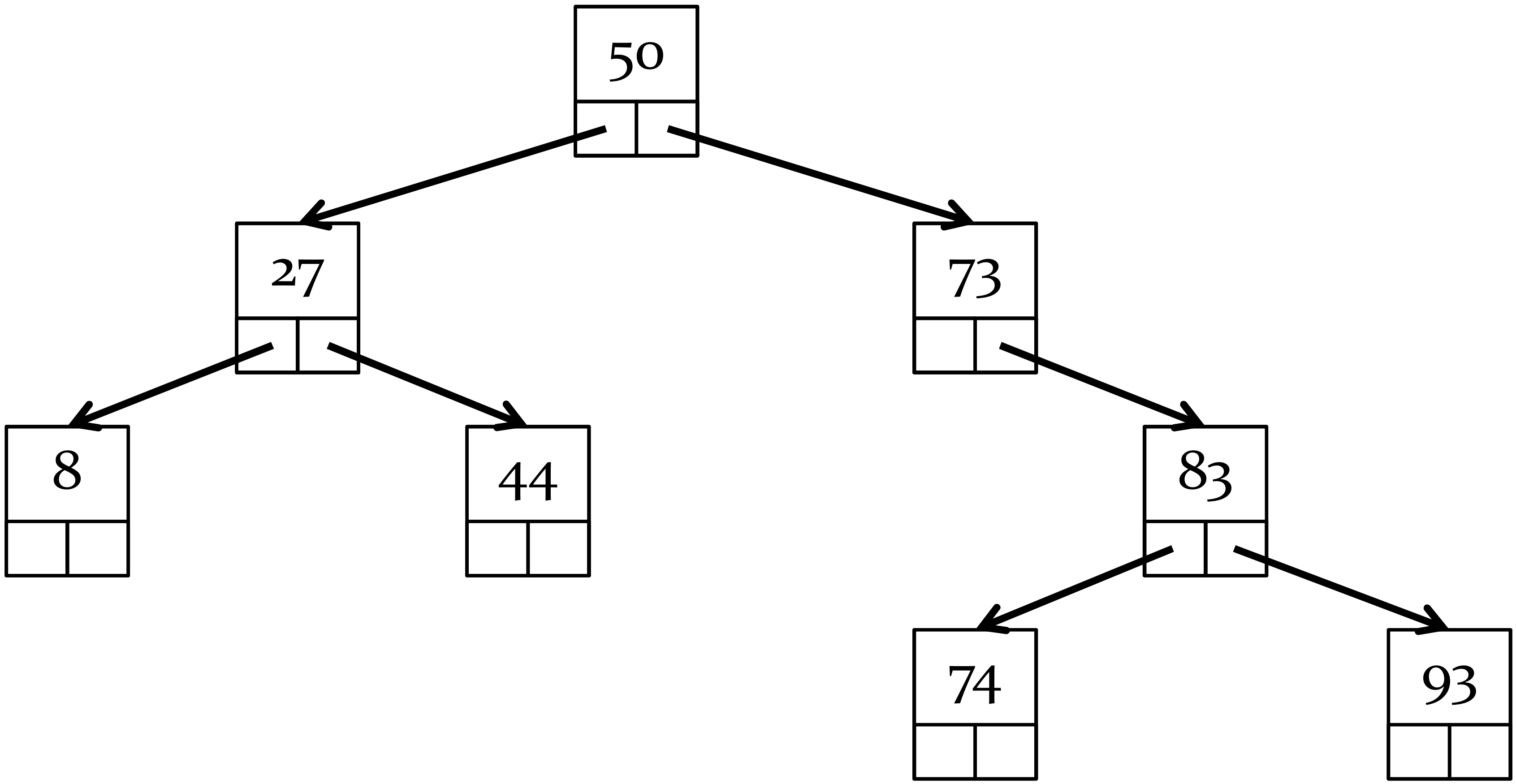
- visiting every element of the tree can also be done recursively
- 3 possibilities based on when the root is visited
  - inorder
    - visit left child, then root, then right child
  - preorder
    - visit root, then left child, then right child
  - postorder
    - visit left child, then right child,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



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



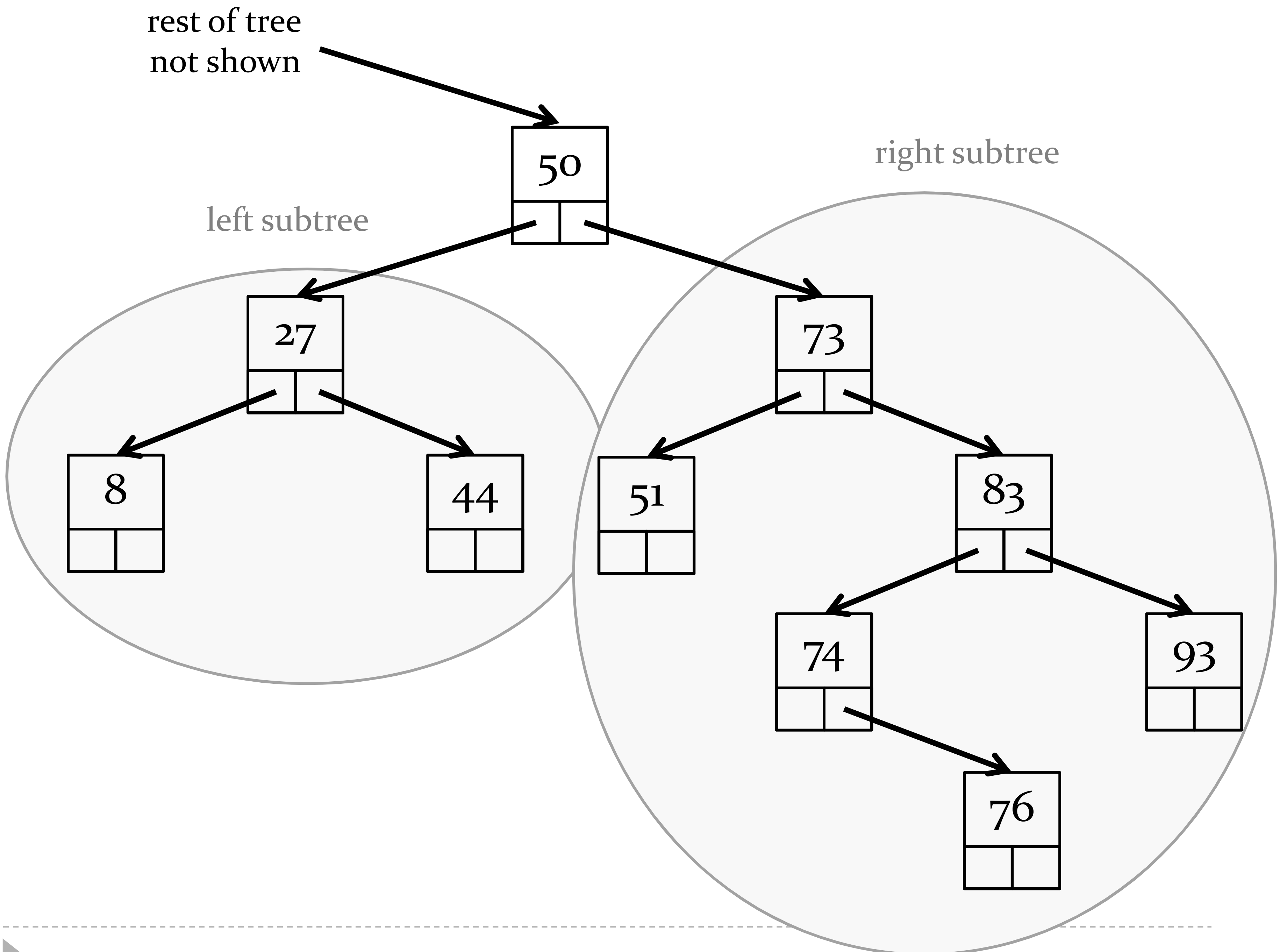
preorder: 50, 27, 8, 44, 73, 83, 74, 93



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

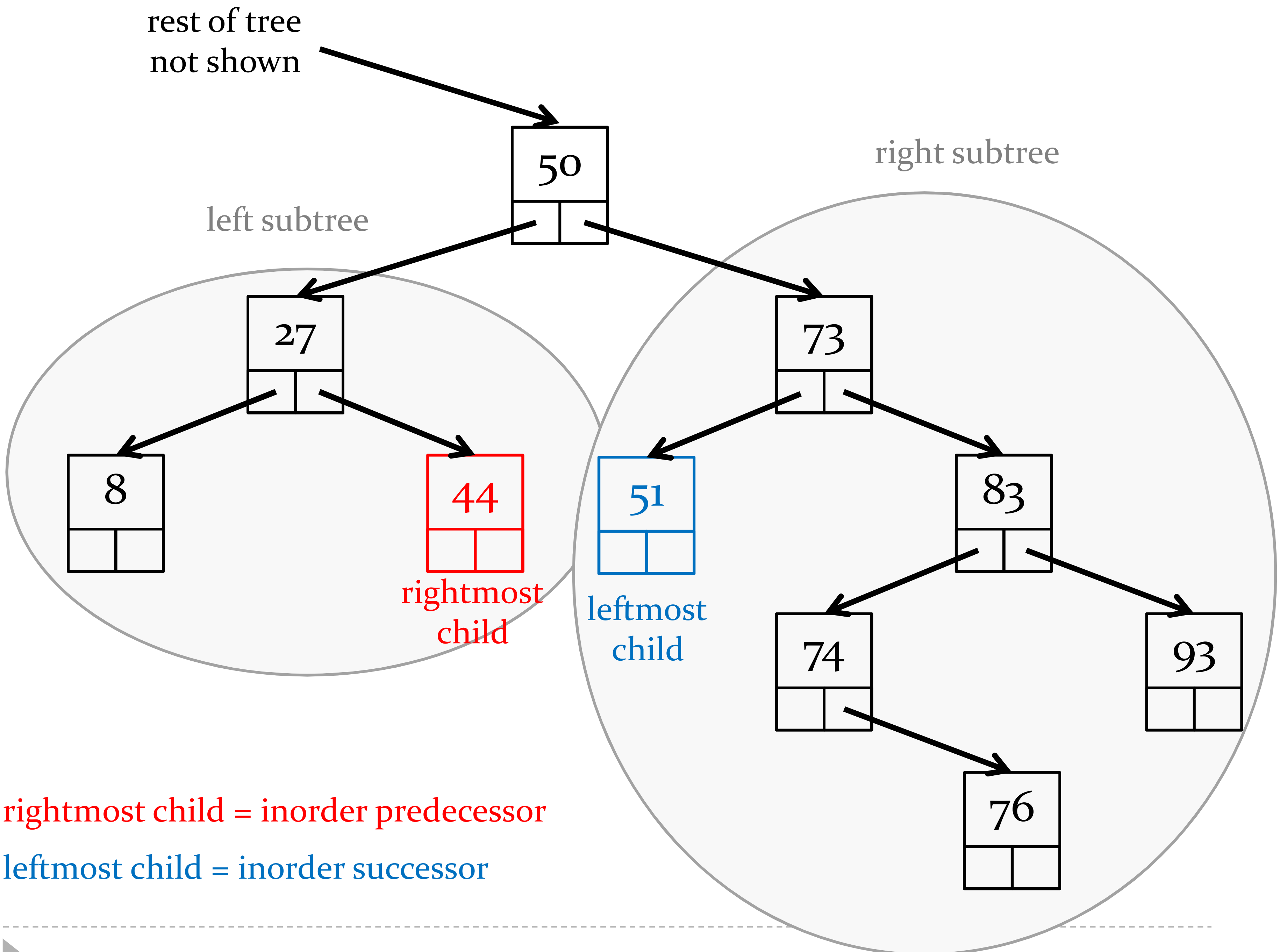
Binary Search Trees (BST)

- 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


rest of tree
not shown
50
left subtree
right subtree
27
73
8
44
51
83
74
93
76

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

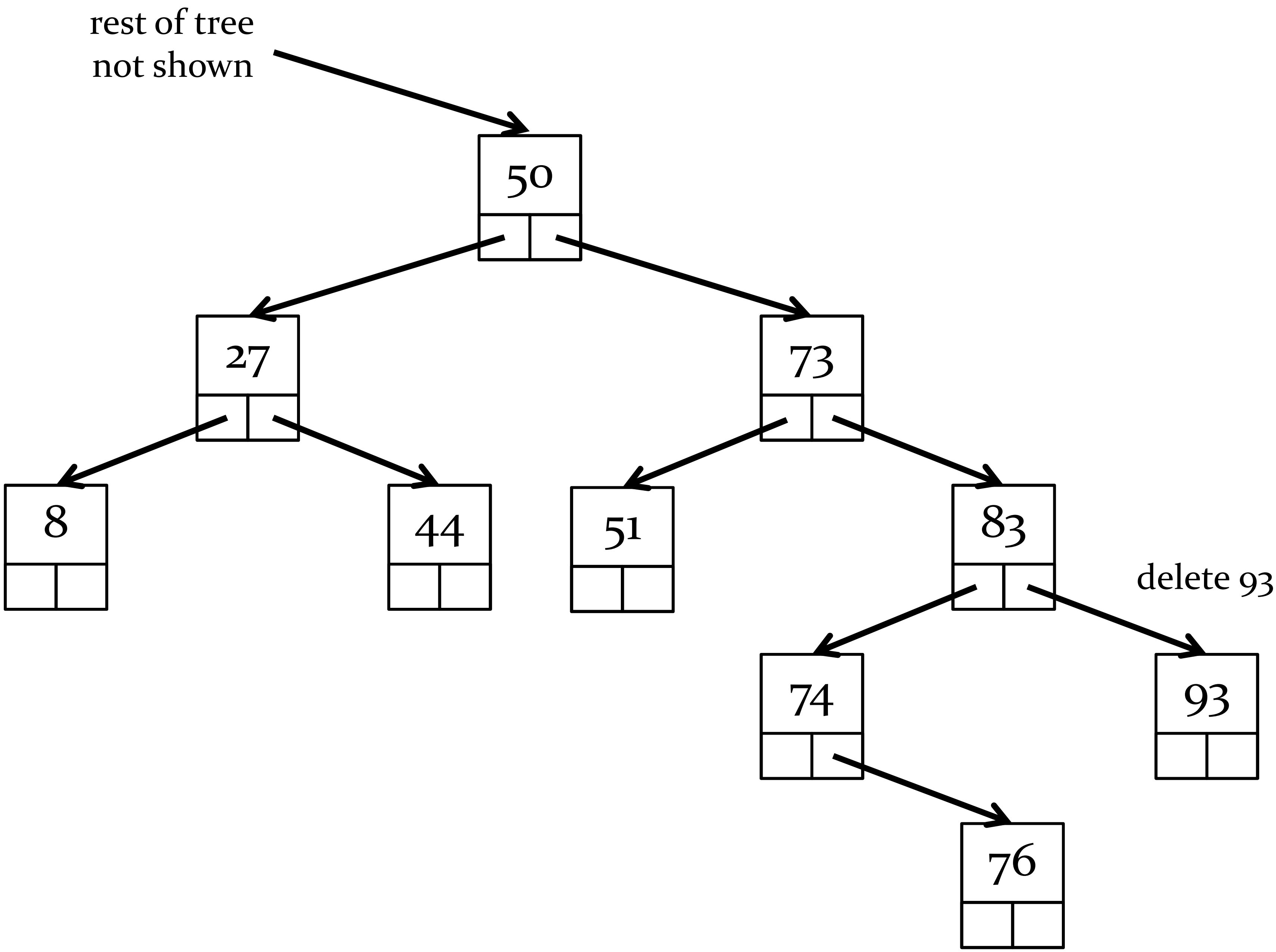
Deletion from a BST

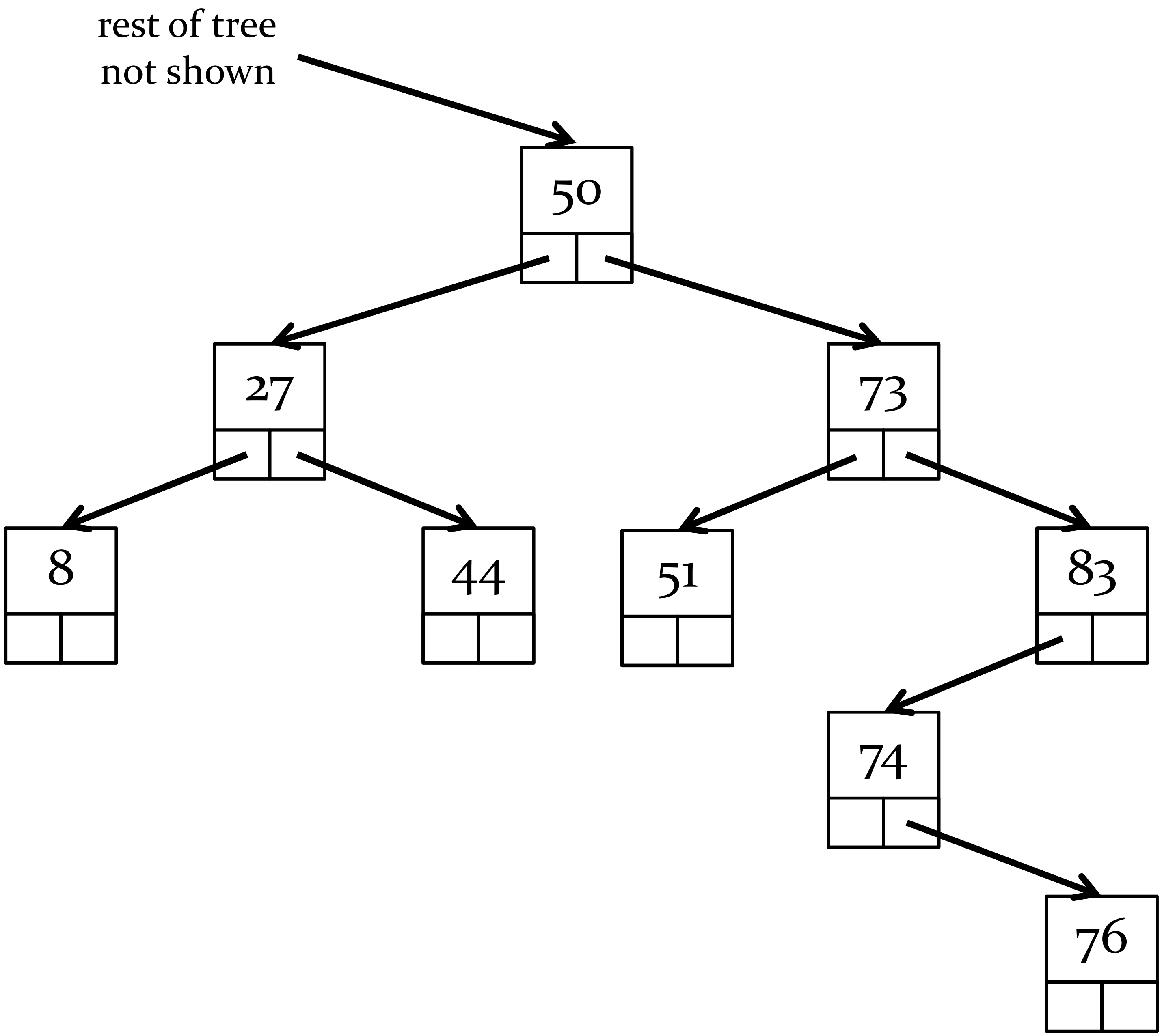
- to delete a node in a BST there are 3 cases to consider:
  1. deleting a leaf node
  2. deleting a node with one child
  3. deleting a node with two children

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

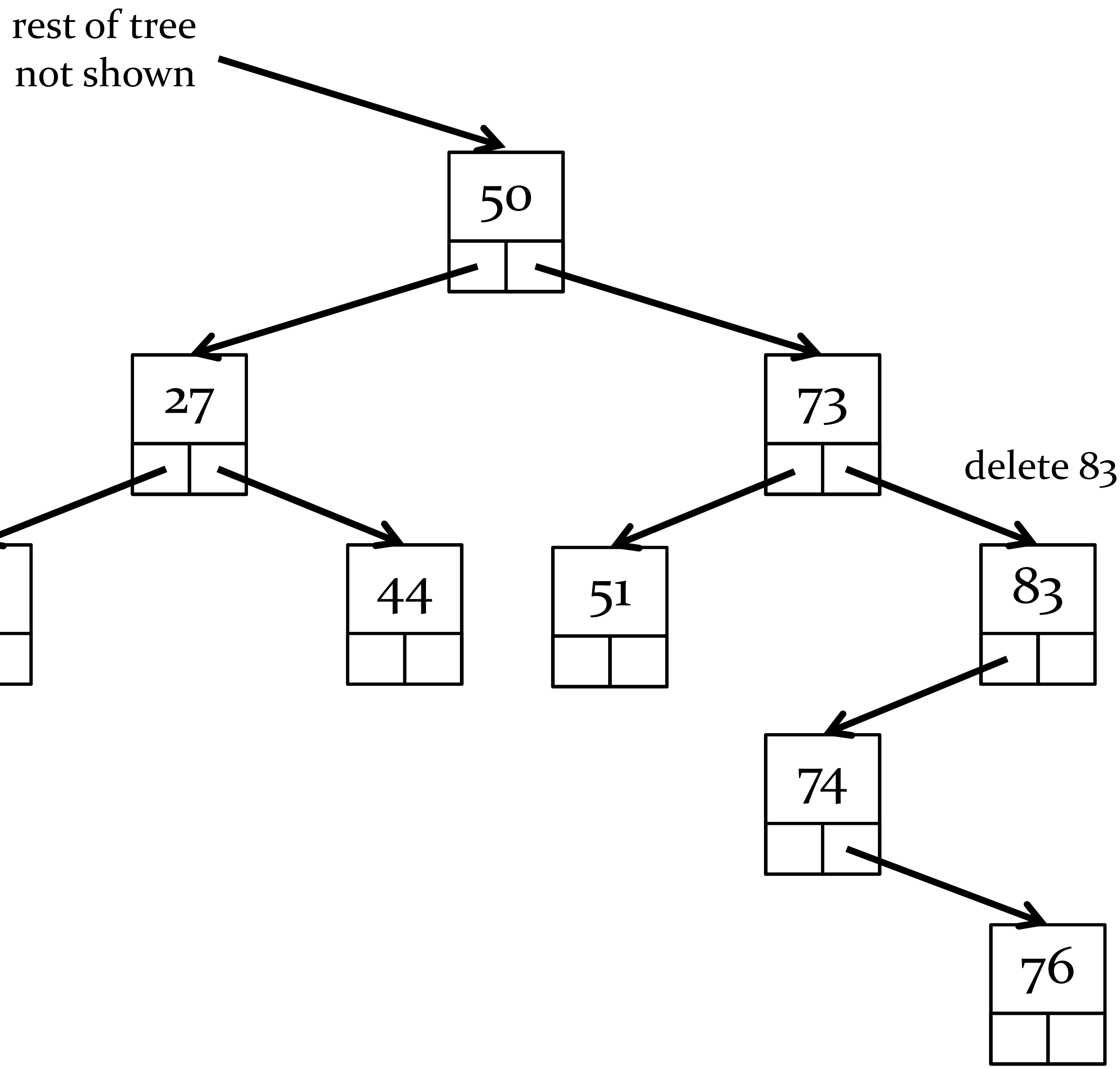

rest of tree
not shown
50
27
73
8
44
51
83
delete 93
74
93
76

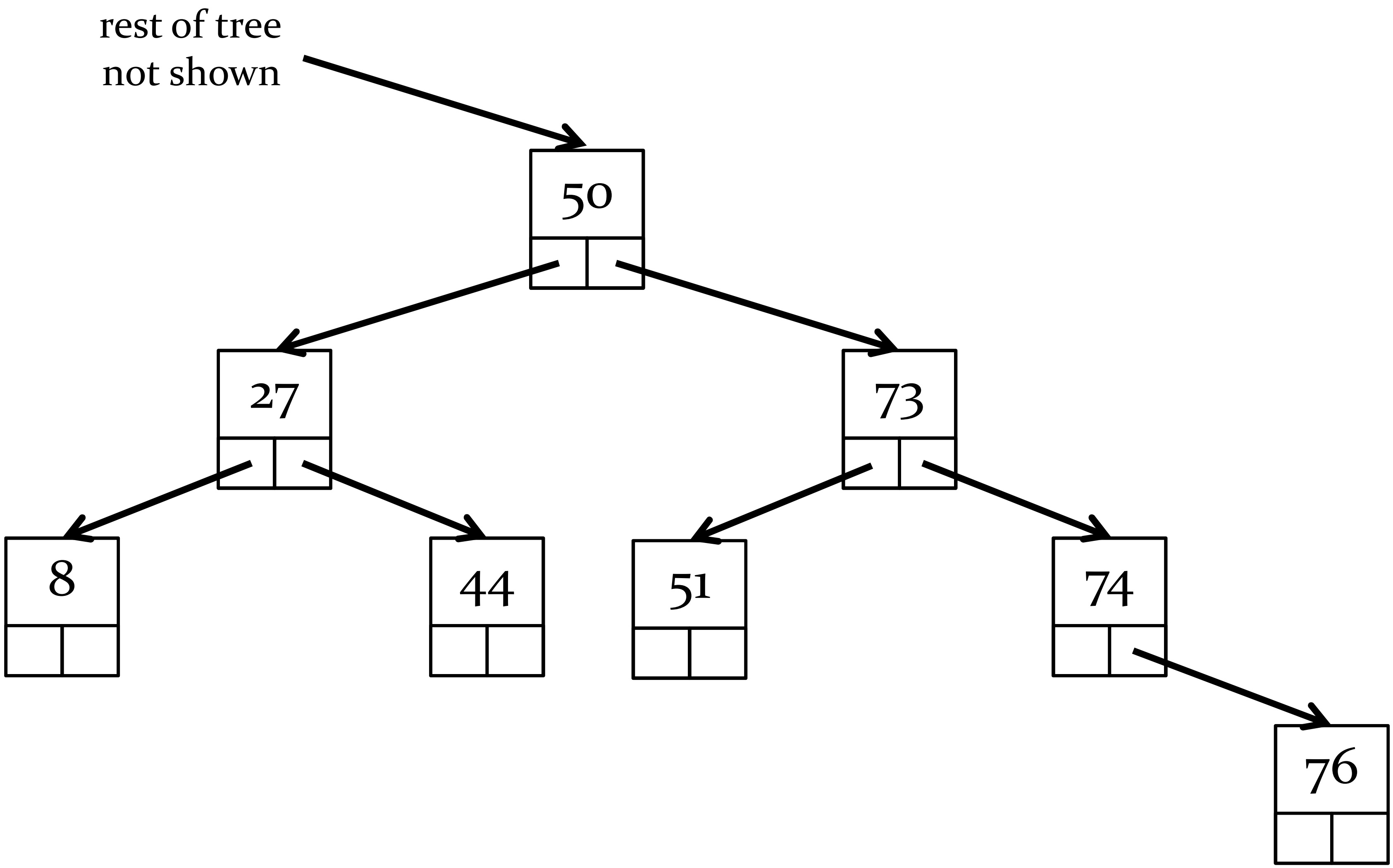

rest of tree
not shown
50
27
73
8
44
51
83
74
76

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

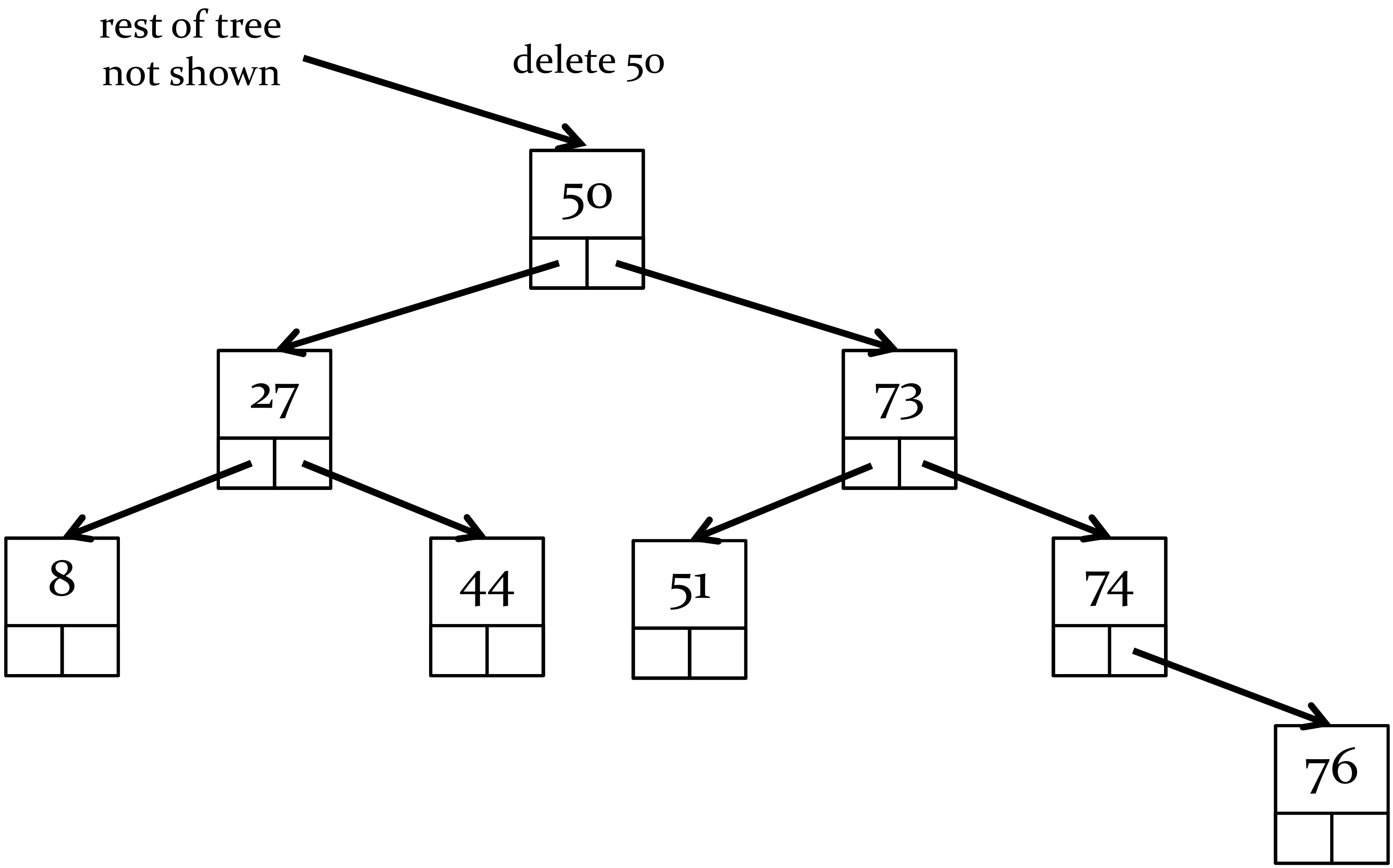

rest of tree
not shown
50
27
73
delete 83
8
44
51
83
74
76

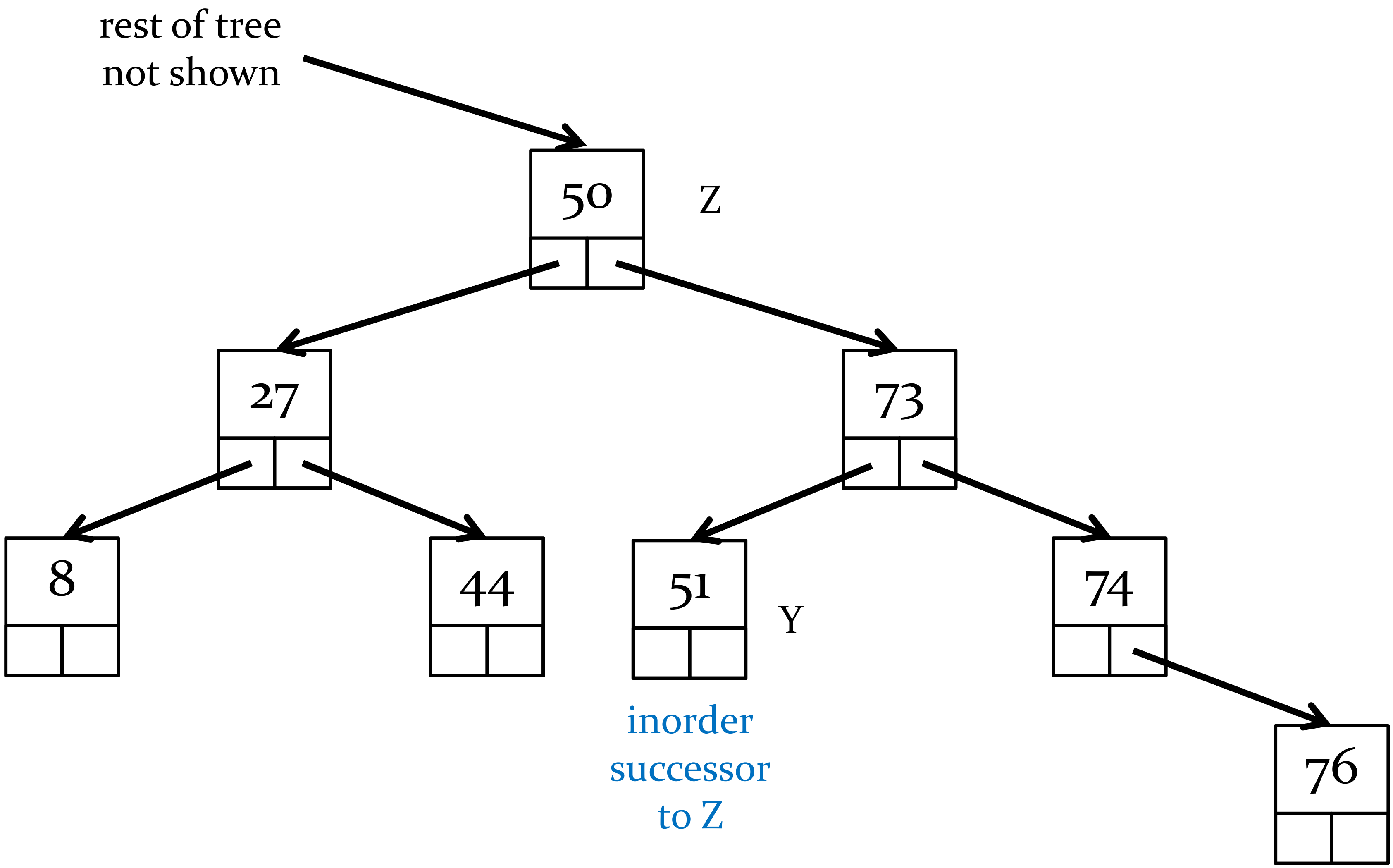

rest of tree
not shown
50
27
73
8
44
51
74
76

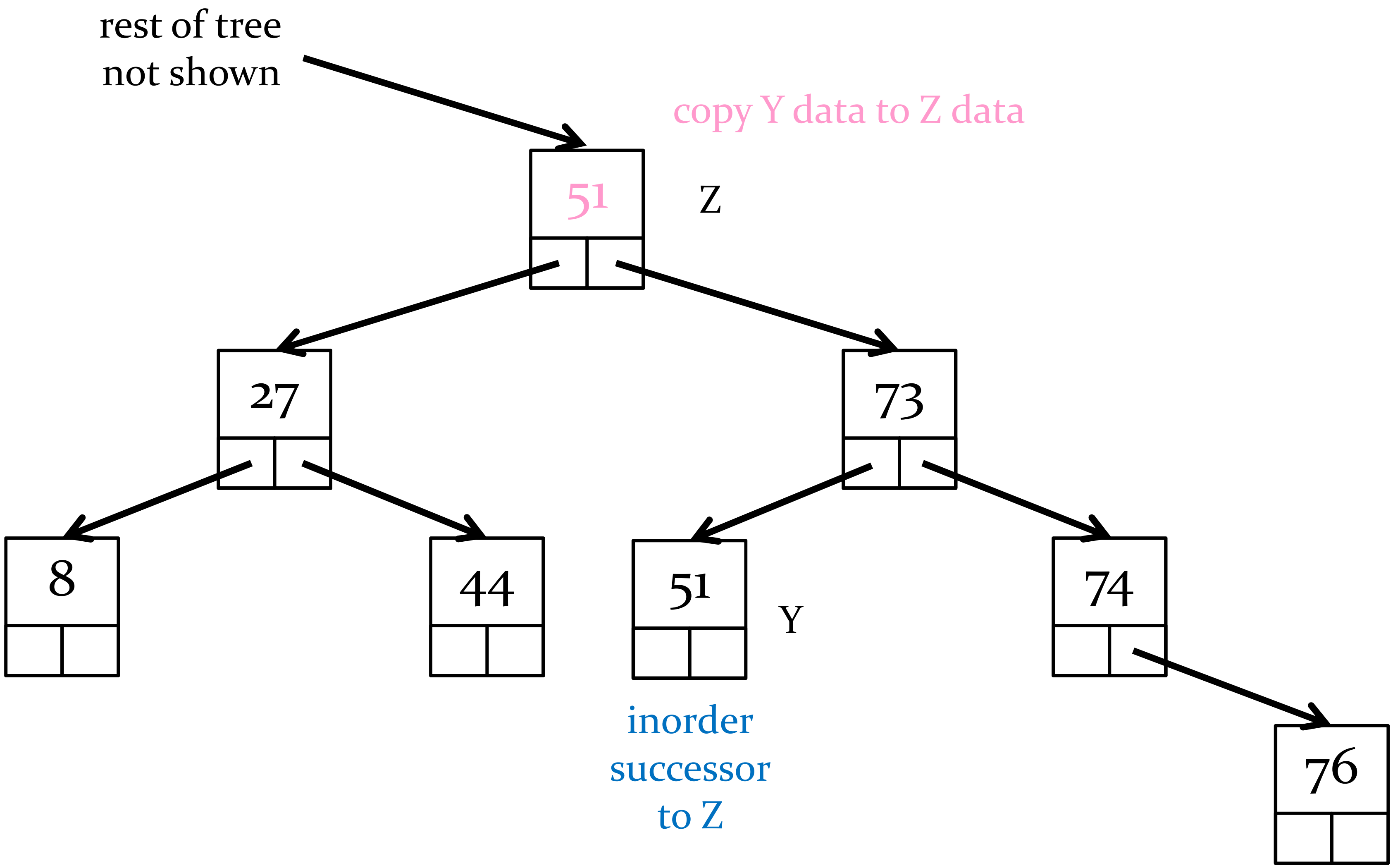
Deleting a Node with Two Children

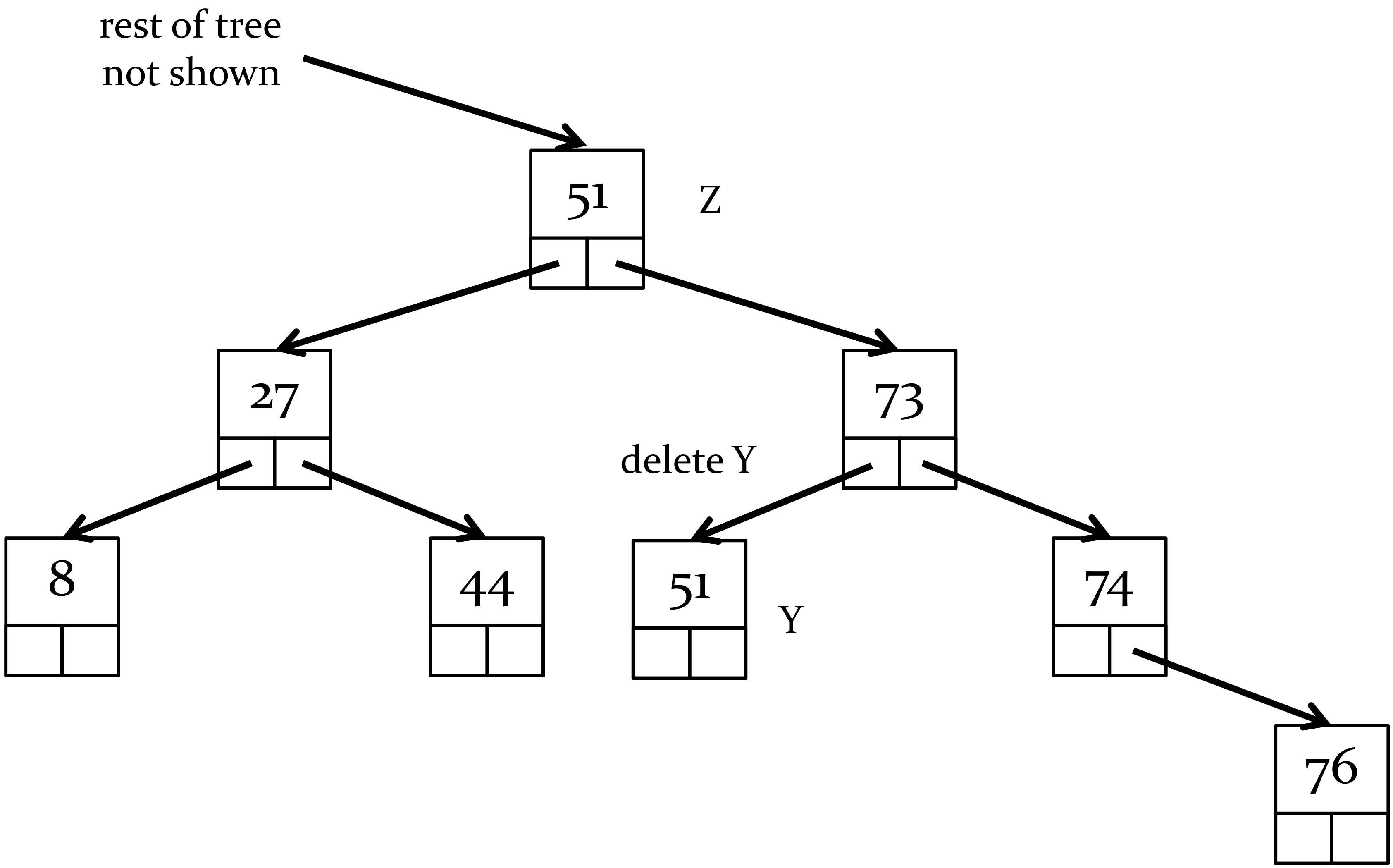
- deleting a node with two children is a little trickier
  - call the node to be deleted Z
  - find the inorder predecessor OR the inorder successor
    - call this node Y
      - if the inorder predecessor does not exist, then you must find the inorder successor (and vice versa)
  - copy the data element of Y into the data element of Z
  - delete Y

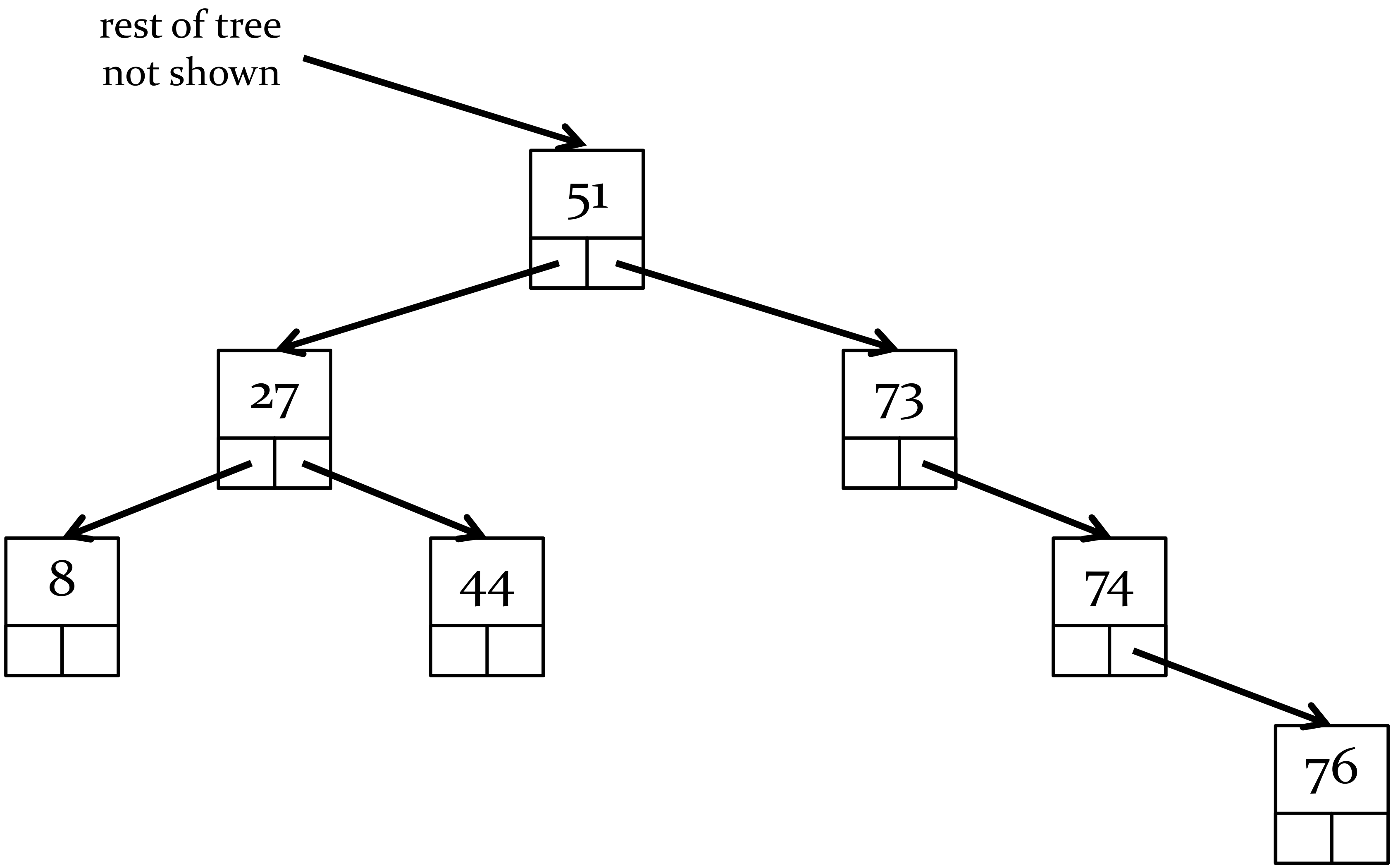
- e.g., delete 50


rest of tree
not shown
delete 50
50
27
73
8
44
51
74
76


rest of tree
not shown
50
Z
27
73
8
44
51
Y
74
inorder
successor
to Z
76


rest of tree
not shown
copy Y data to Z data
51
Z
27
73
8
44
51
Y
74
inorder
successor
to Z
76


rest of tree
not shown
51
Z
27
73
delete Y
8
44
51
Y
74
76


rest of tree
not shown
51
27
73
8
44
74
76