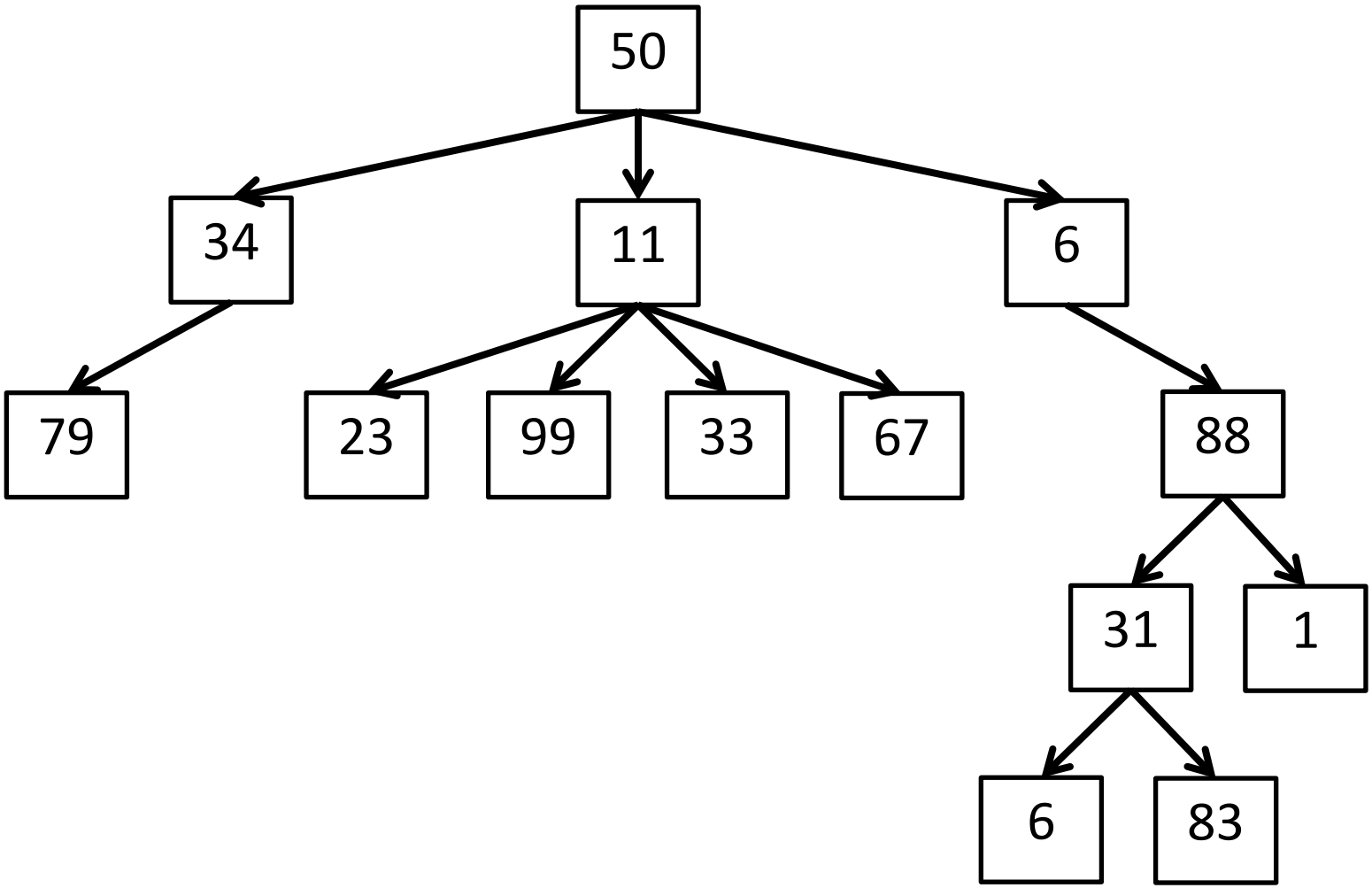
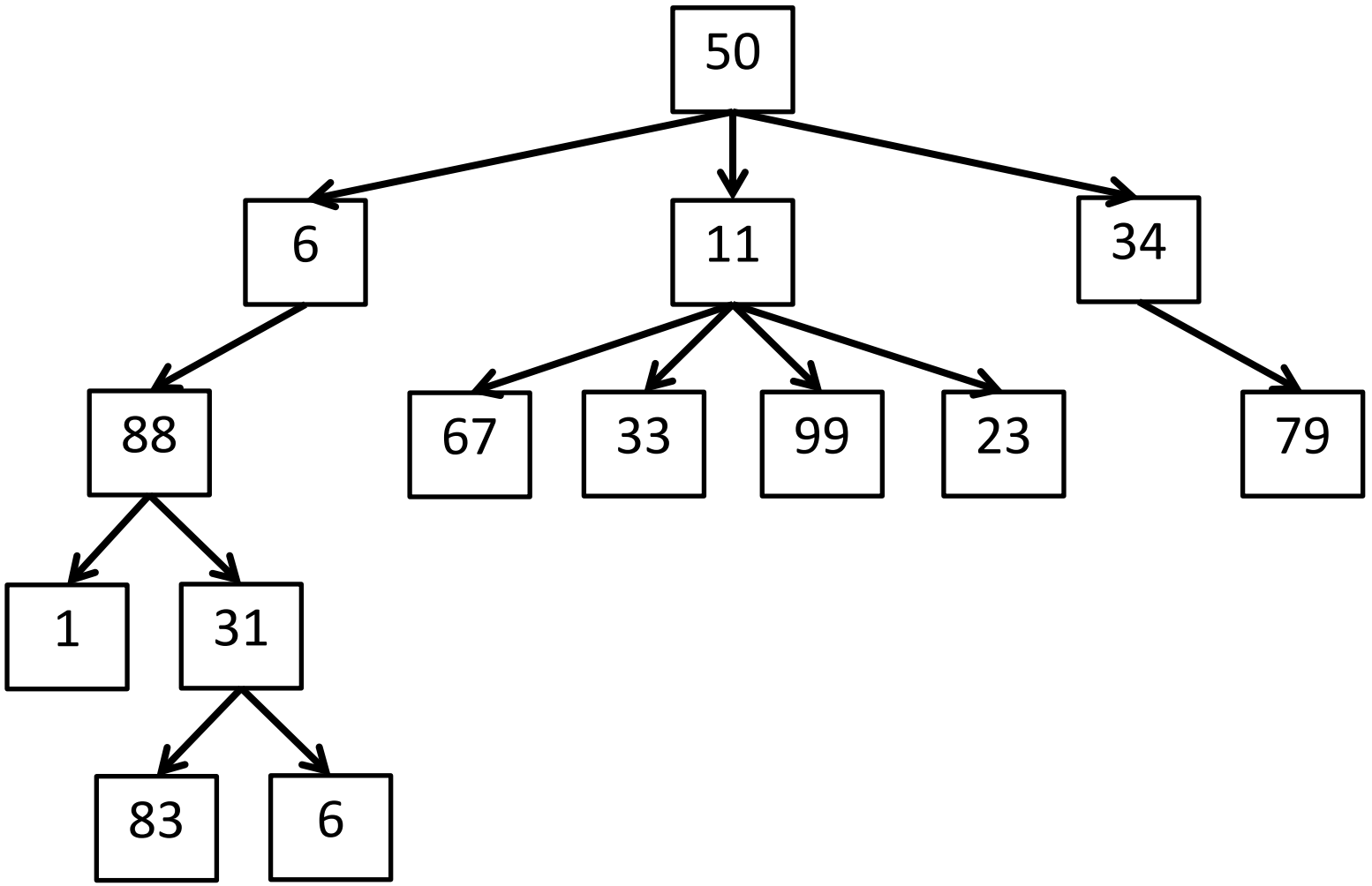
Navigating Trees

Based on slides by Prof. Burton Ma

Trees

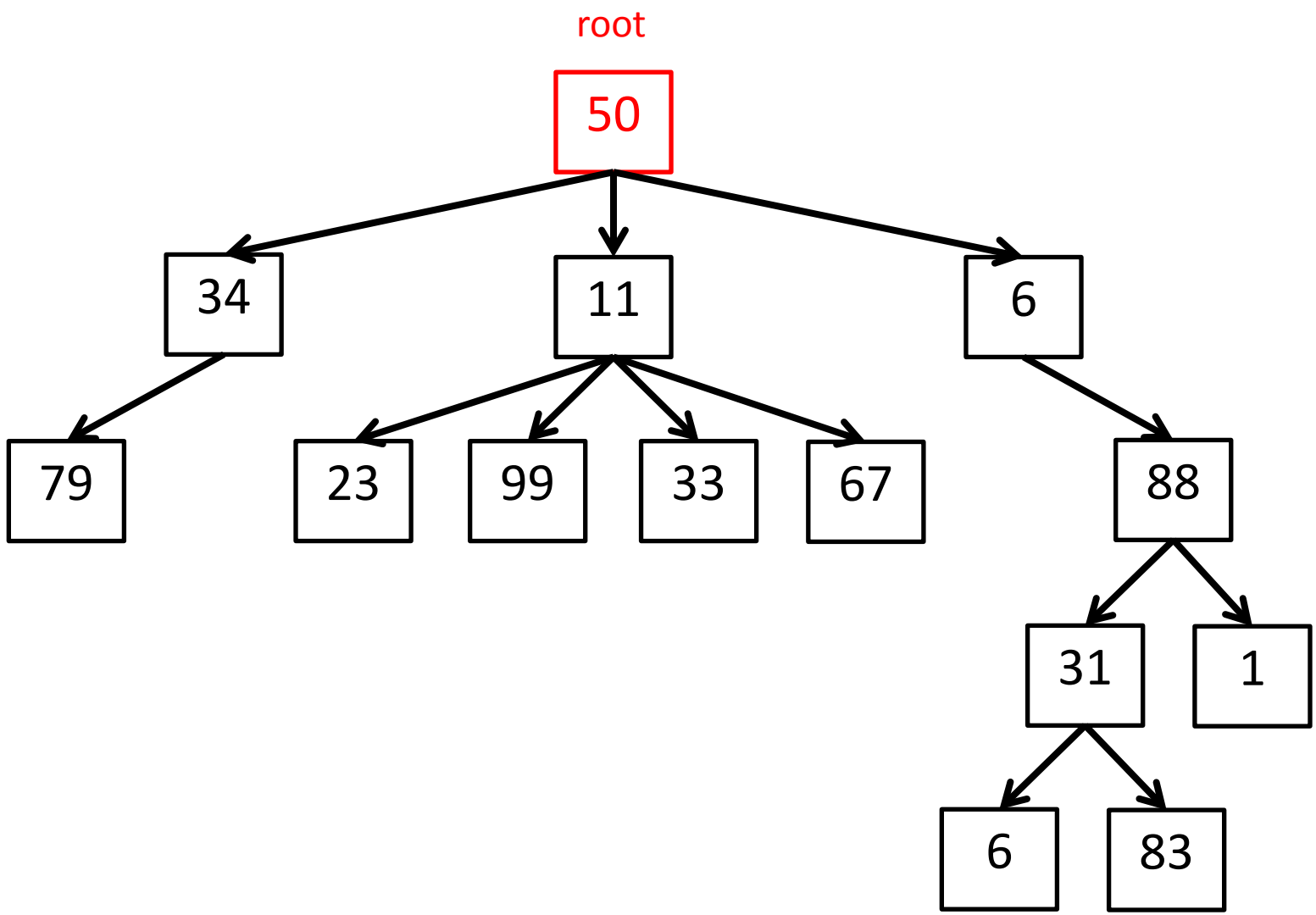
- A tree is a data structure made up of nodes
  - Each node stores data
  - Each node has links to zero or more nodes in the next level of the tree
    - Children of the node
  - Each node has exactly one parent node
    - Except for the root node


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

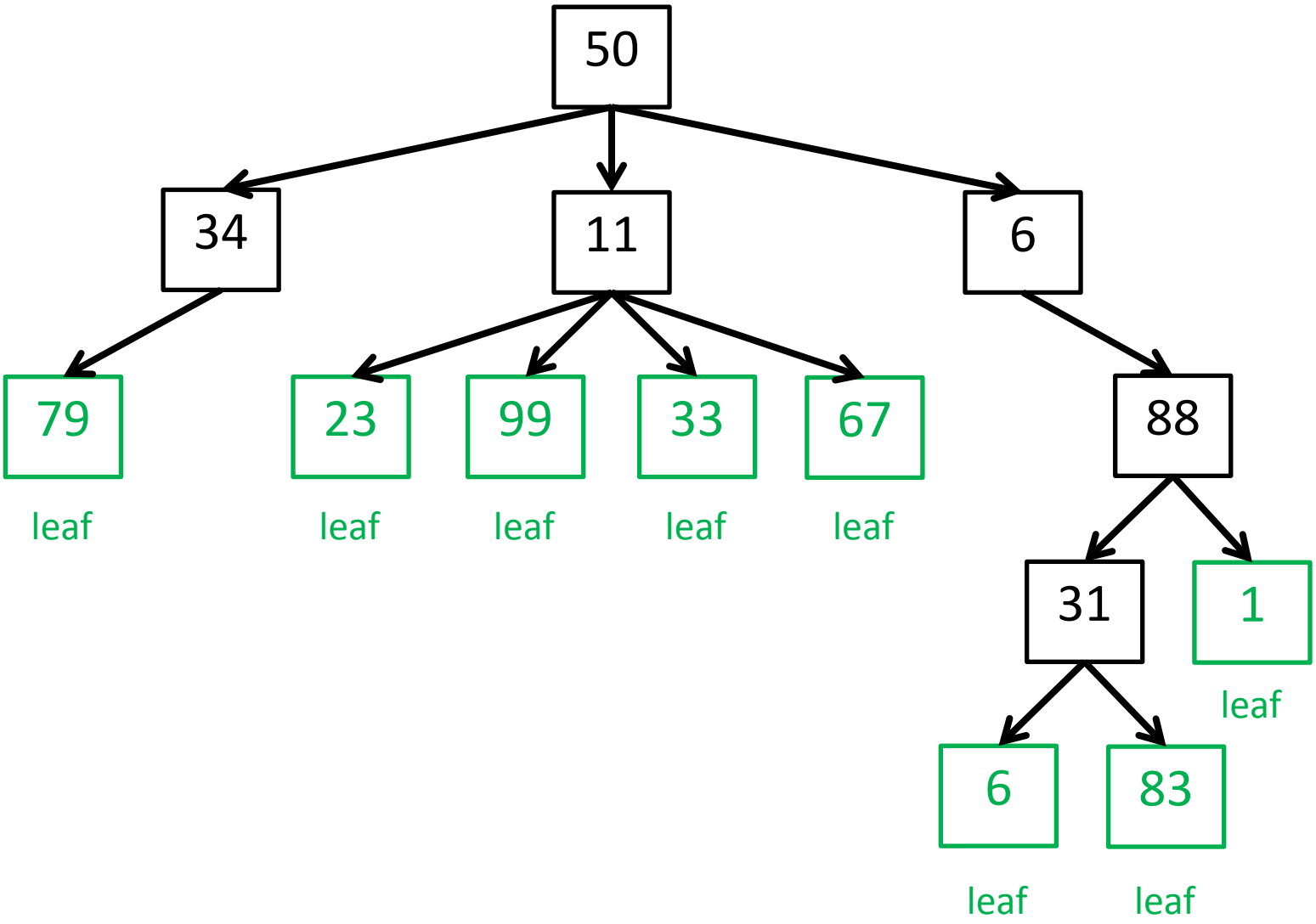
Trees

- 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

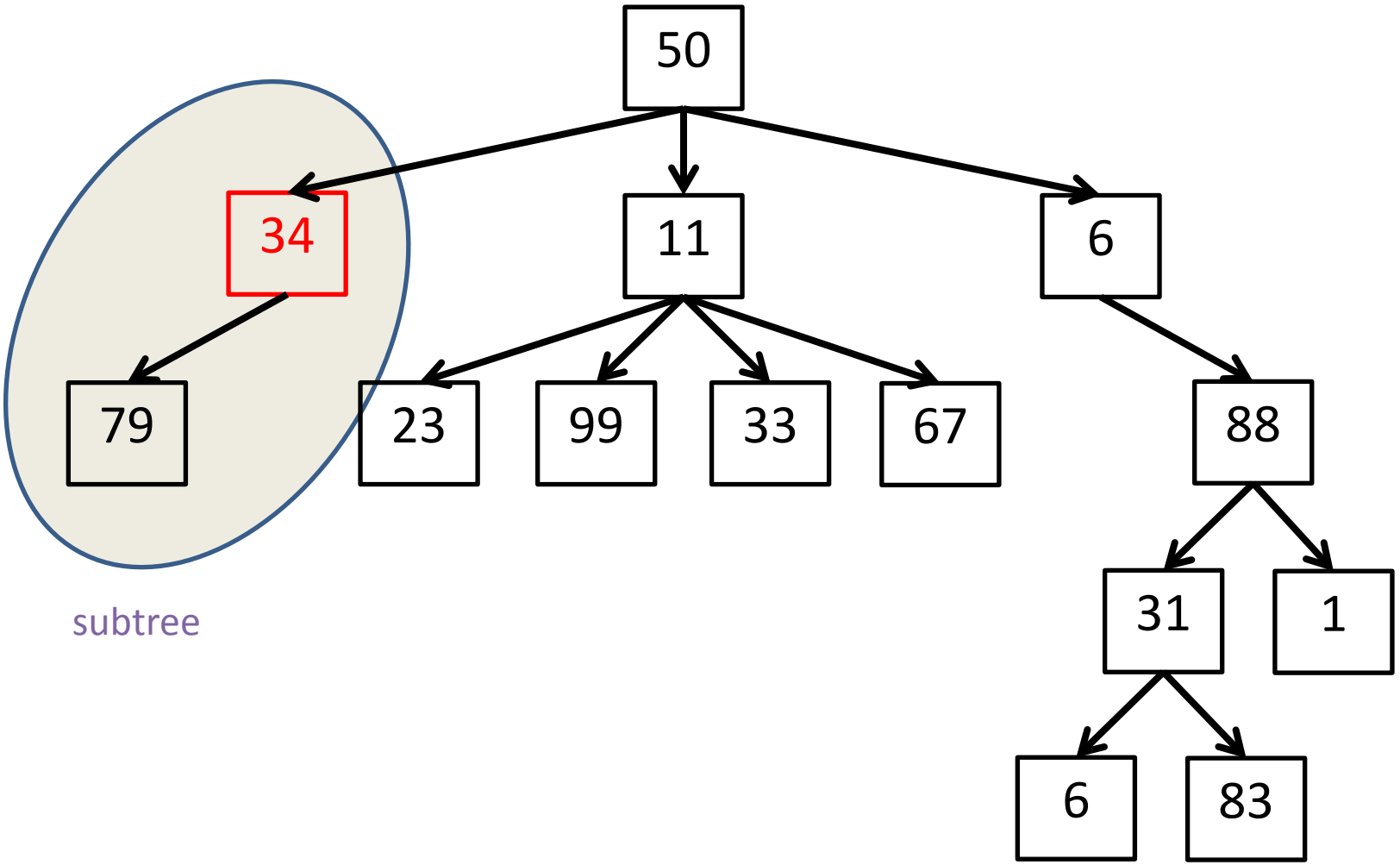
Trees

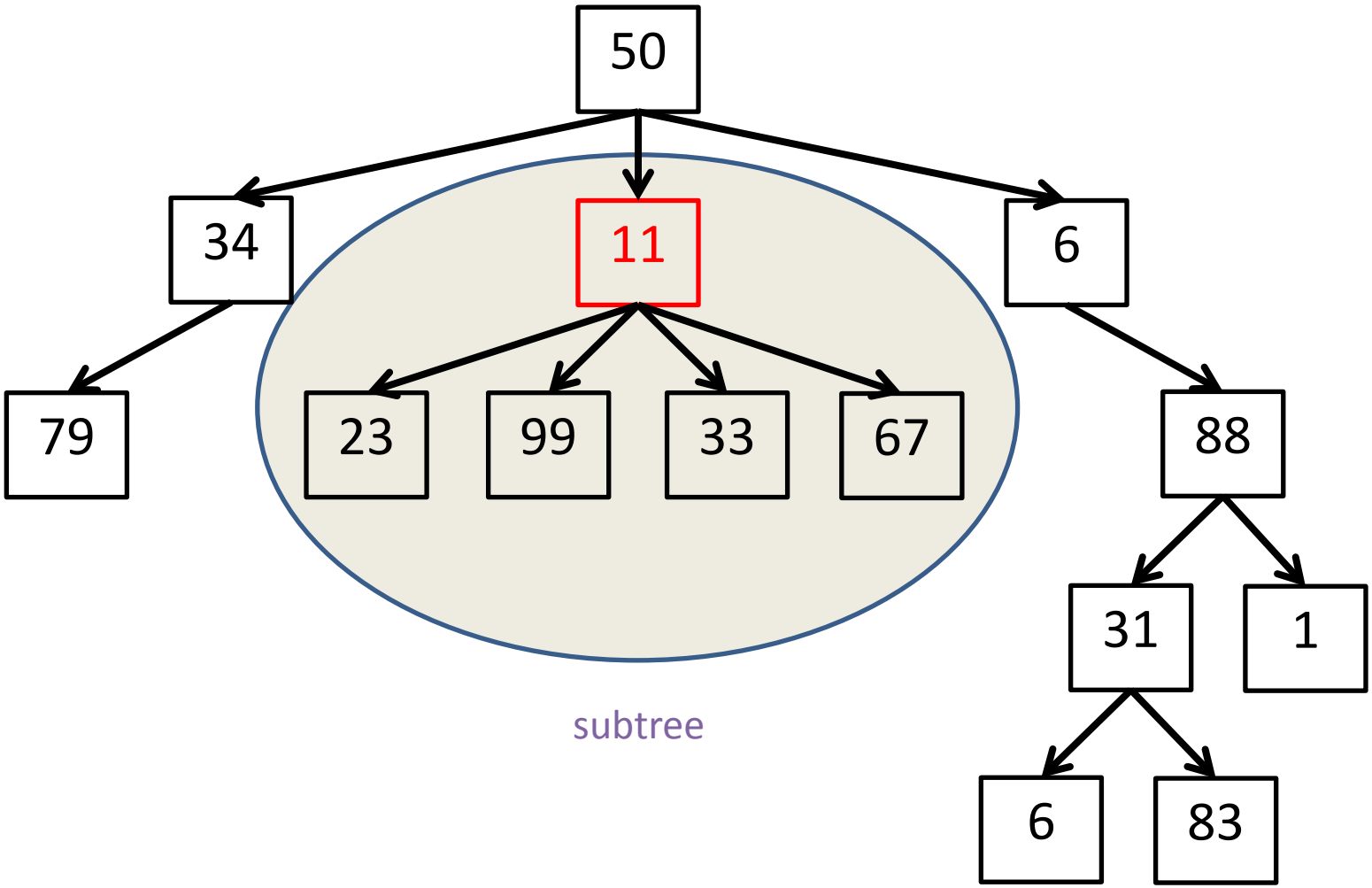
- A node without any children is called a leaf

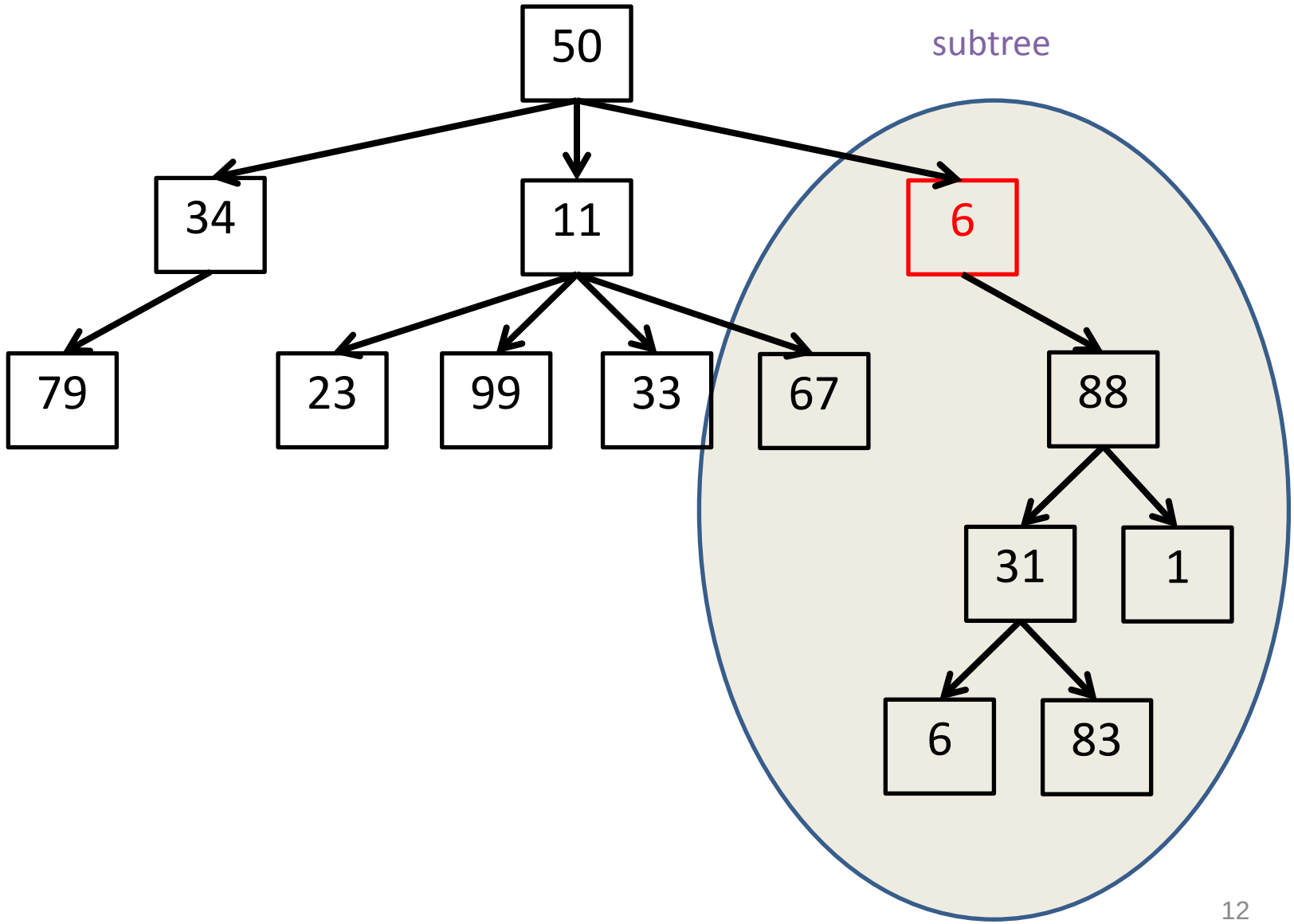

50
34
11
6
79
23
99
33
67
88
leaf
leaf
leaf
leaf
leaf
31
1
leaf
6
83
leaf
leaf

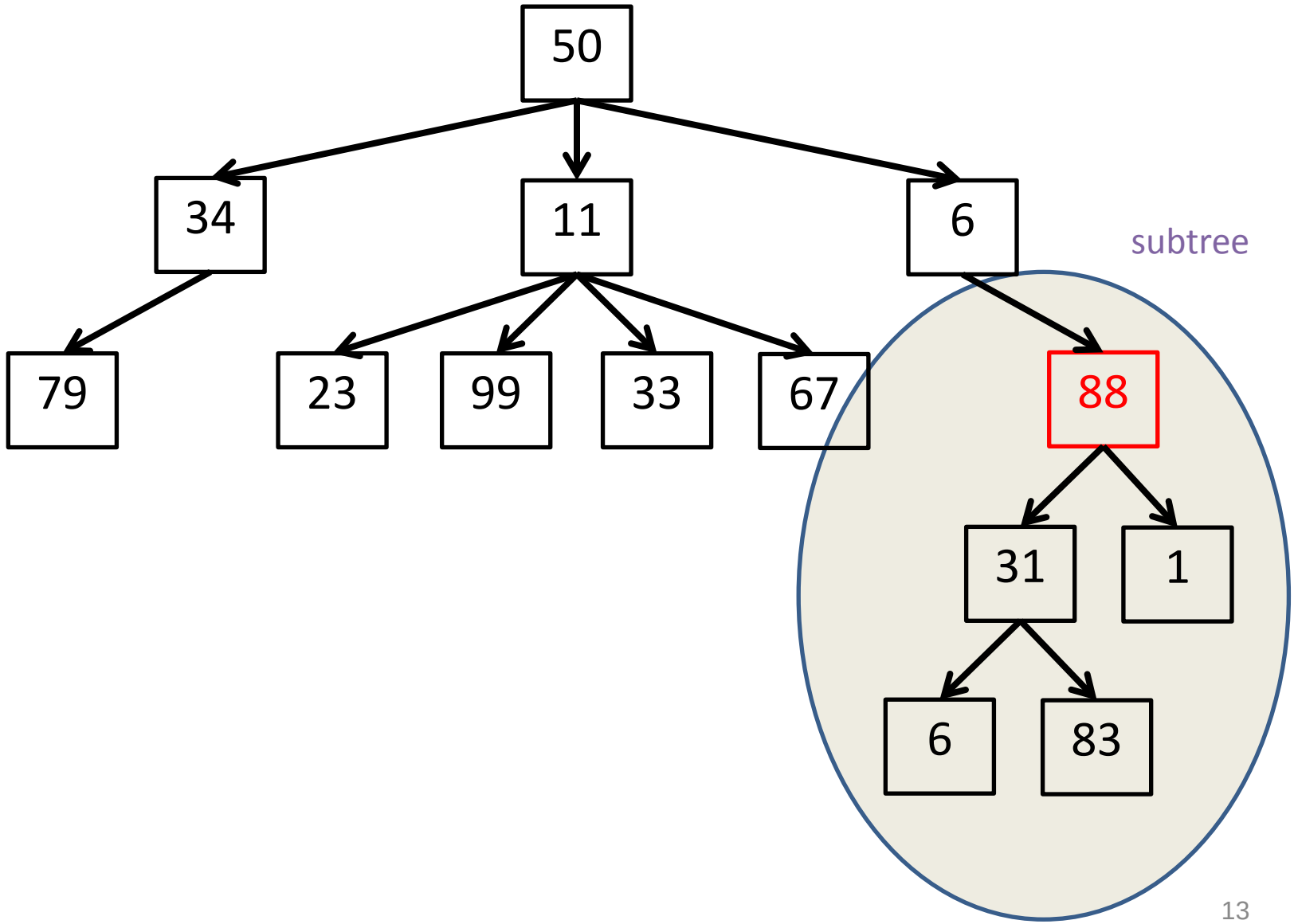
Trees

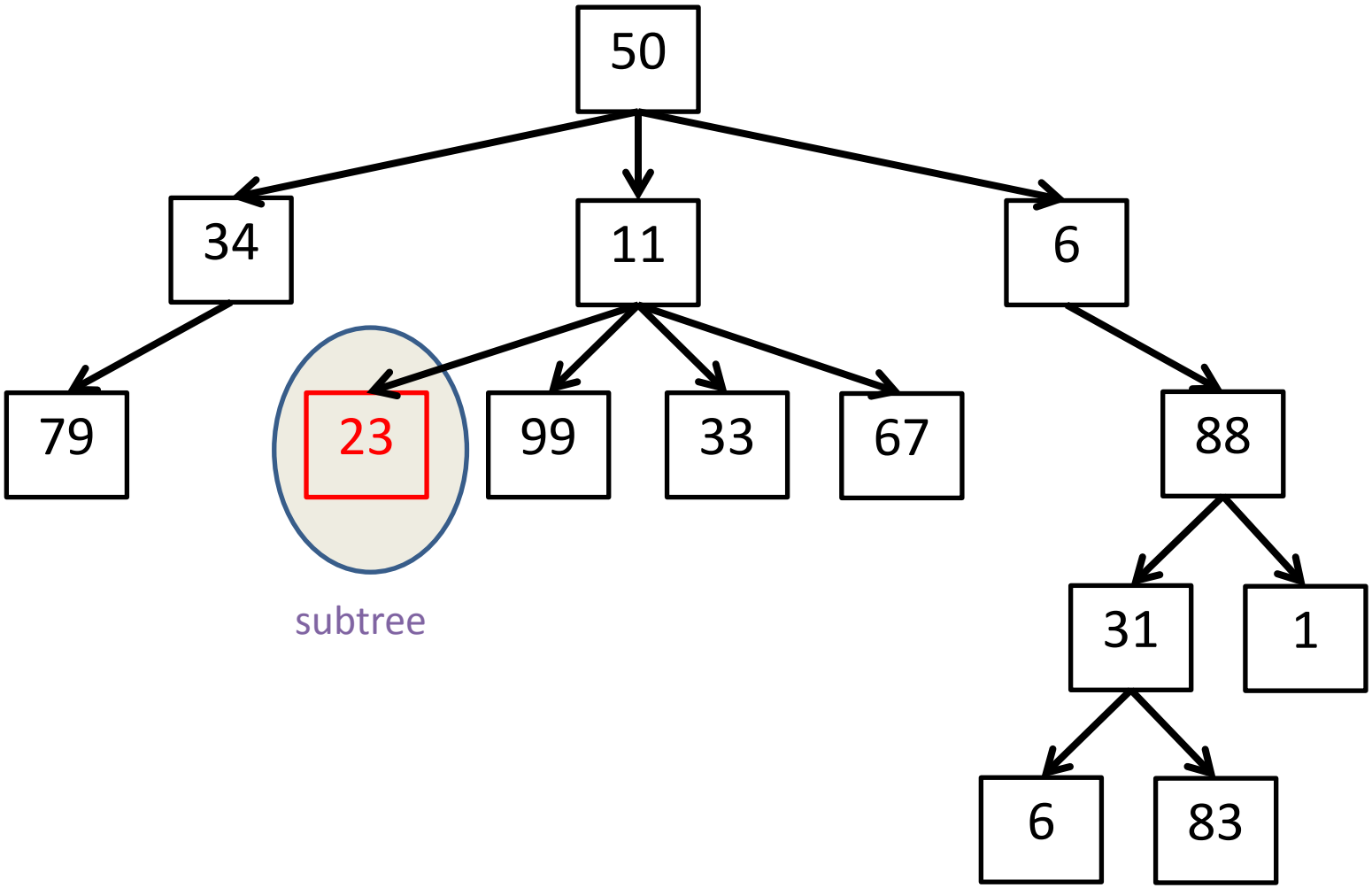
- The recursive structure of a tree means that every node is the root of a tree


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

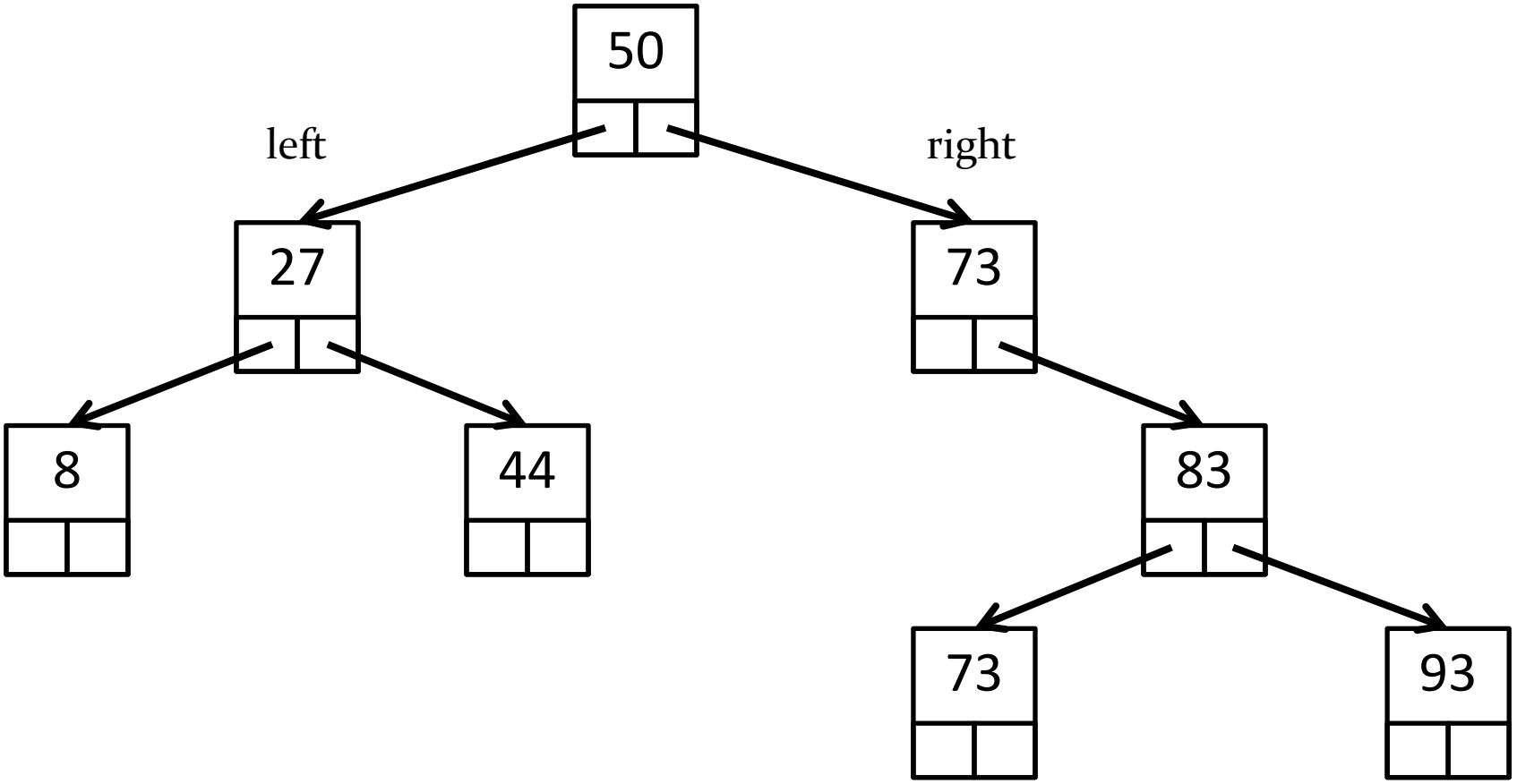

50
subtree
34
11
6
79
23
99
33
67
88
31
1
6
83

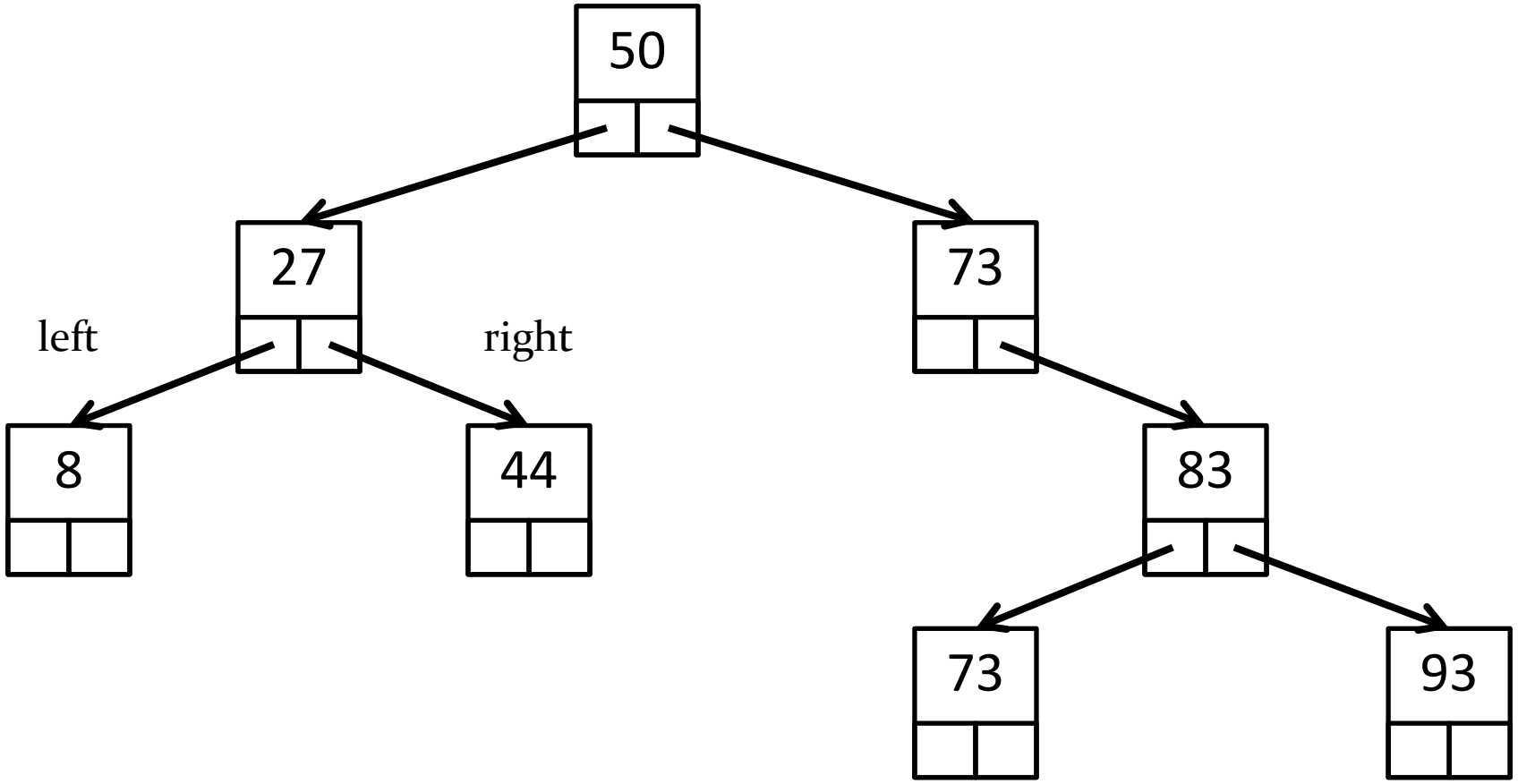

50
34
11
6
79
23
99
33
67
88
31
1
6
83
subtree

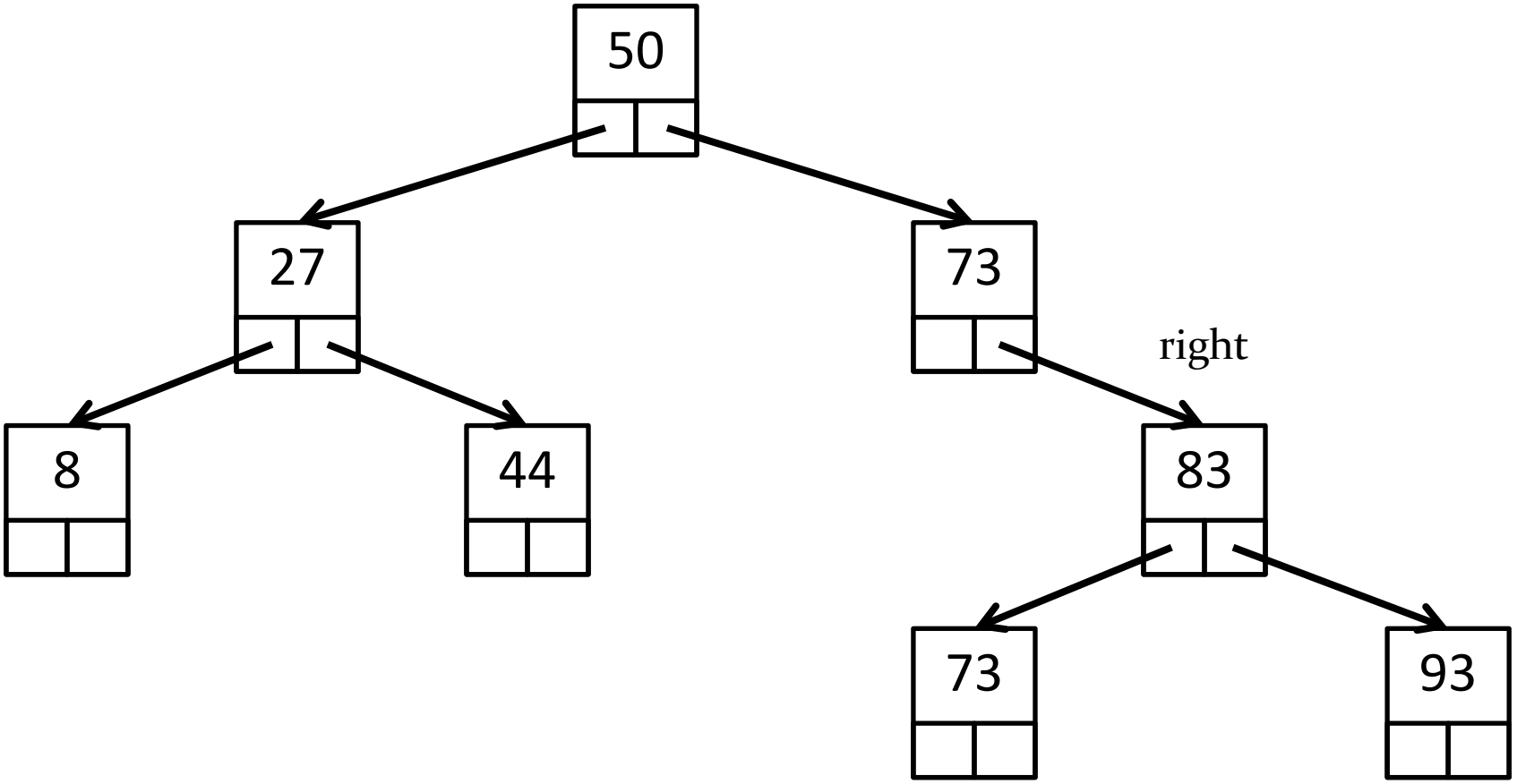

50
34
11
6
79
23
99
33
67
88
subtree
31
1
6
83

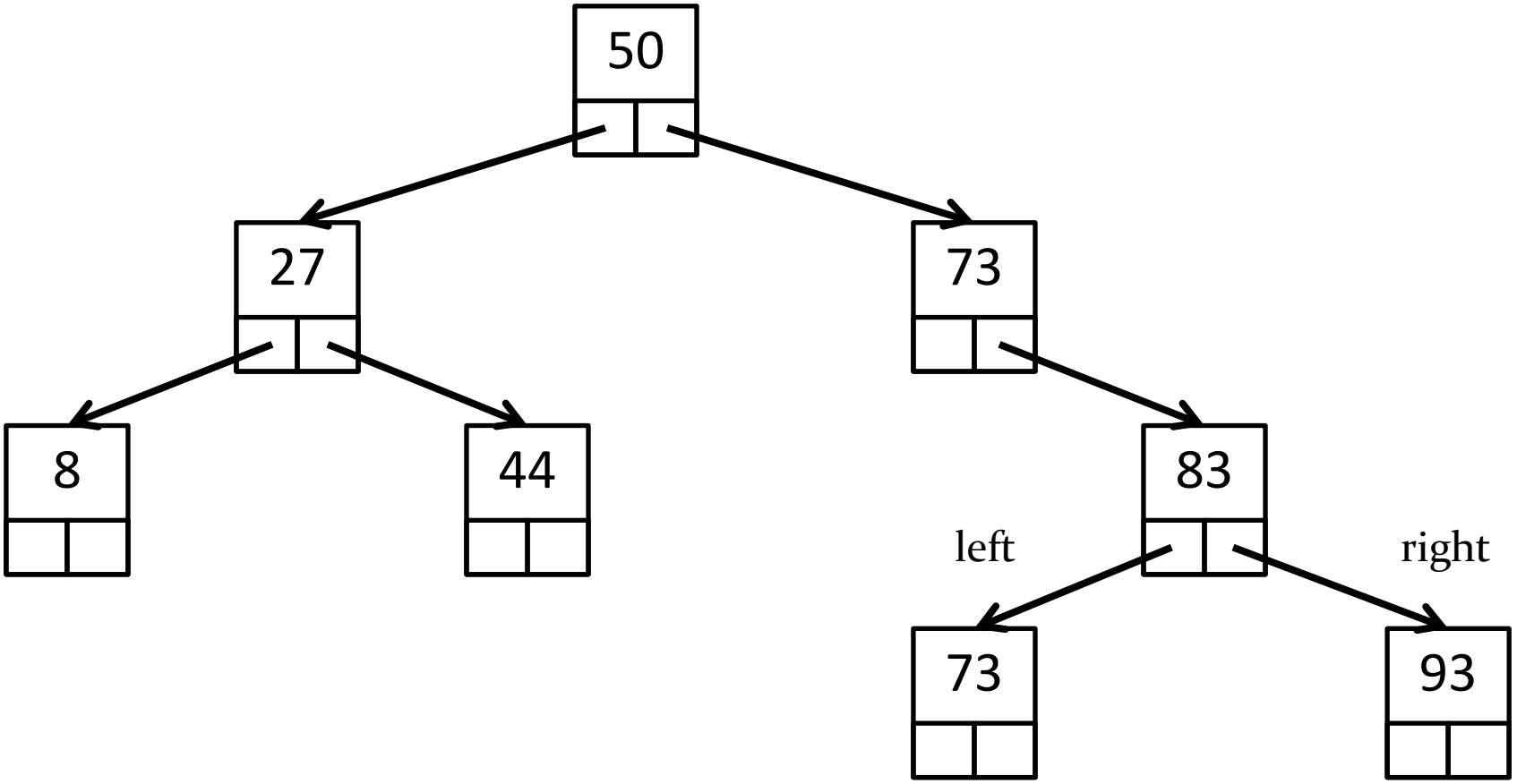
Binary Tree

- 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
73
93

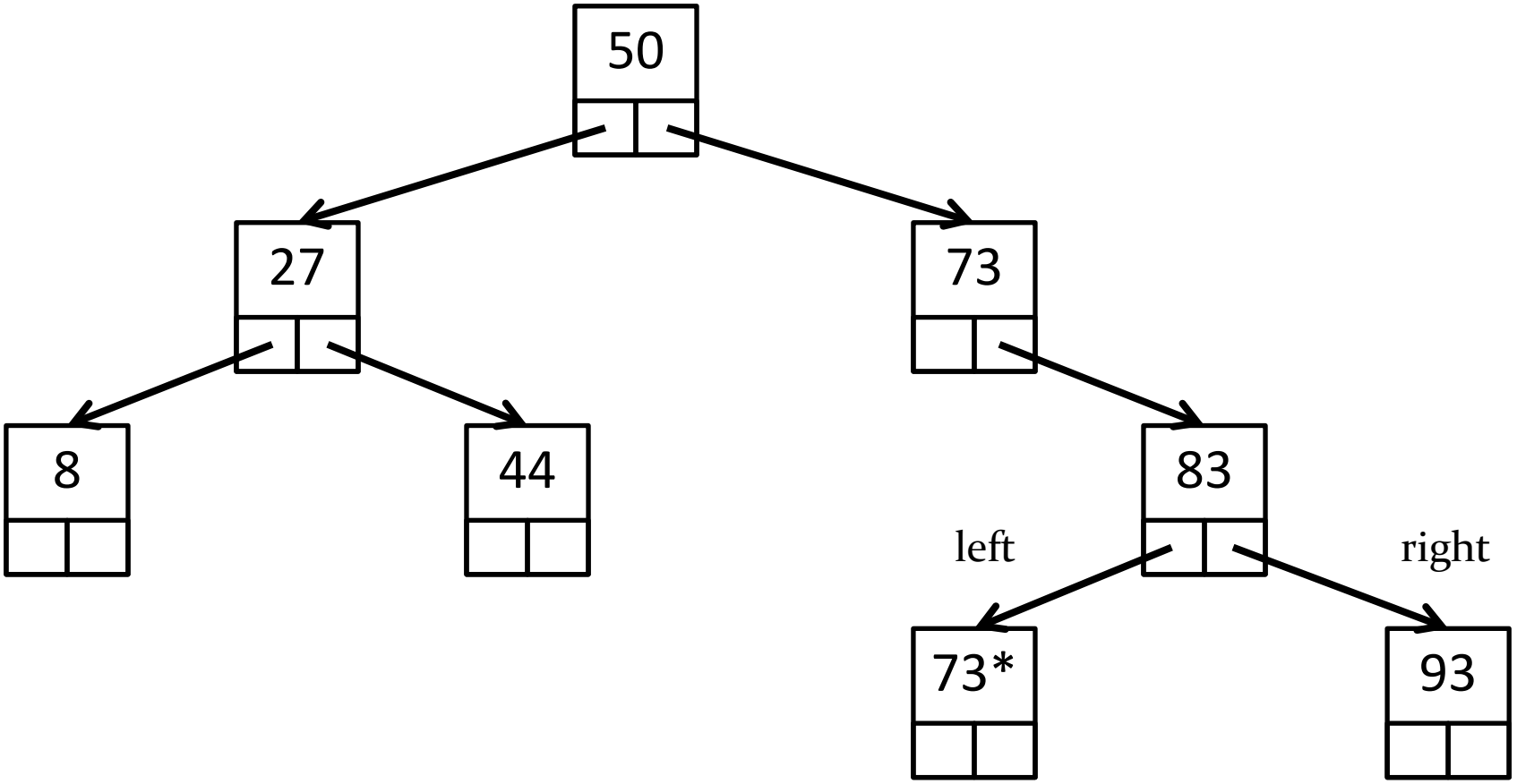
Binary Tree Algorithms

- 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;
}
```

Iteration

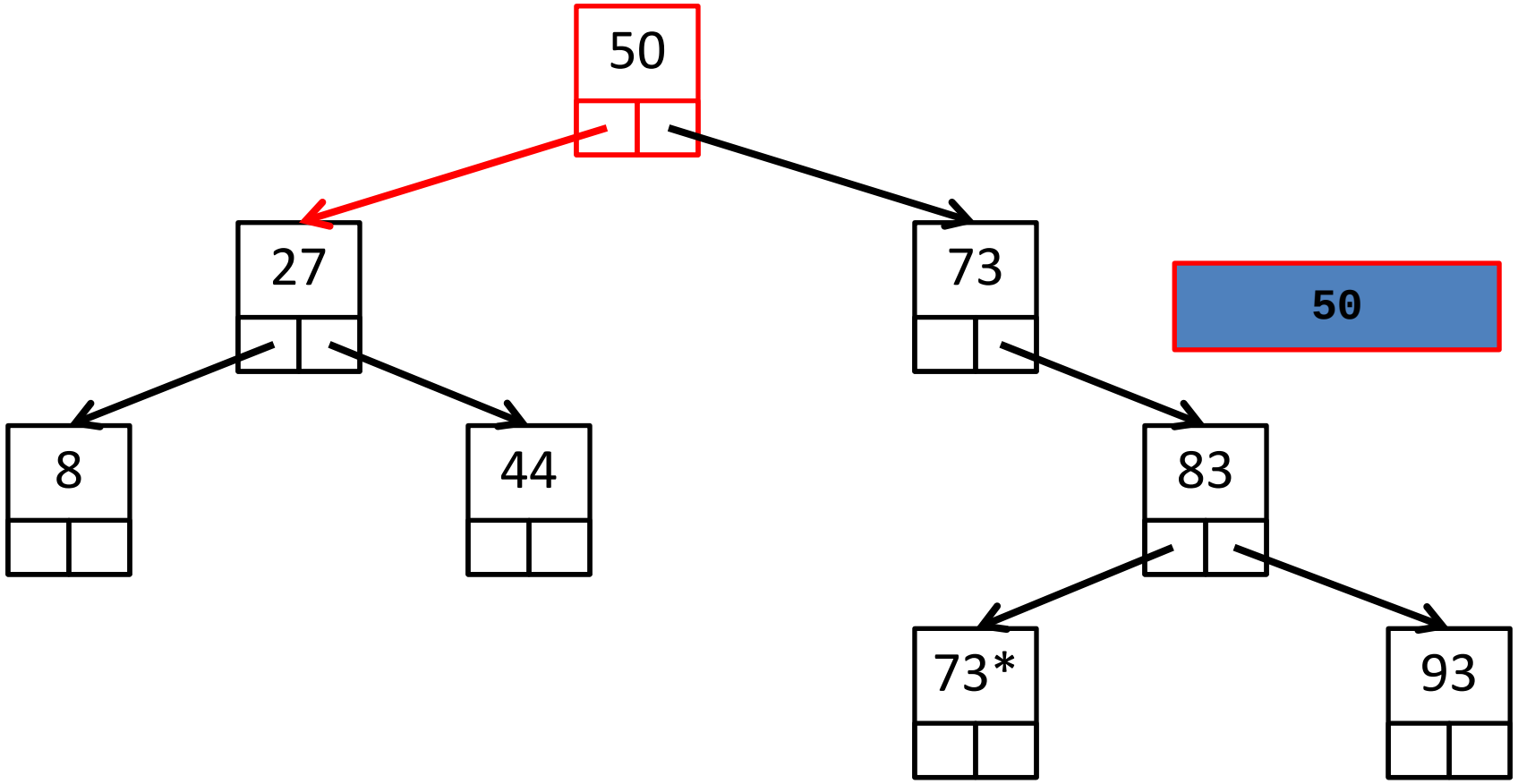
- Visiting every element of the tree can also be done recursively
- 3 possibilities based on when the root is visited
  - Inorder
    - Visit left child, then root, then right child
  - Preorder
    - Visit root, then left child, then right child
  - Postorder
    - Visit left child, then right child, then root



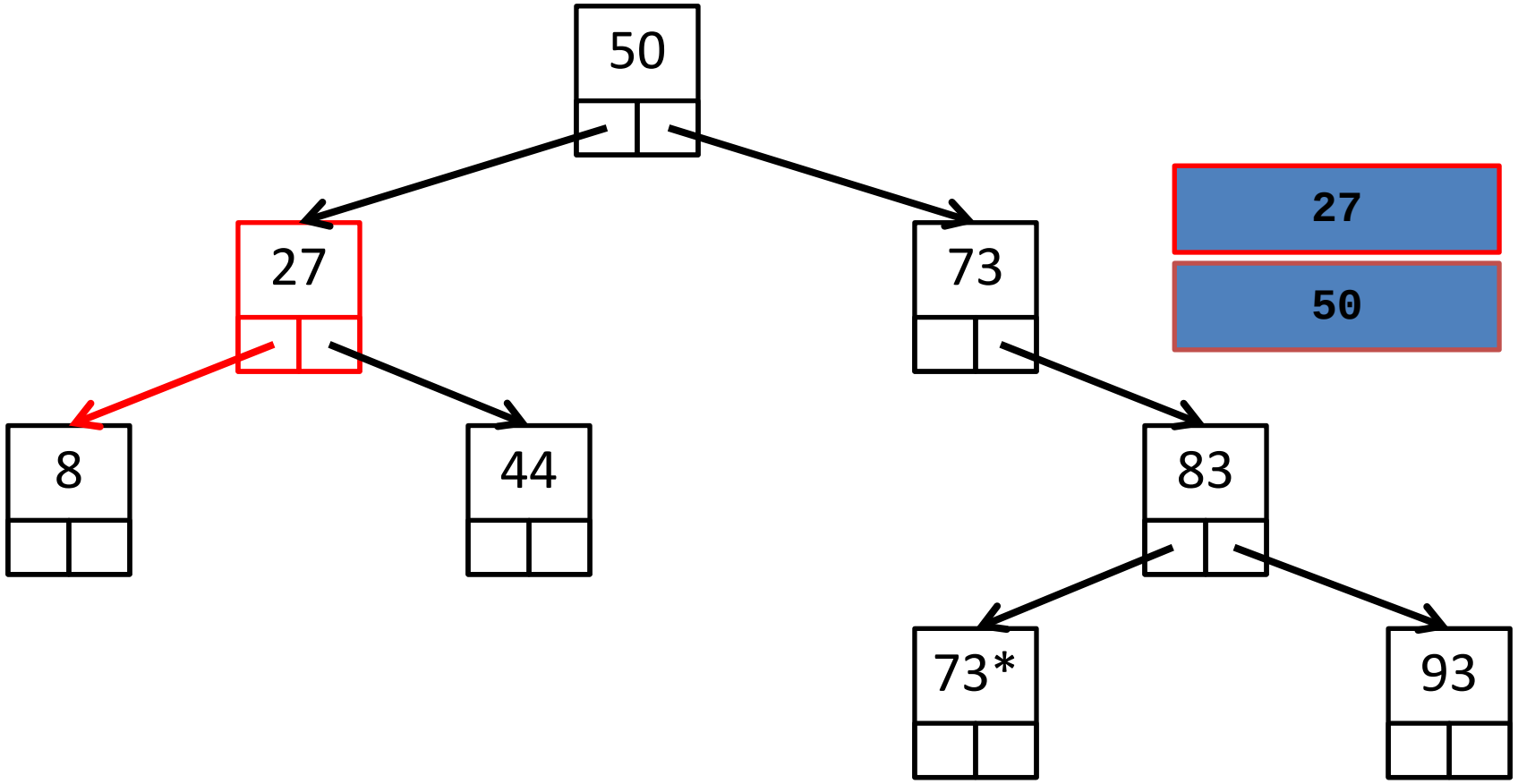
Inorder: 8, 27, 44, 50, 73, 73*, 83, 93

Example: Tree traversal

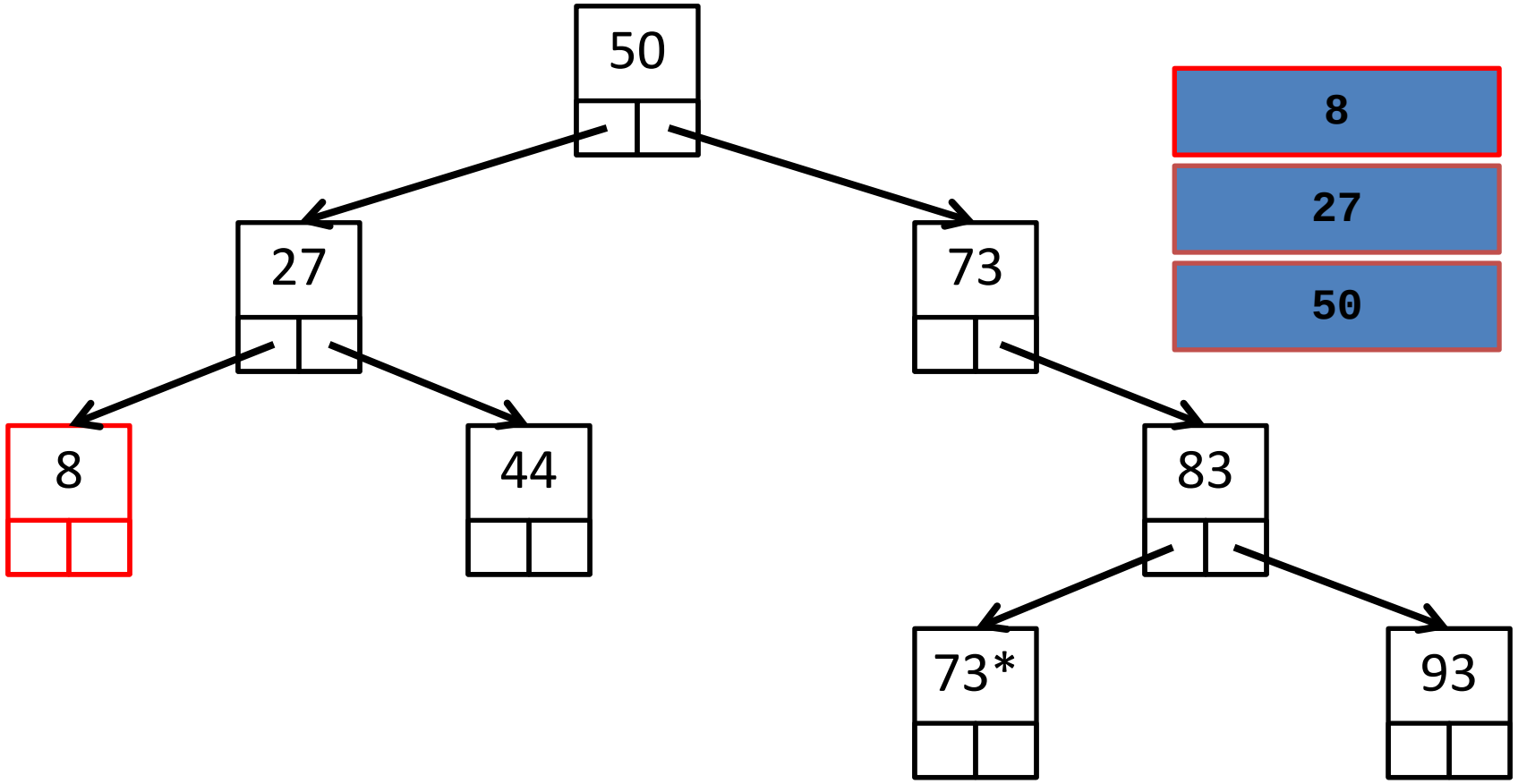
- A stack can be used in place of recursion for visiting all of the nodes of a tree
  - Basic idea is to push nodes onto the stack as you traverse the tree
  - Pushing the node onto the stack allows you to remember that you have to visit the other branch of the tree rooted at the node



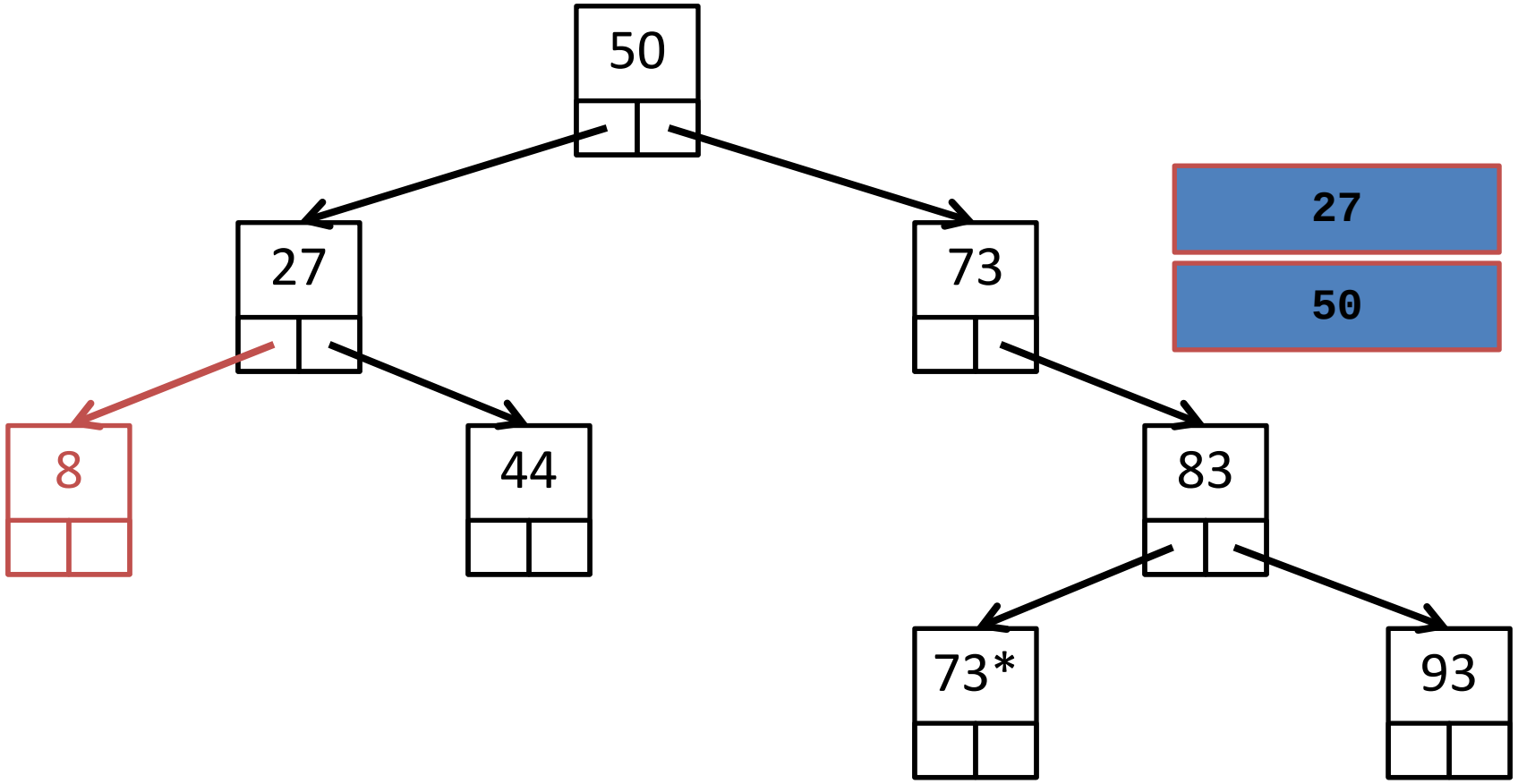
Inorder: 8, 27, 44, 50, 73, 73*, 83, 93



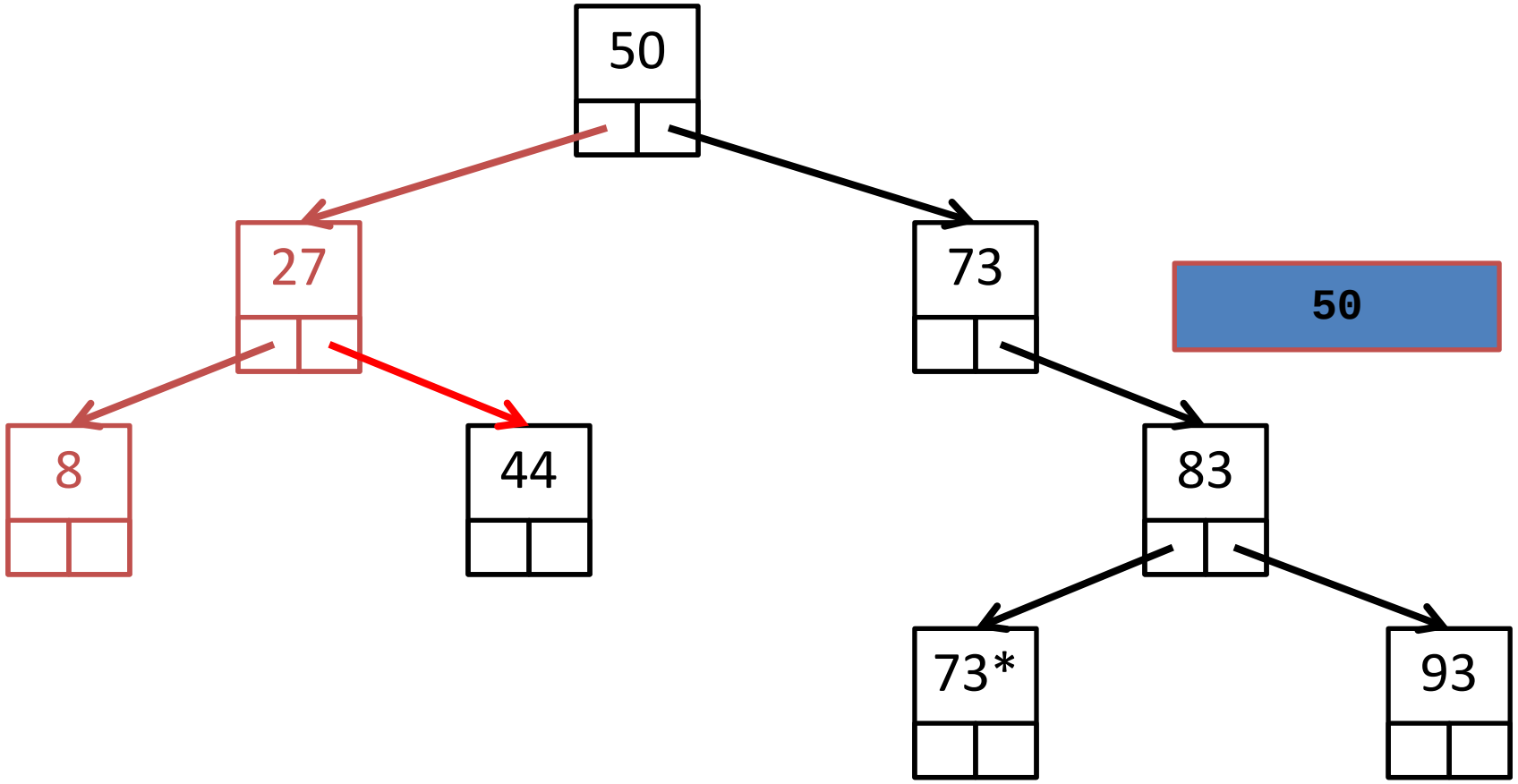
Inorder: 8, 27, 44, 50, 73, 73*, 83, 93



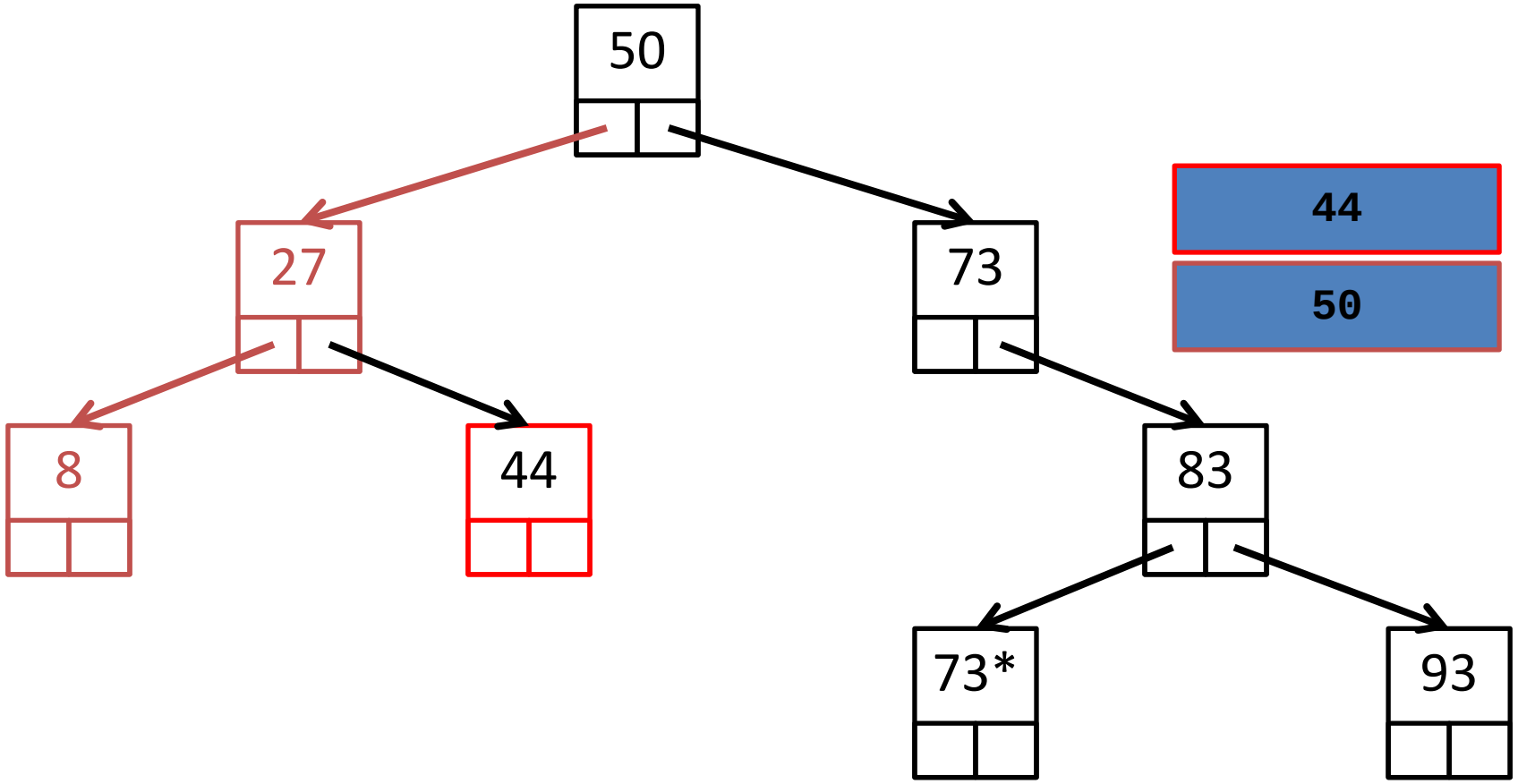
Inorder: 8, 27, 44, 50, 73, 73*, 83, 93



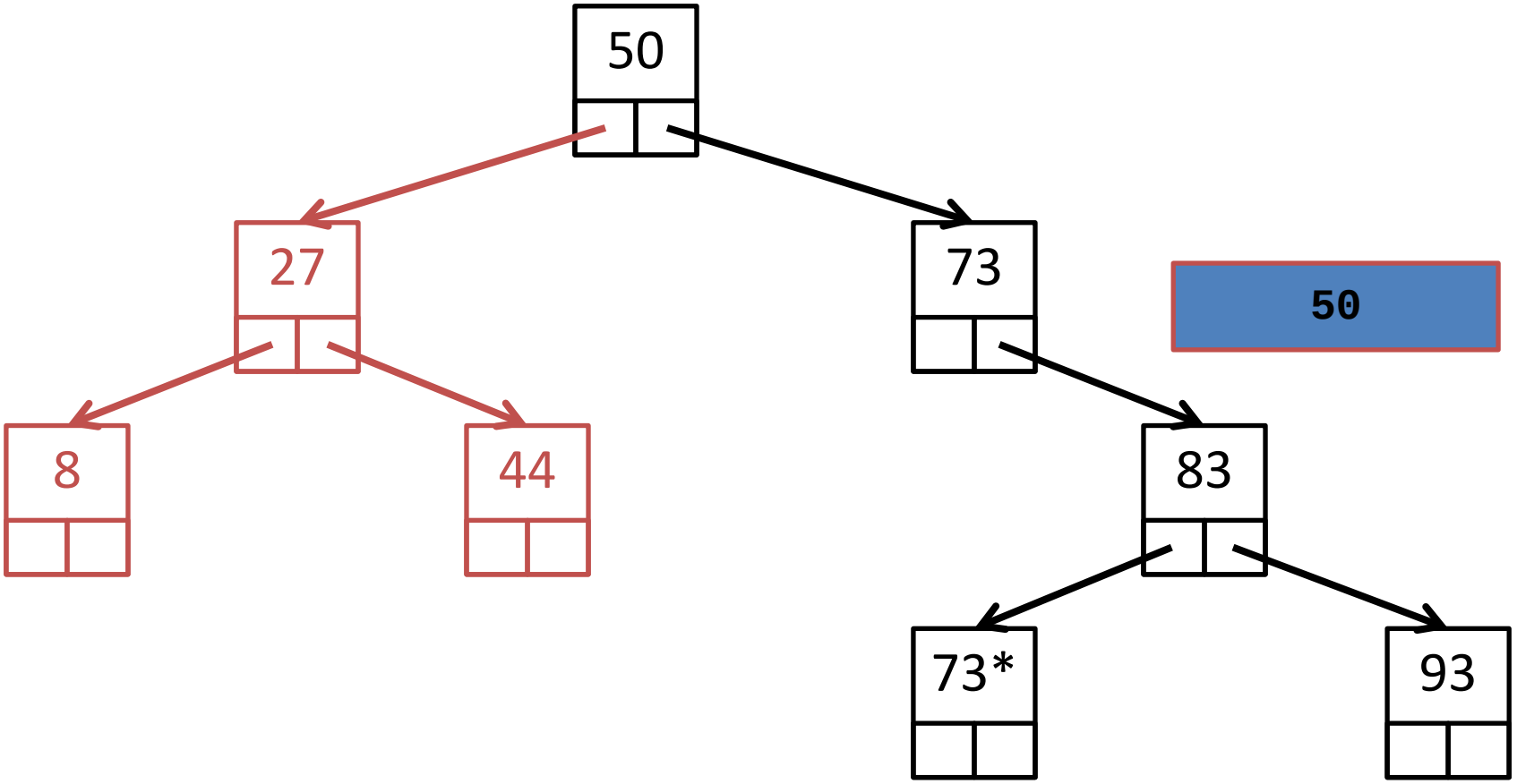
Inorder: 8, 27, 44, 50, 73, 73*, 83, 93



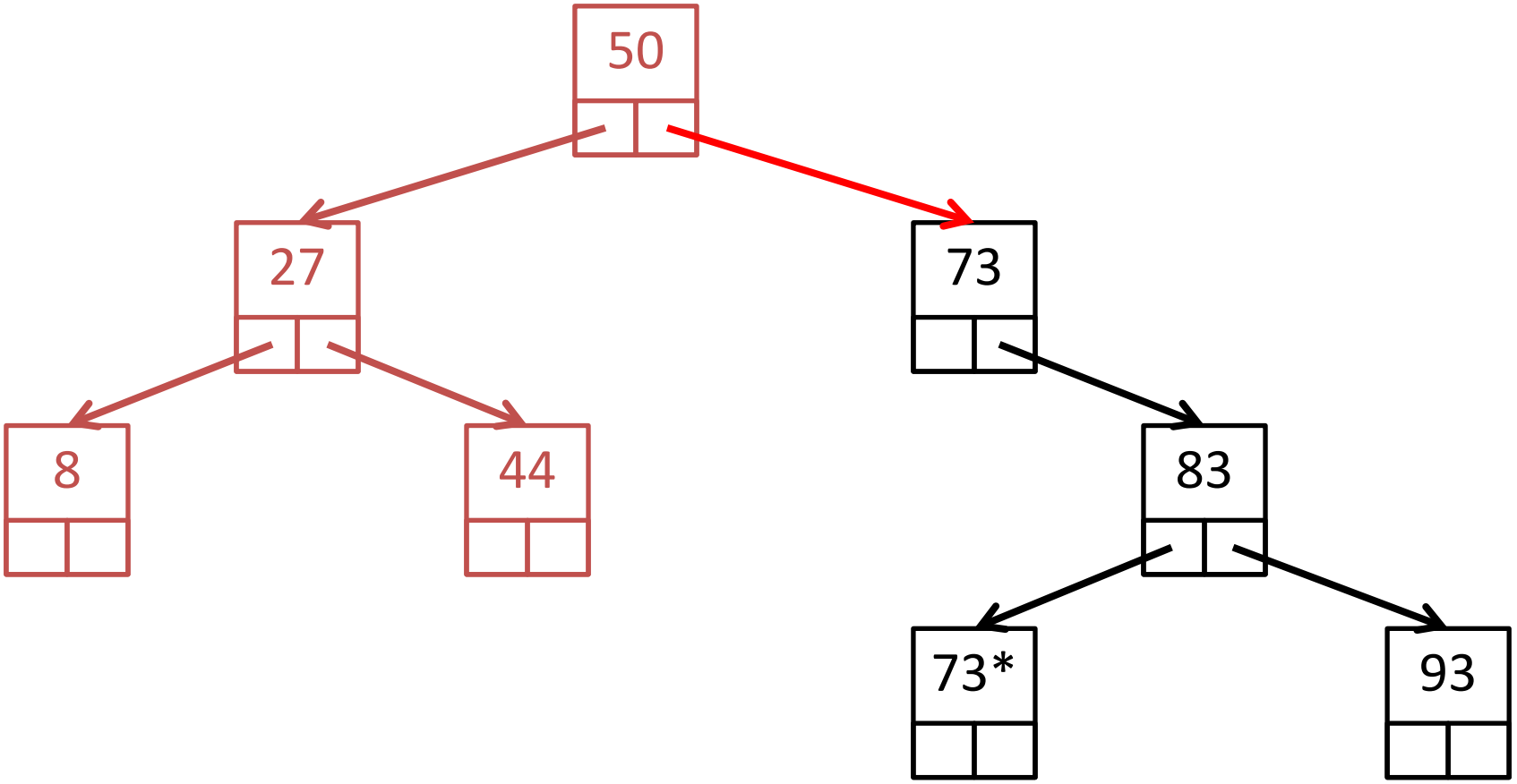
Inorder: 8, 27, 44, 50, 73, 73*, 83, 93



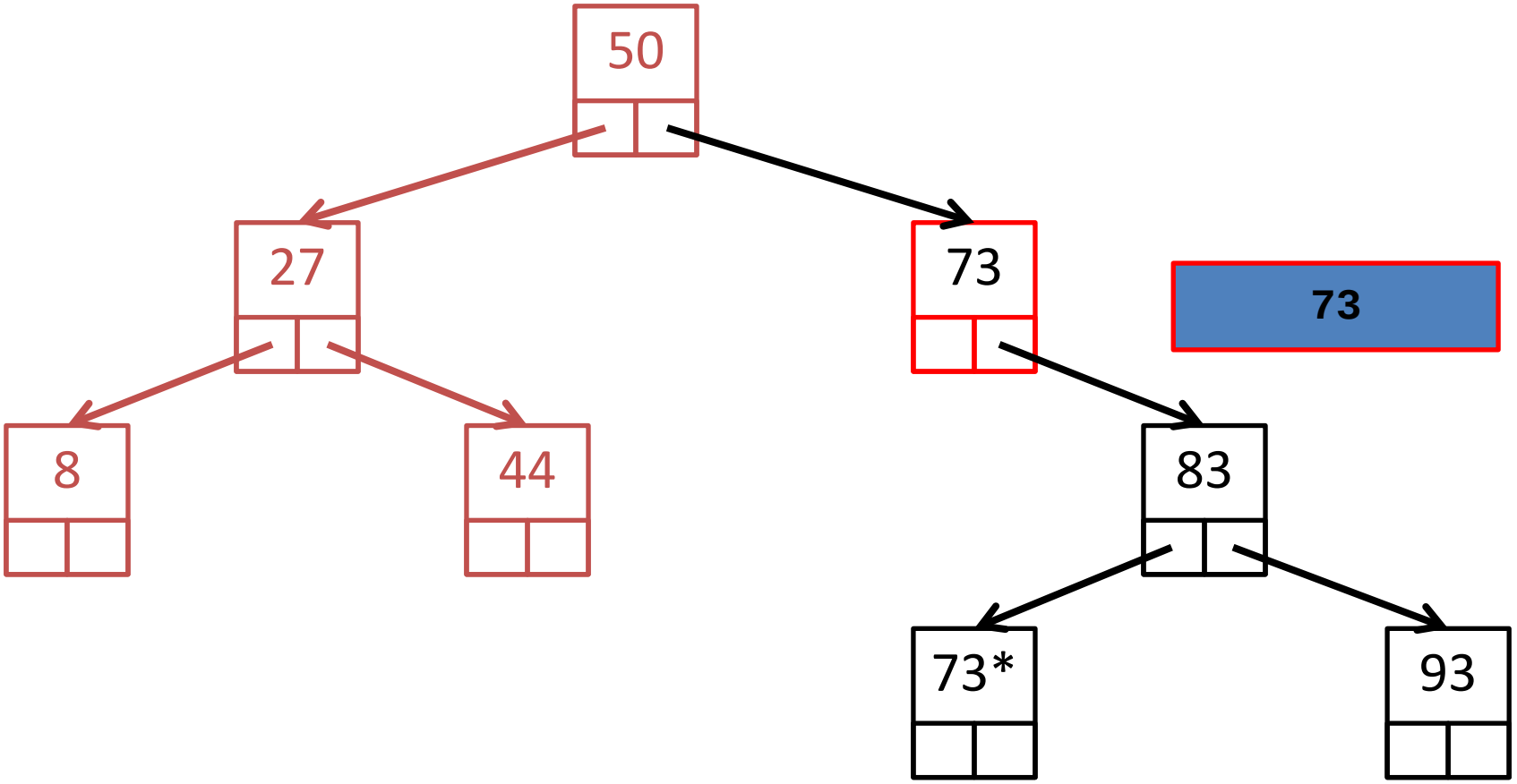
Inorder: 8, 27, 44, 50, 73, 73*, 83, 93



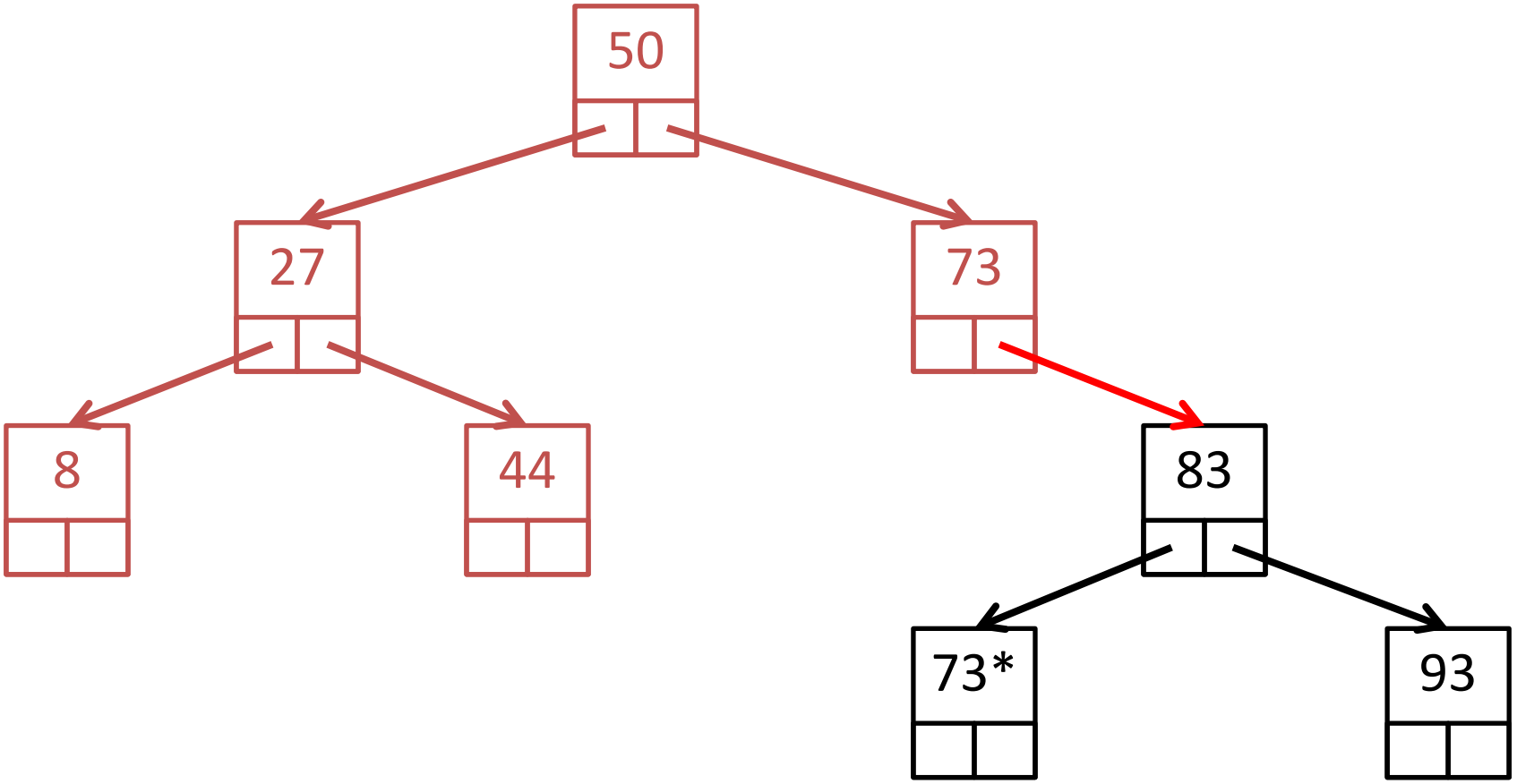
Inorder: 8, 27, 44, 50, 73, 73*, 83, 93



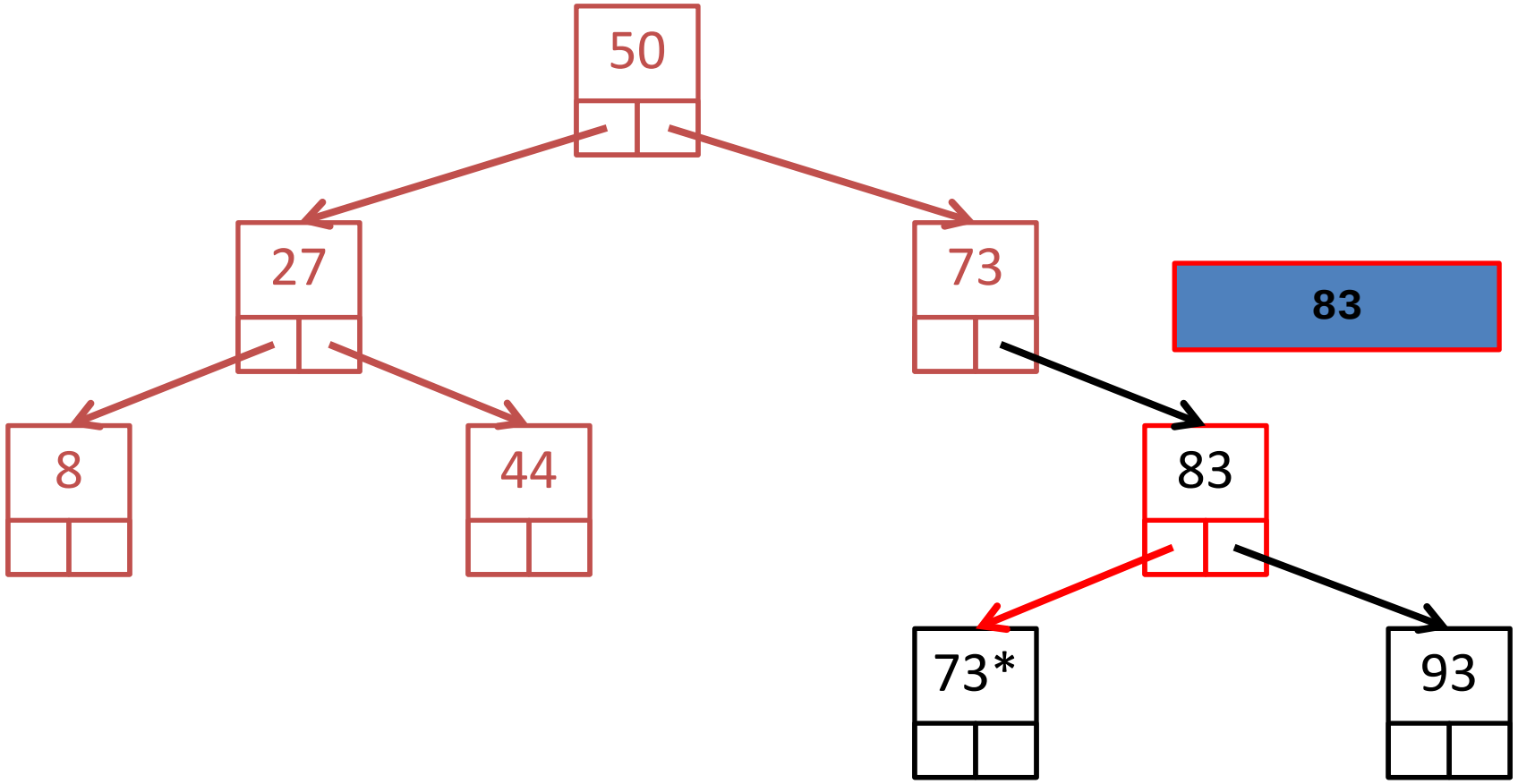
Inorder: 8, 27, 44, 50, 73, 73*, 83, 93



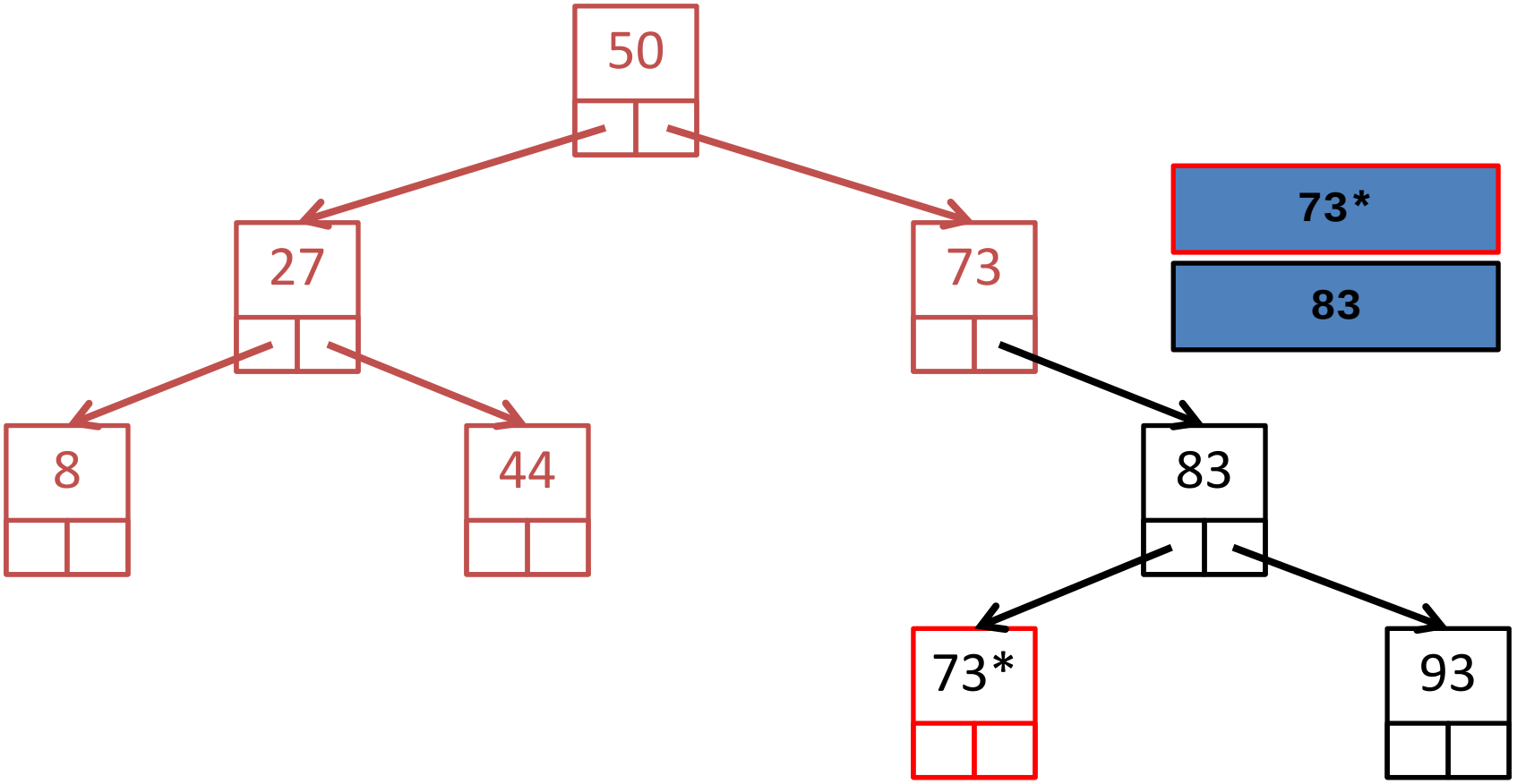
Inorder: 8, 27, 44, 50, 73, 73*, 83, 93



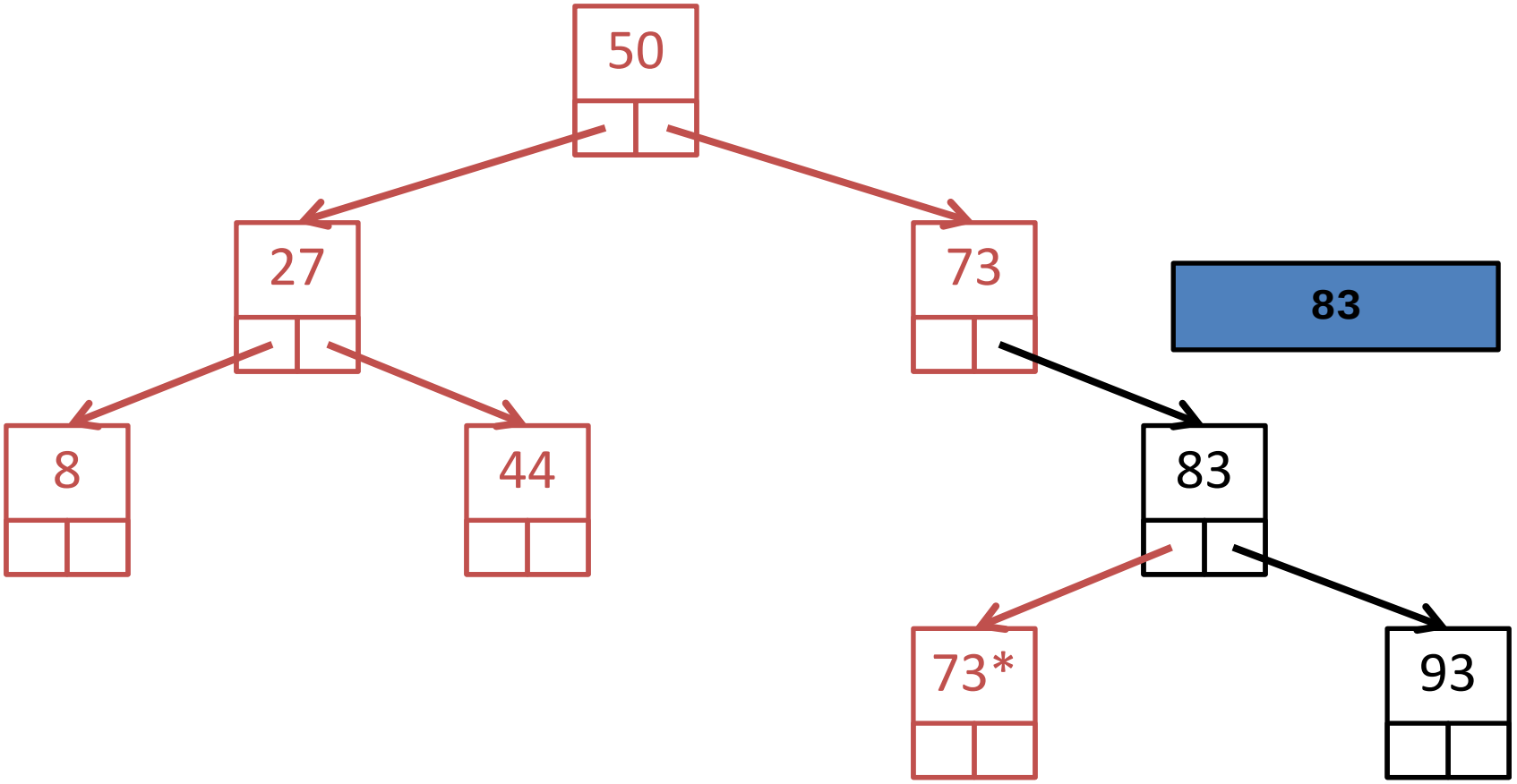
Inorder: 8, 27, 44, 50, 73, 73*, 83, 93



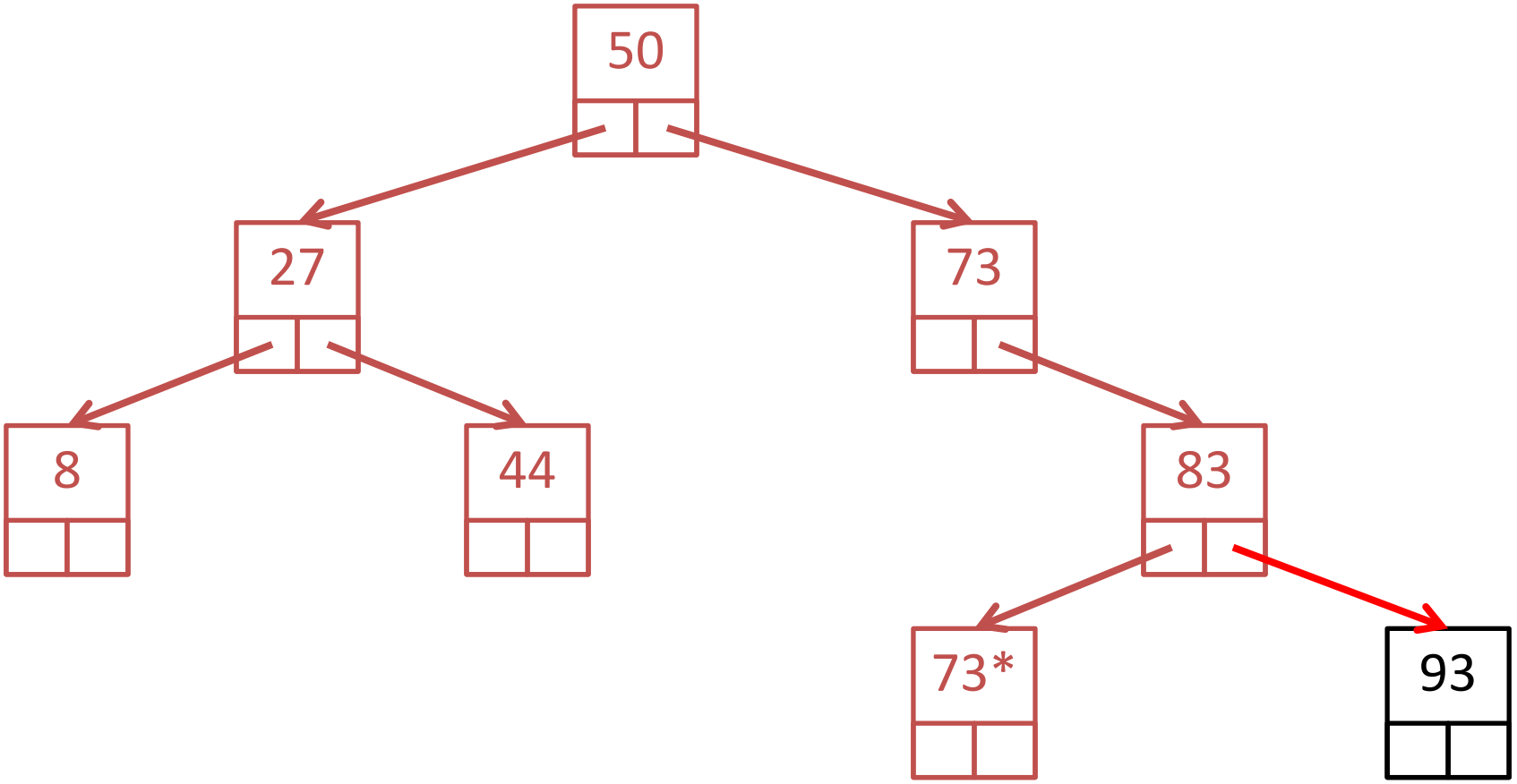
Inorder: 8, 27, 44, 50, 73, 73*, 83, 93



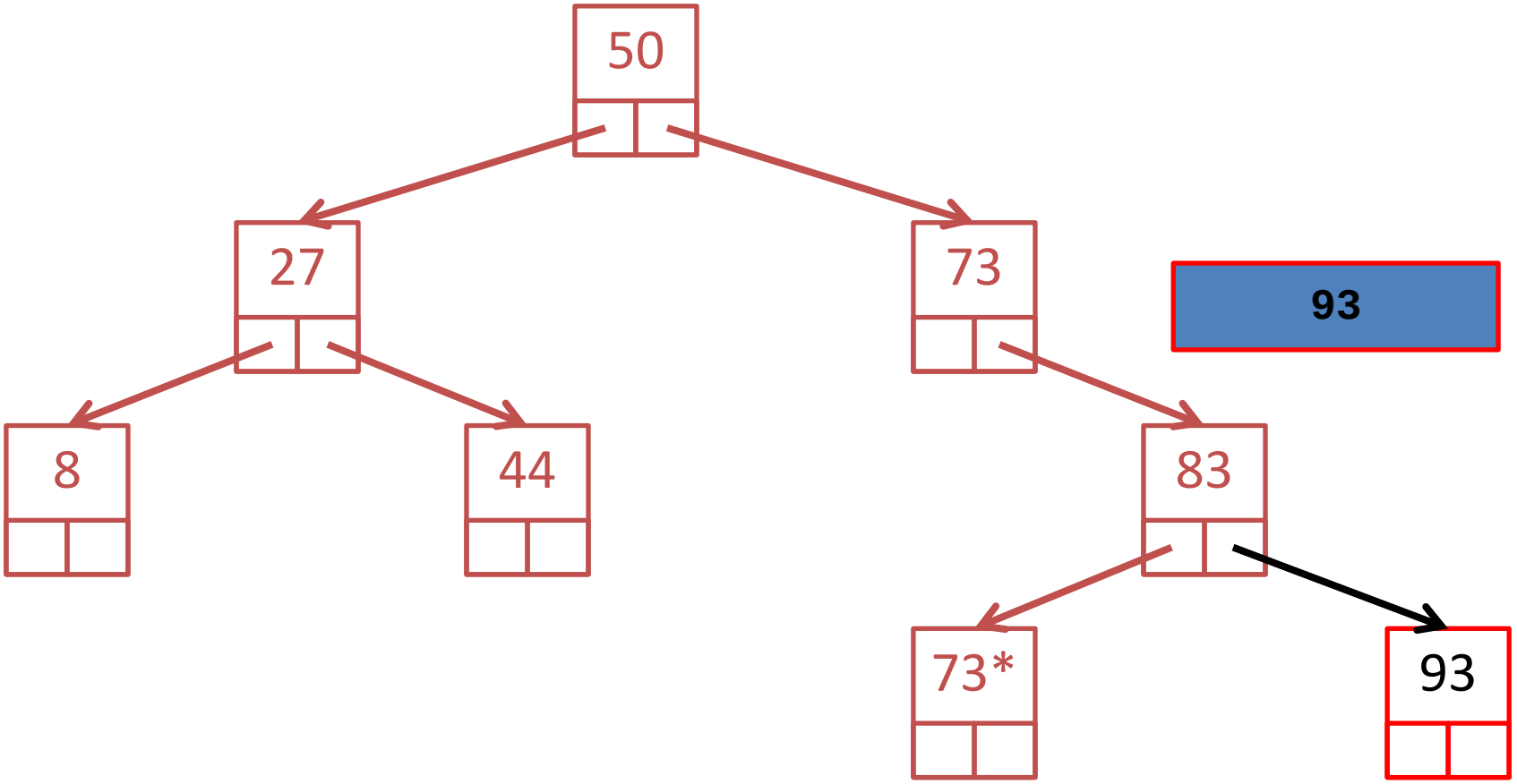
Inorder: 8, 27, 44, 50, 73, 73*, 83, 93



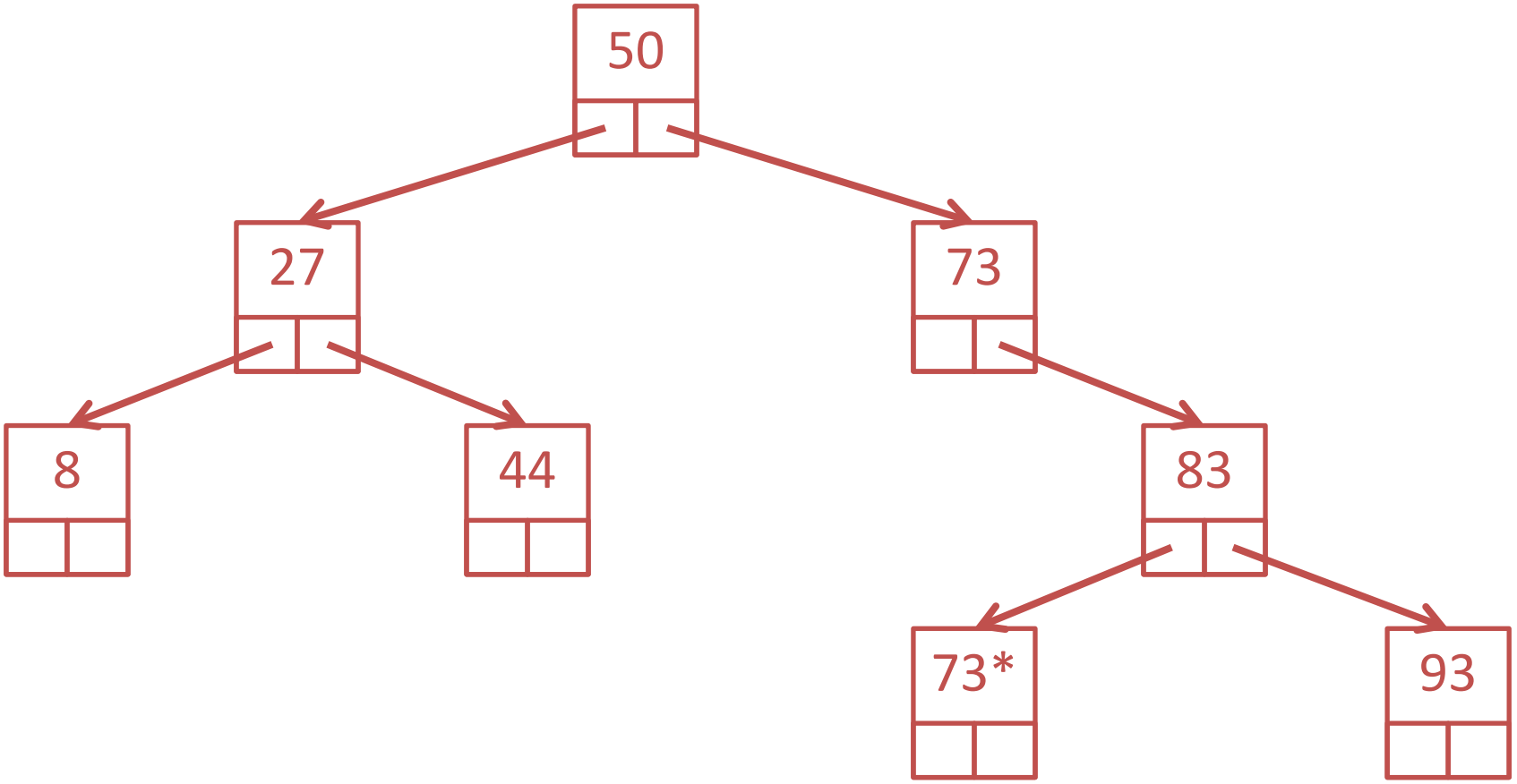
Inorder: 8, 27, 44, 50, 73, 73*, 83, 93



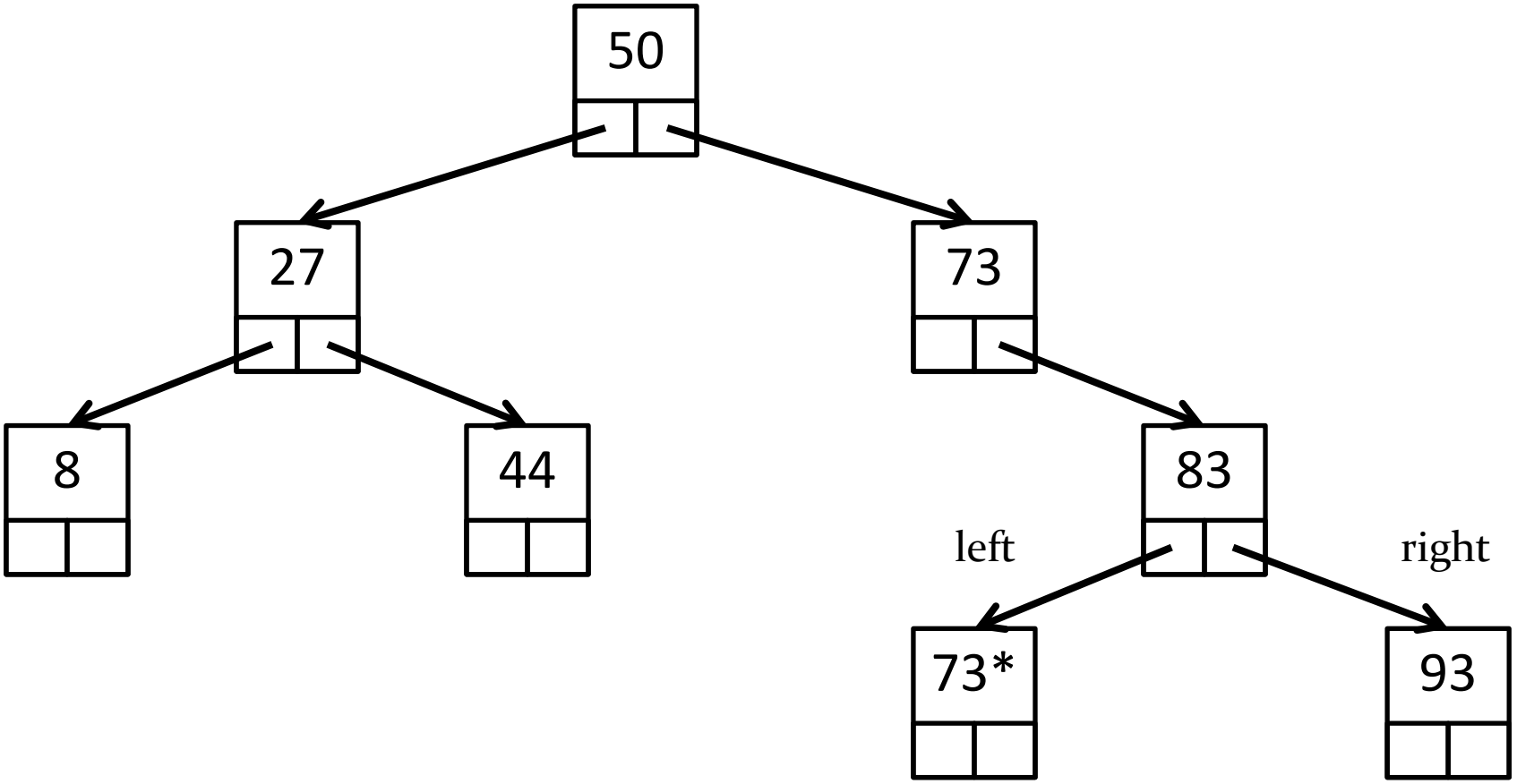
Inorder: 8, 27, 44, 50, 73, 73*, 83, 93



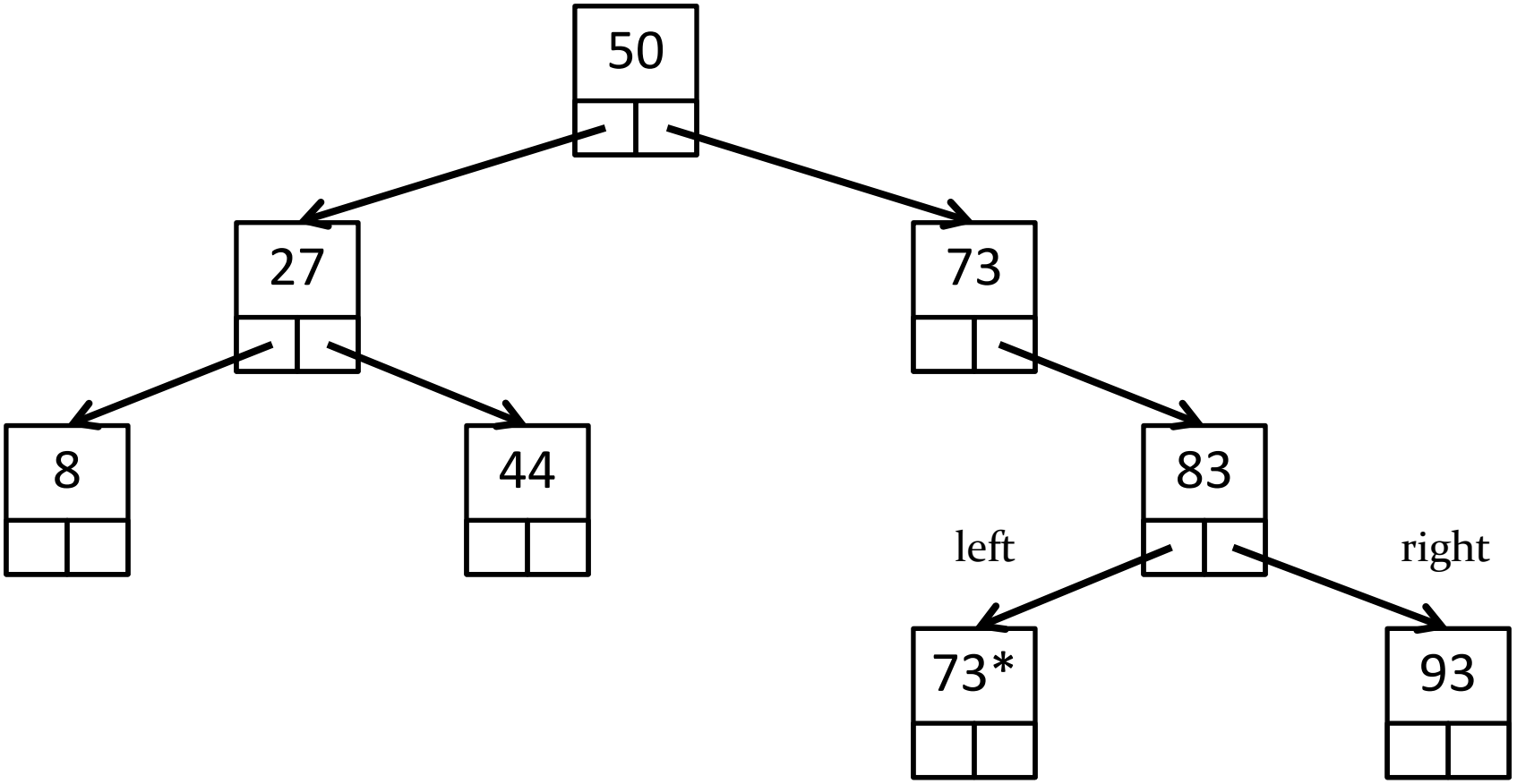
Inorder: 8, 27, 44, 50, 73, 73*, 83, 93



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



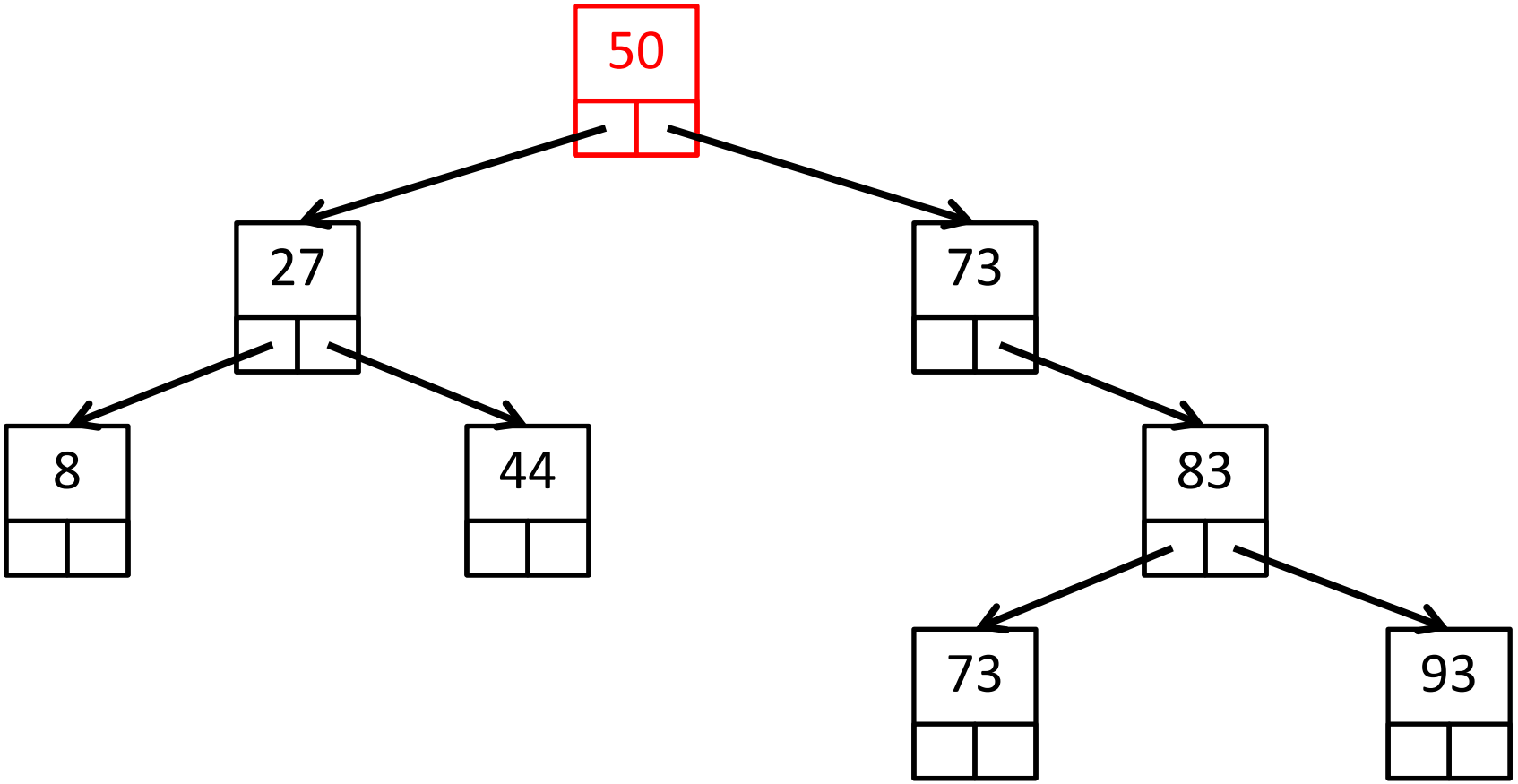
Preorder: 50, 27, 8, 44, 73, 83, 73*, 93



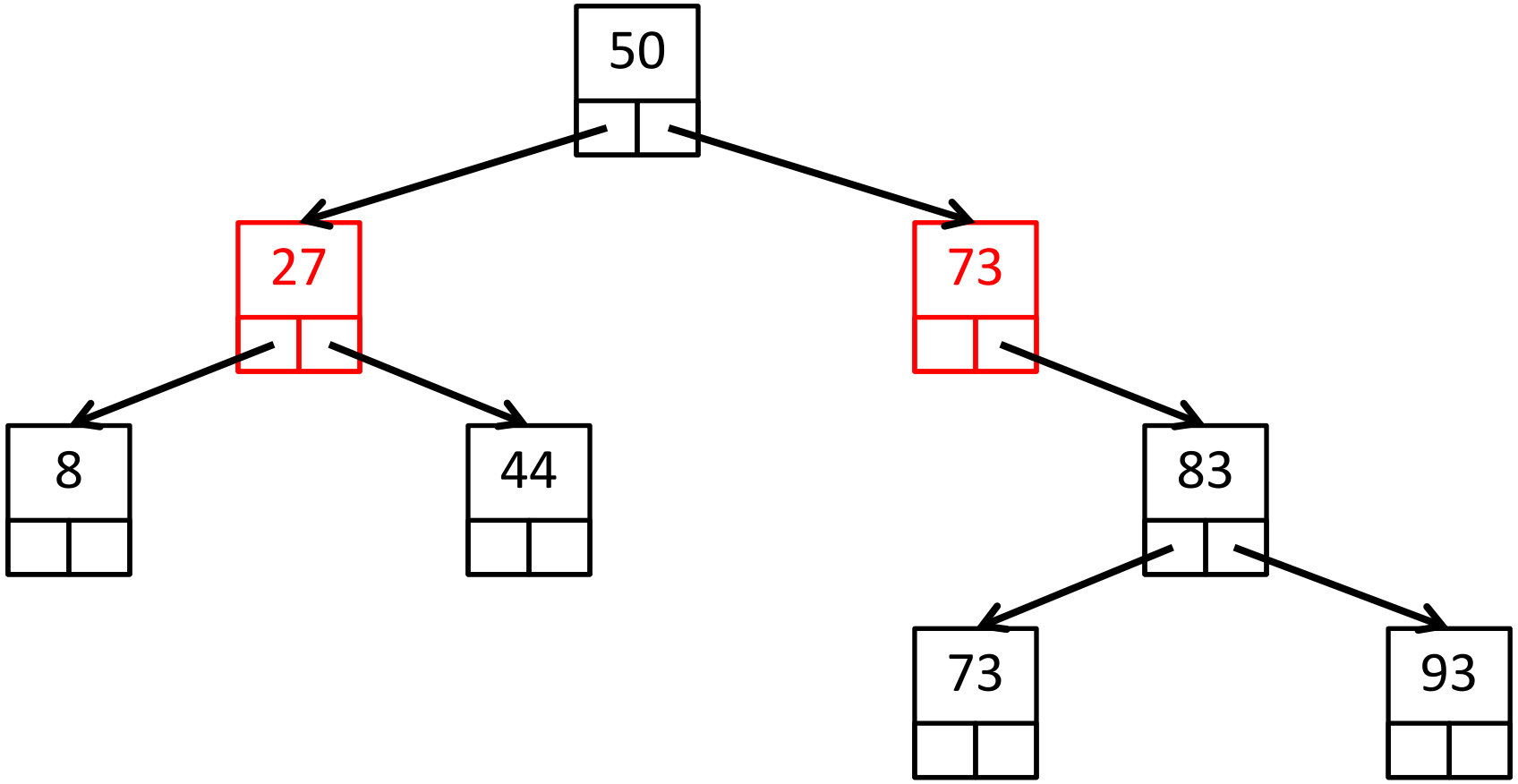
Postorder: 8, 44, 27, 73*, 93, 83, 73, 50

Breadth-first search

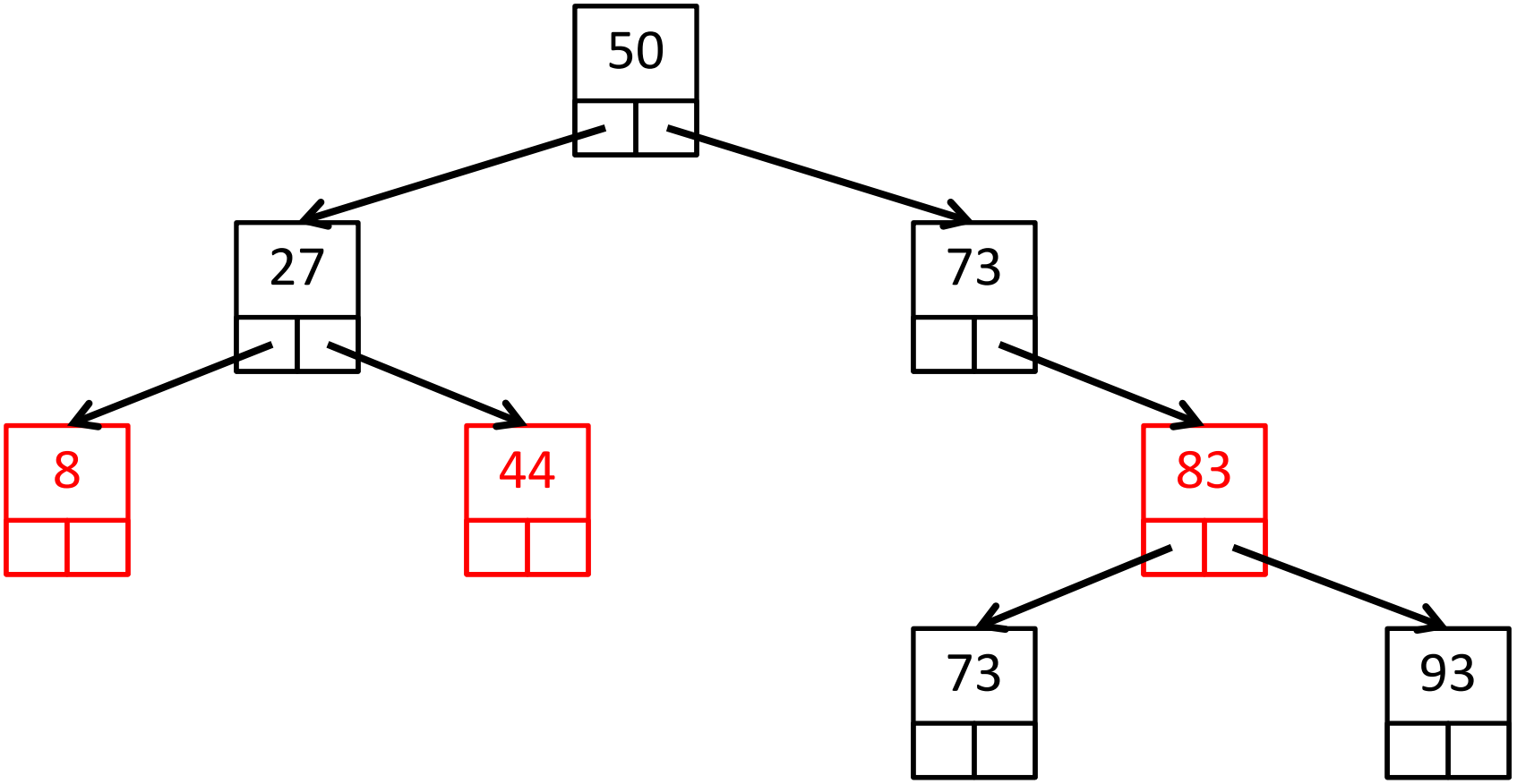
- Visiting every node of a tree using breadth-first search results in visiting nodes in order of their level in the tree



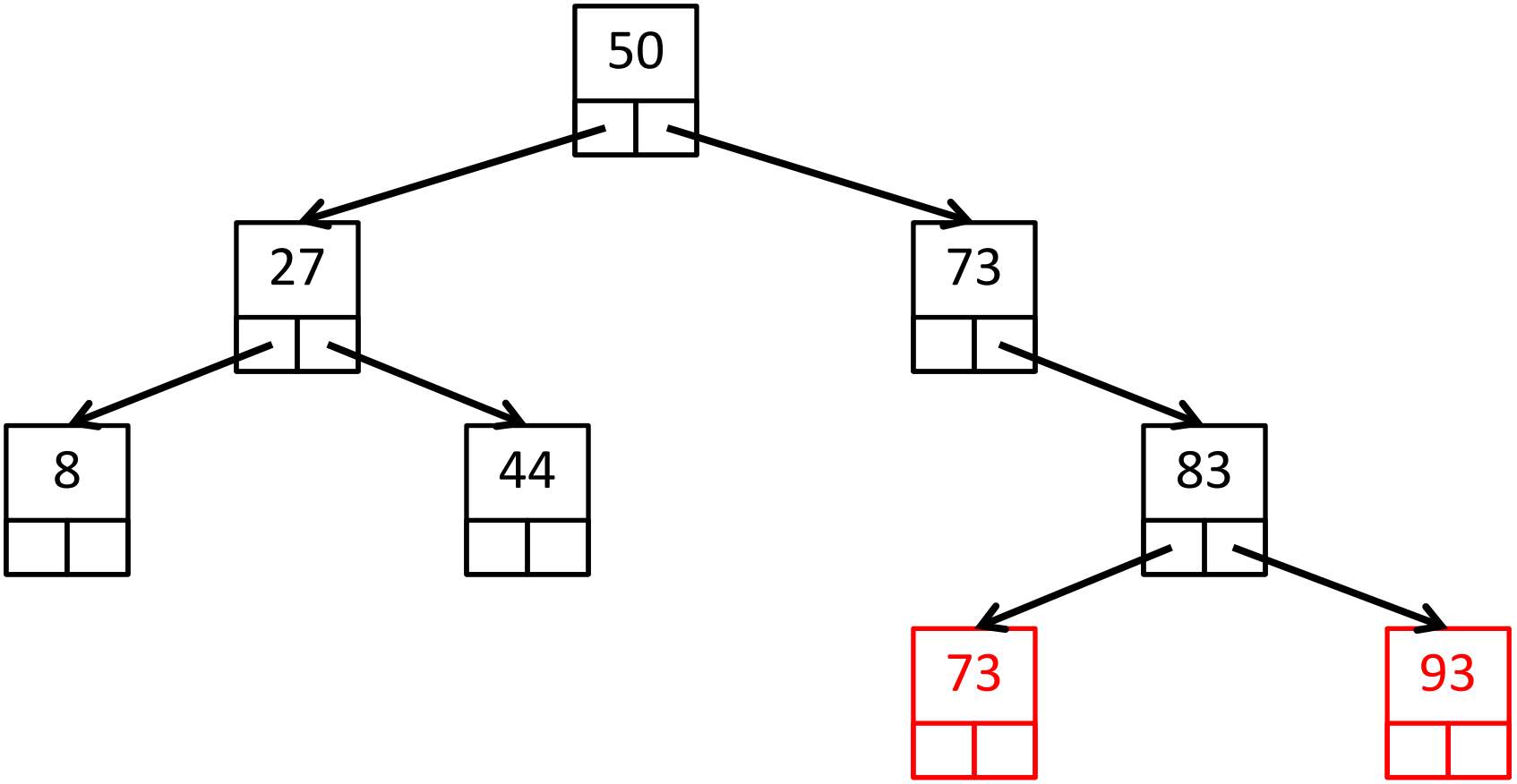
BFS: 50



BFS: 50, 27, 73



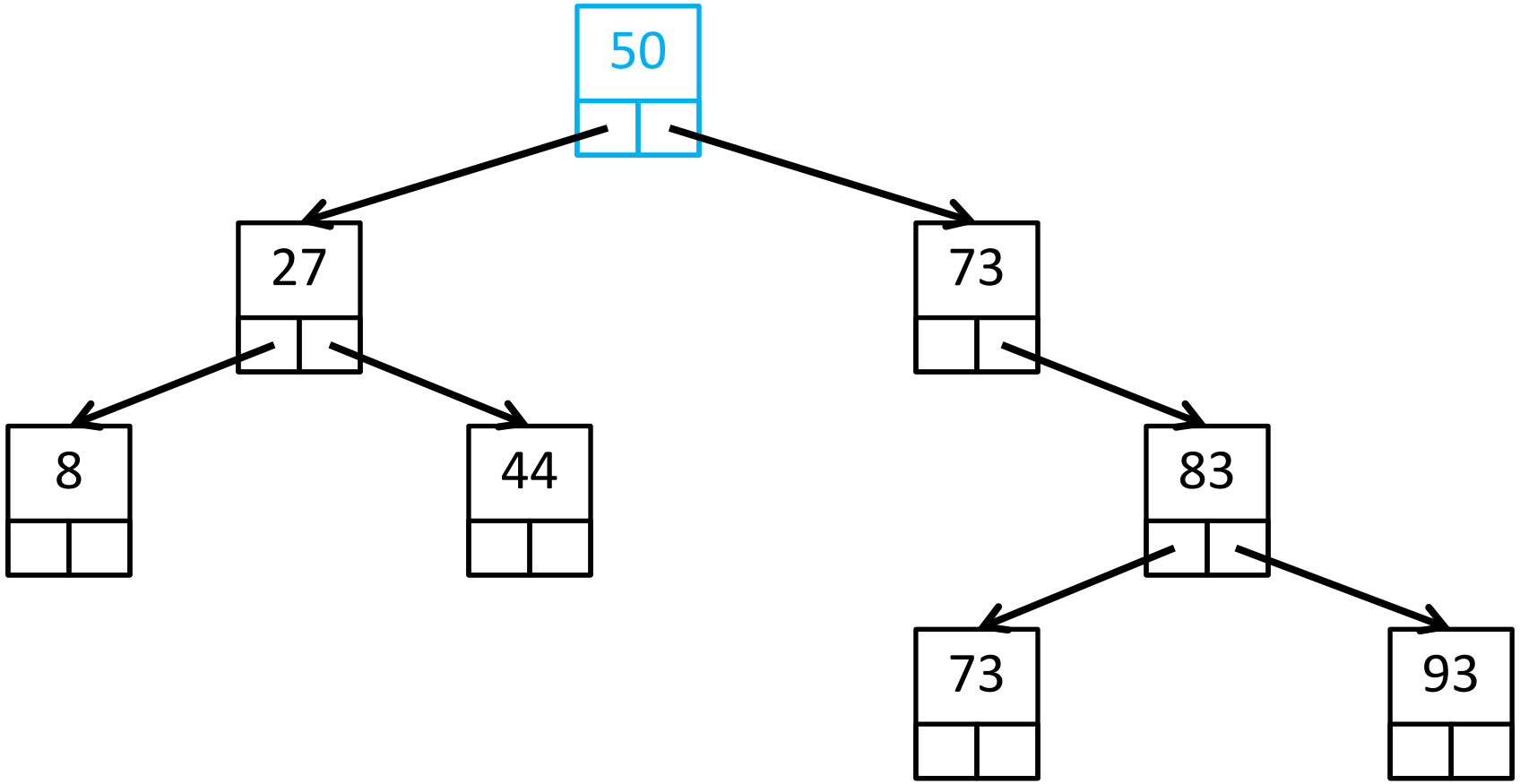
BFS: 50, 27, 73, 8, 44, 83



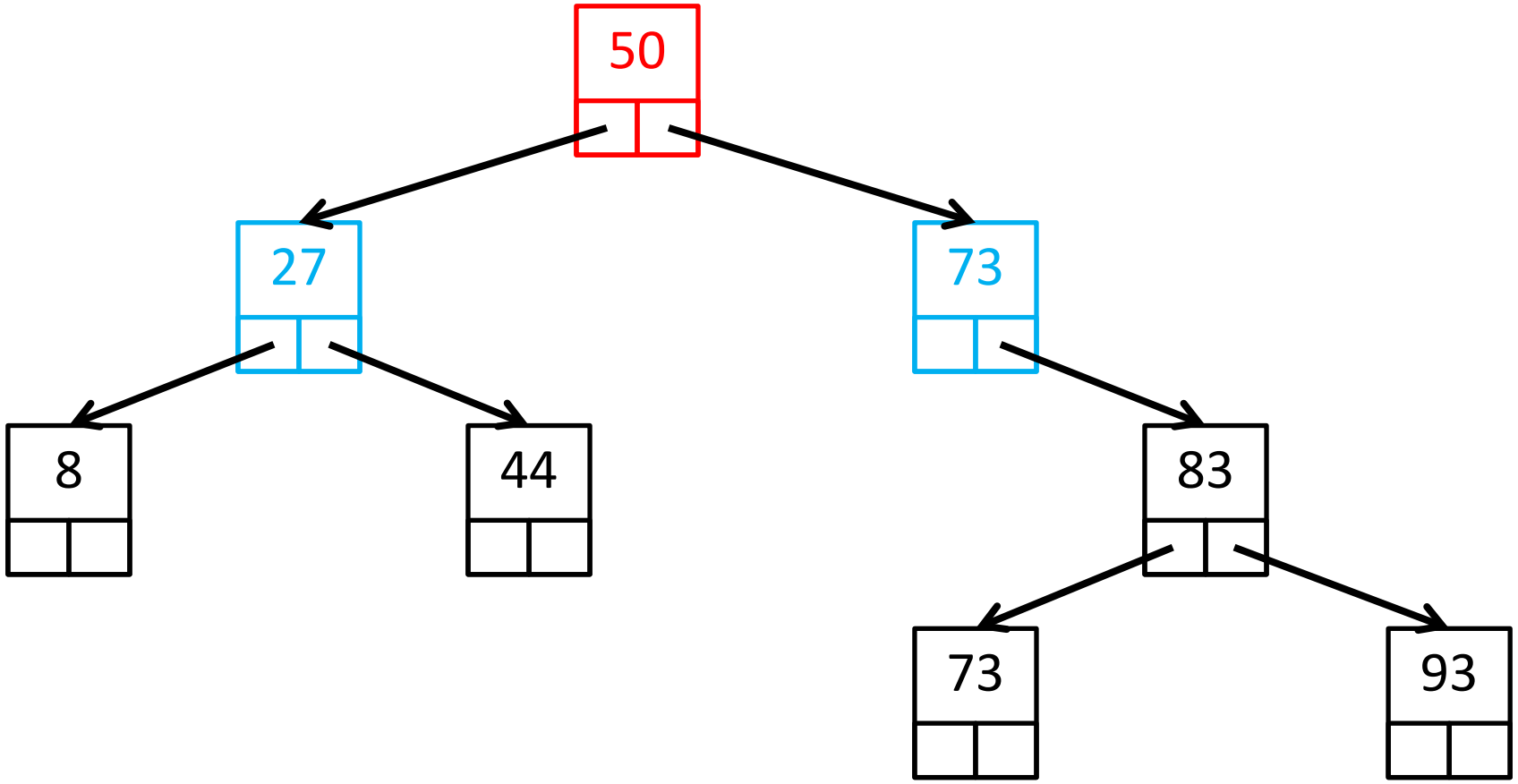
BFS: 50, 27, 73, 8, 44, 83, 73, 93

Breadth-first search algorithm

```
Q.enqueue(root node)
while Q is not empty {
  n = Q.dequeue()
  if n.left != null {
    Q.enqueue(n.left)
  }
  if n.right != null {
    Q.enqueue(n.right)
  }
}
```

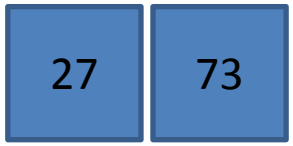


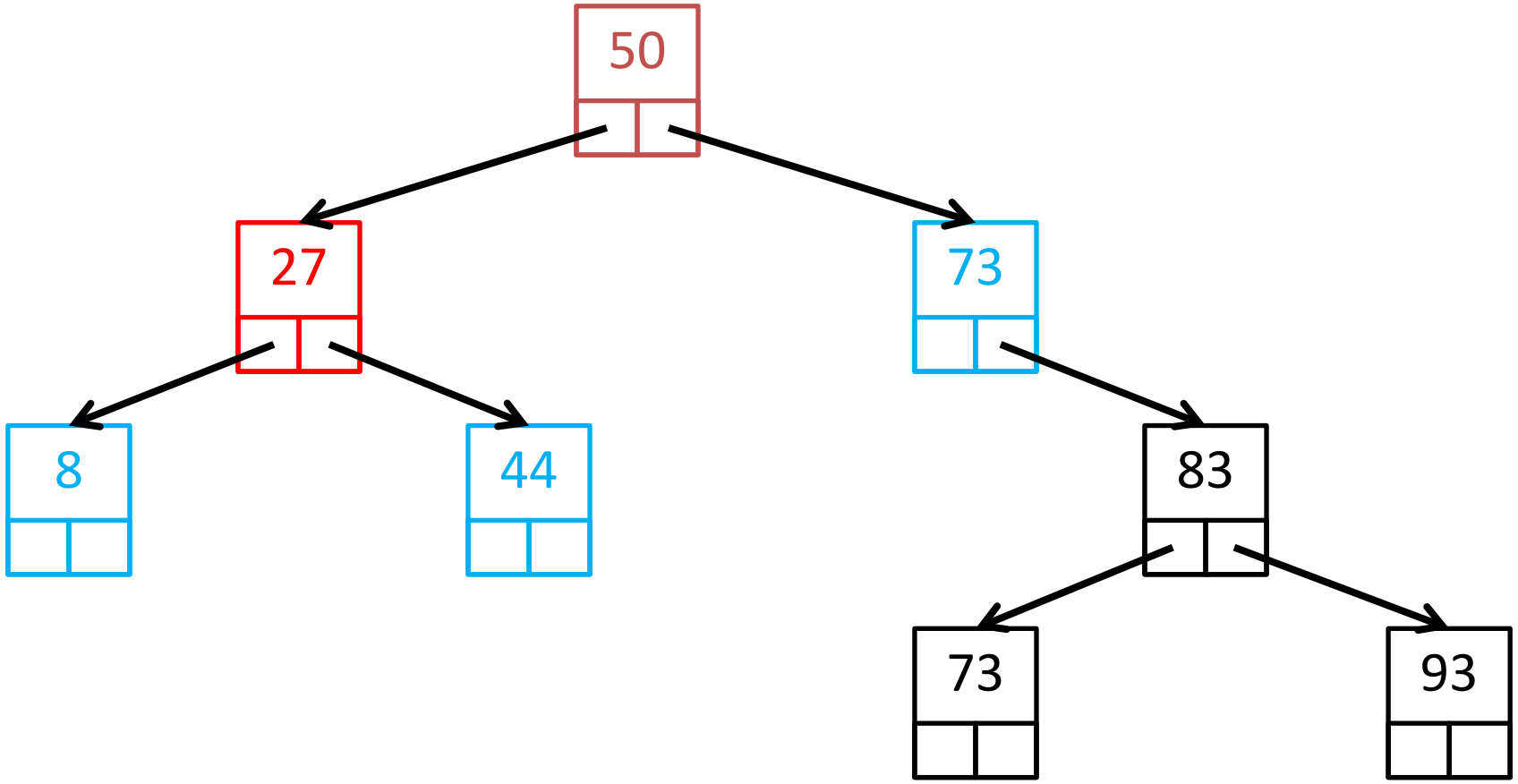
BFS:



BFS: 50

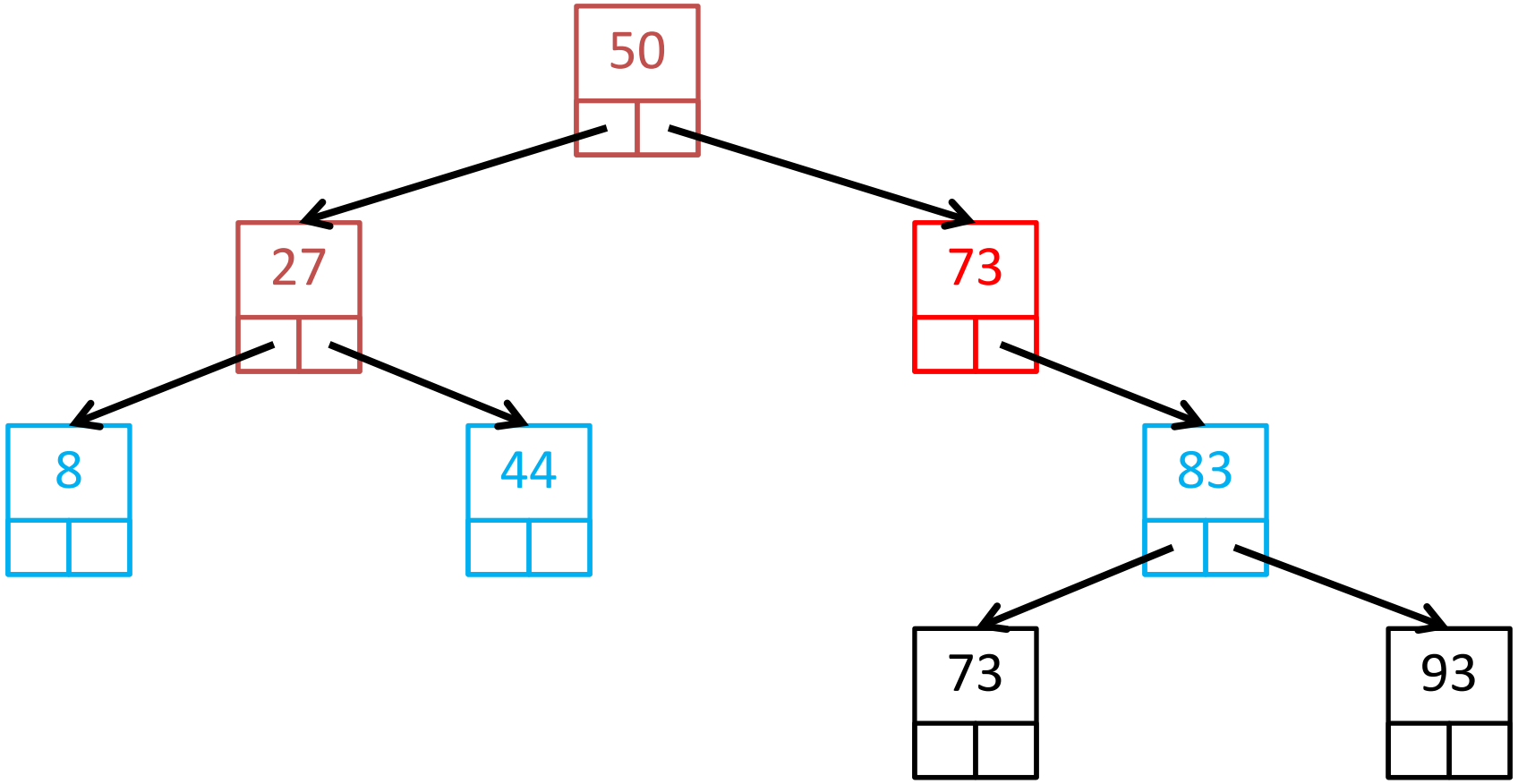
dequeue 50,
enqueue left and right





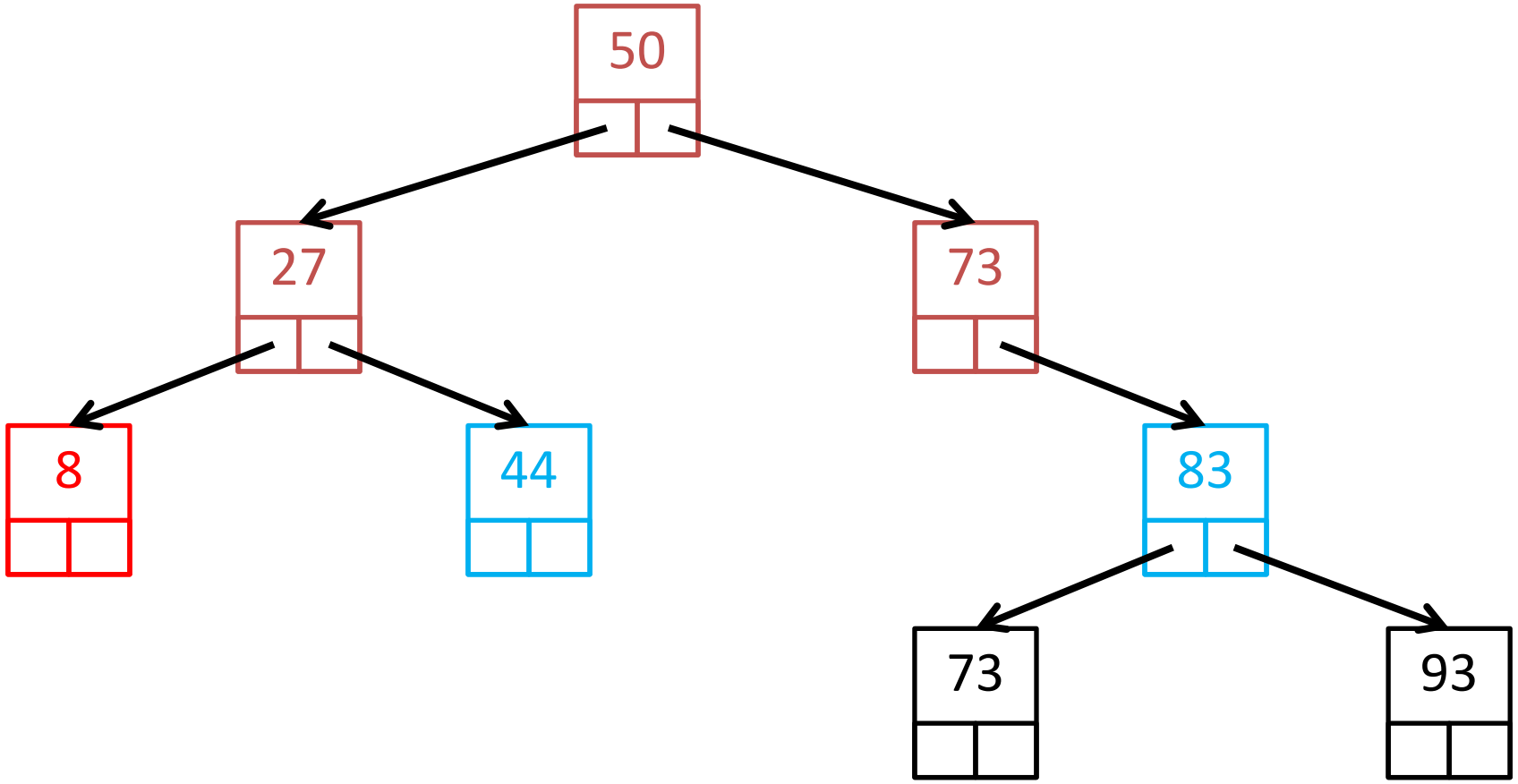
BFS: 50, 27

dequeue 27,
enqueue left and right



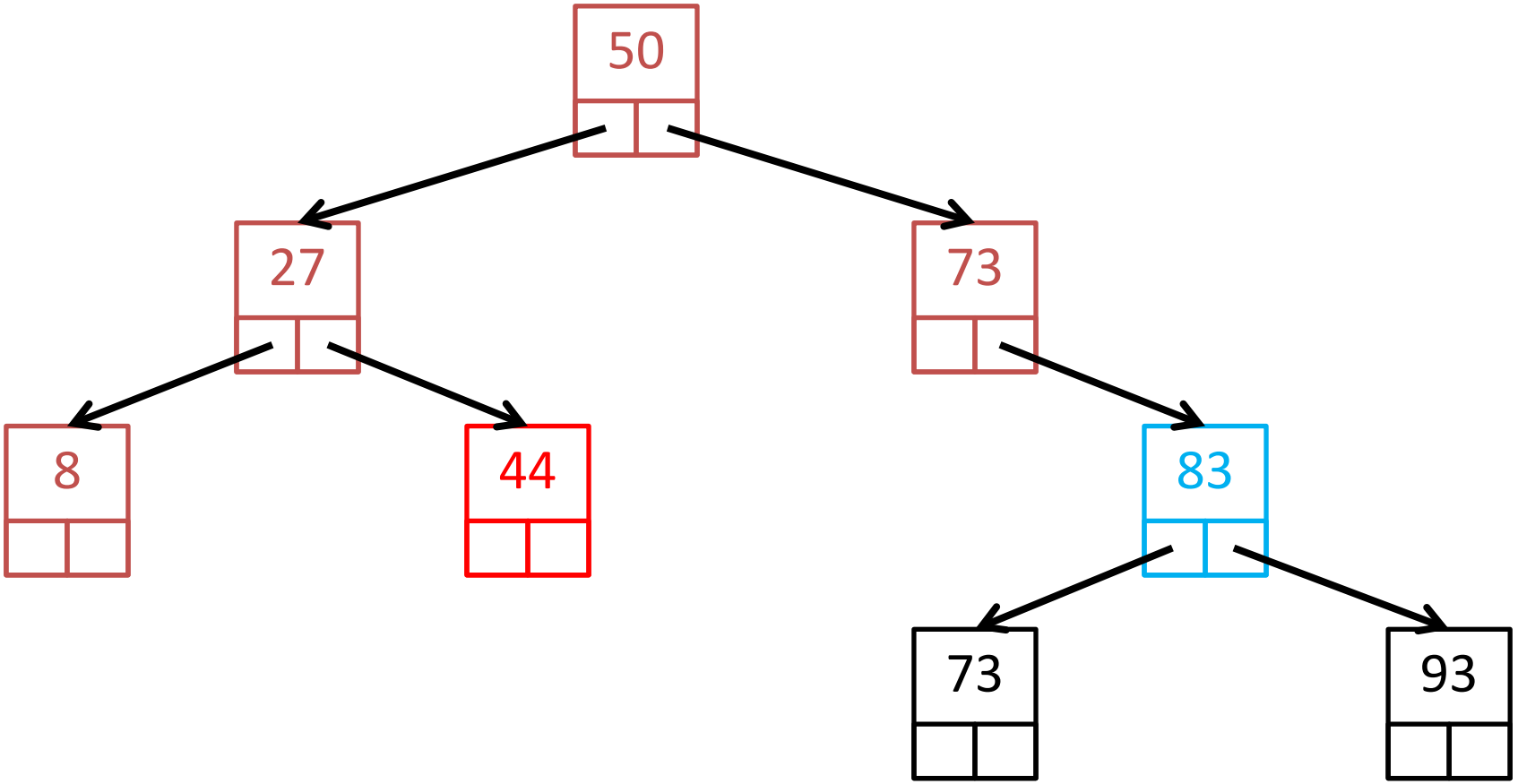
BFS: 50, 27, 73

dequeue 73,
enqueue right



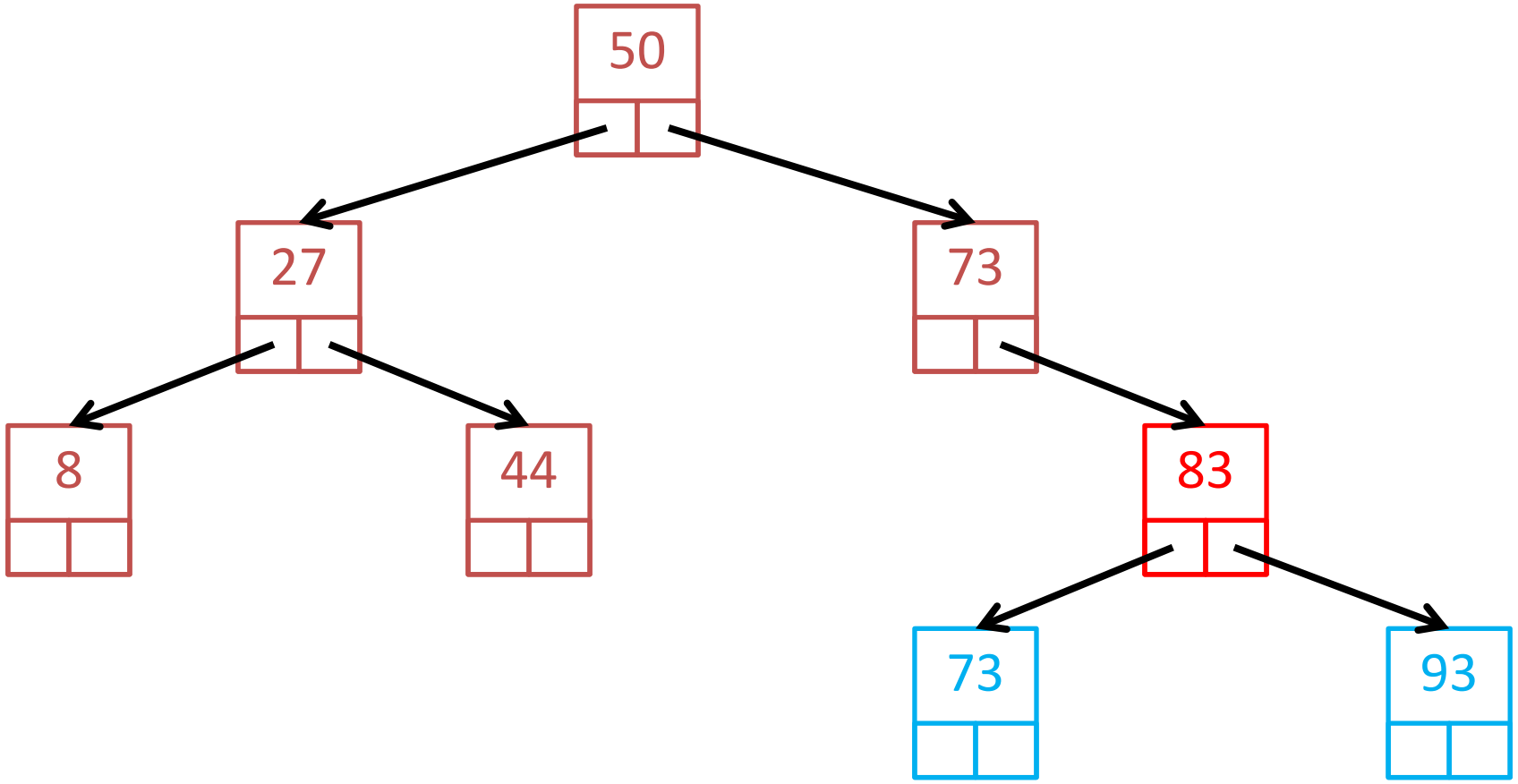
BFS: 50, 27, 73, 8

dequeue 8



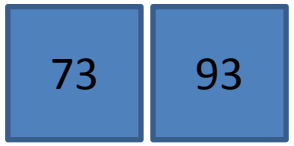
BFS: 50, 27, 73, 8, 44

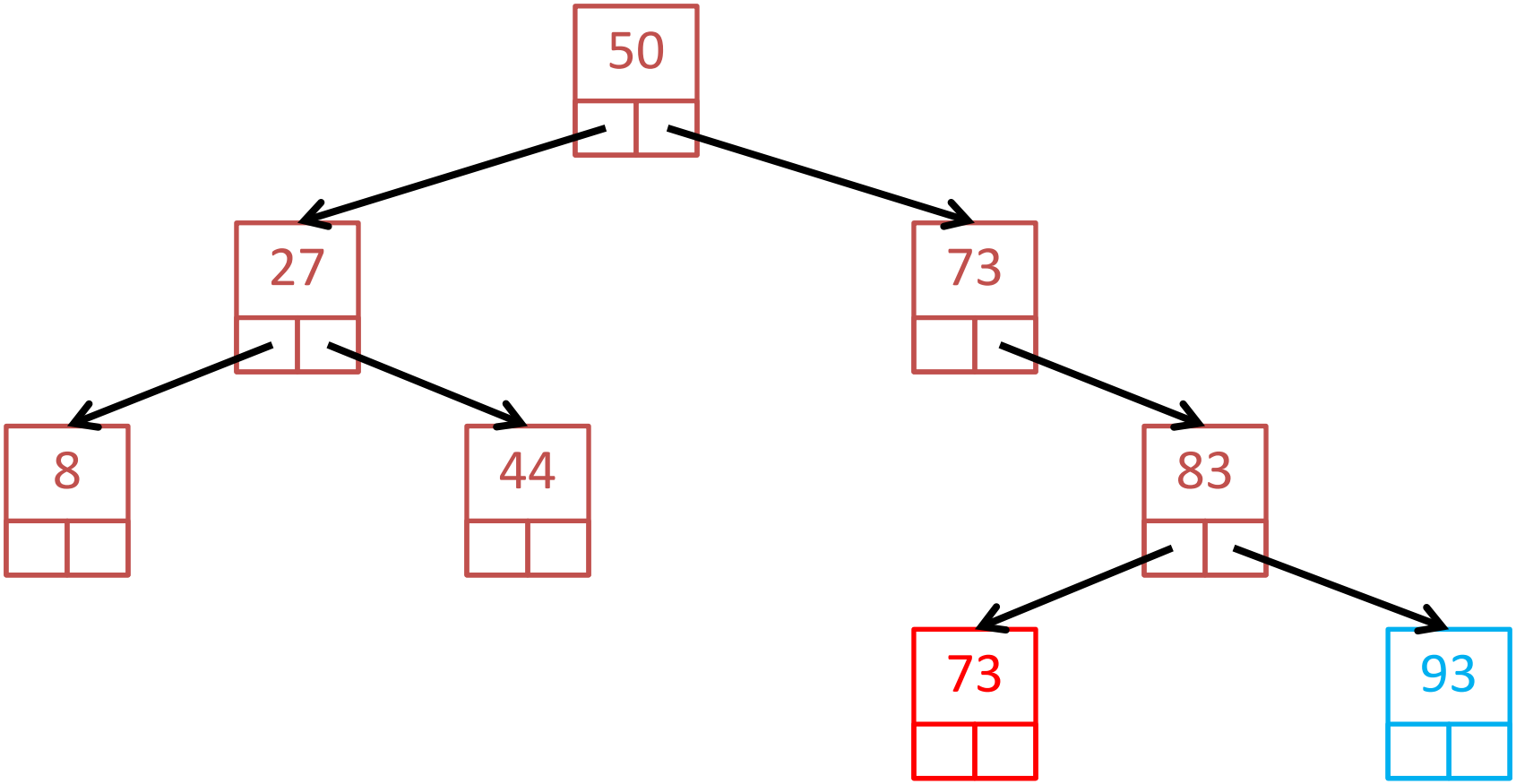
dequeue 44



BFS: 50, 27, 73, 8, 44, 83

dequeue 83,
enqueue left and right

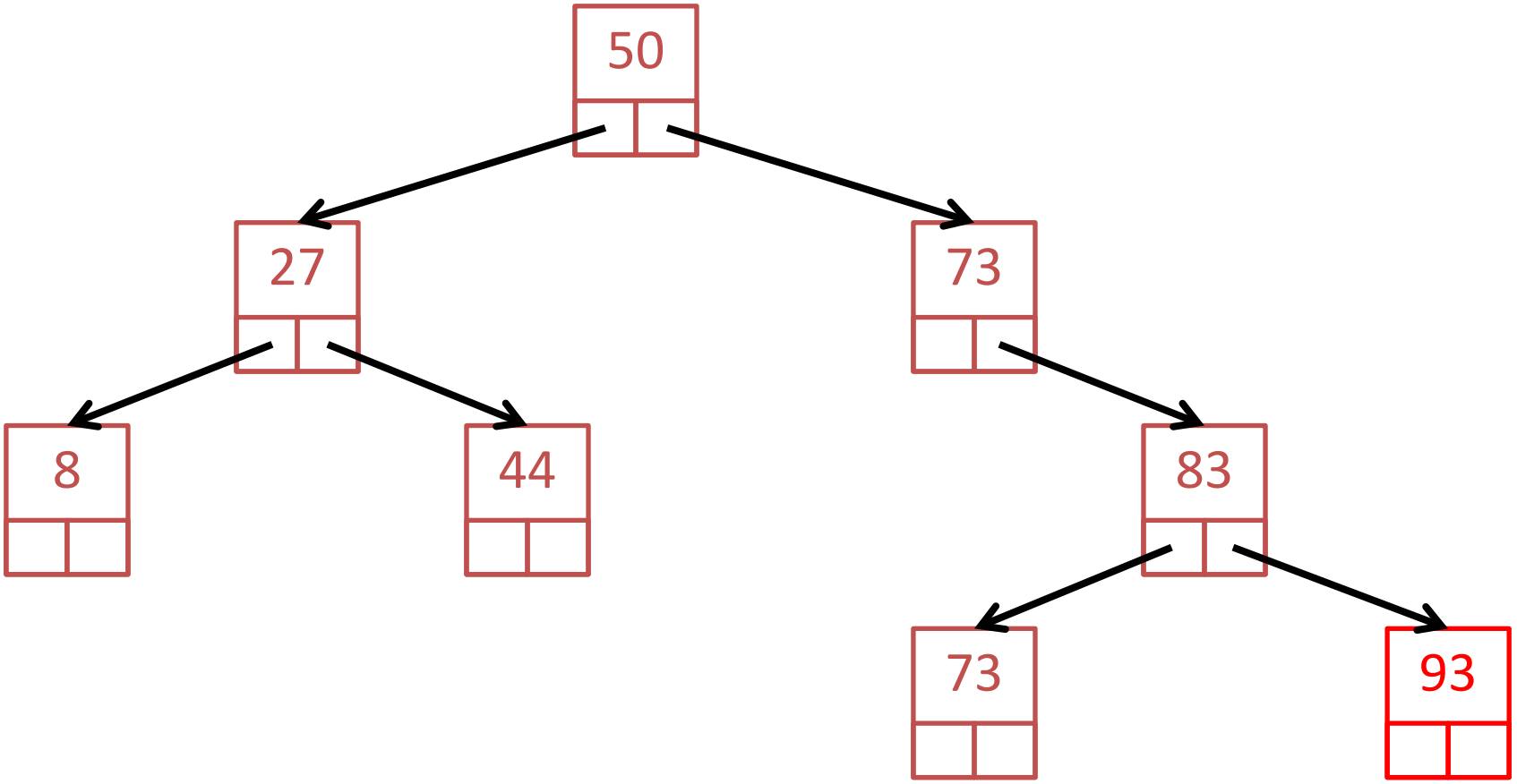




BFS: 50, 27, 73, 8, 44, 83, 73

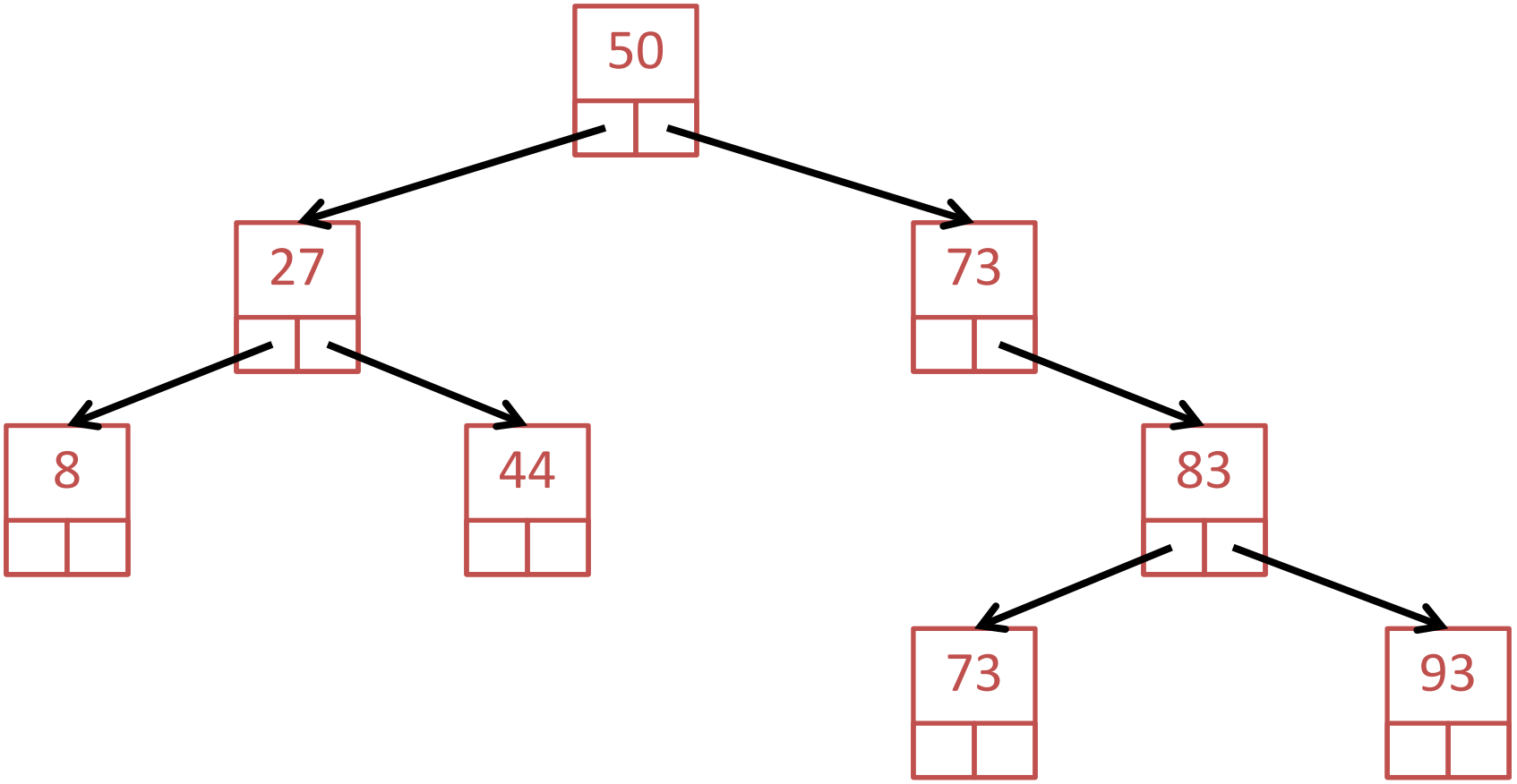
dequeue 73

93



BFS: 50, 27, 73, 8, 44, 83, 73, 93

dequeue 93



BFS: 50, 27, 73, 8, 44, 83, 73, 93

queue empty