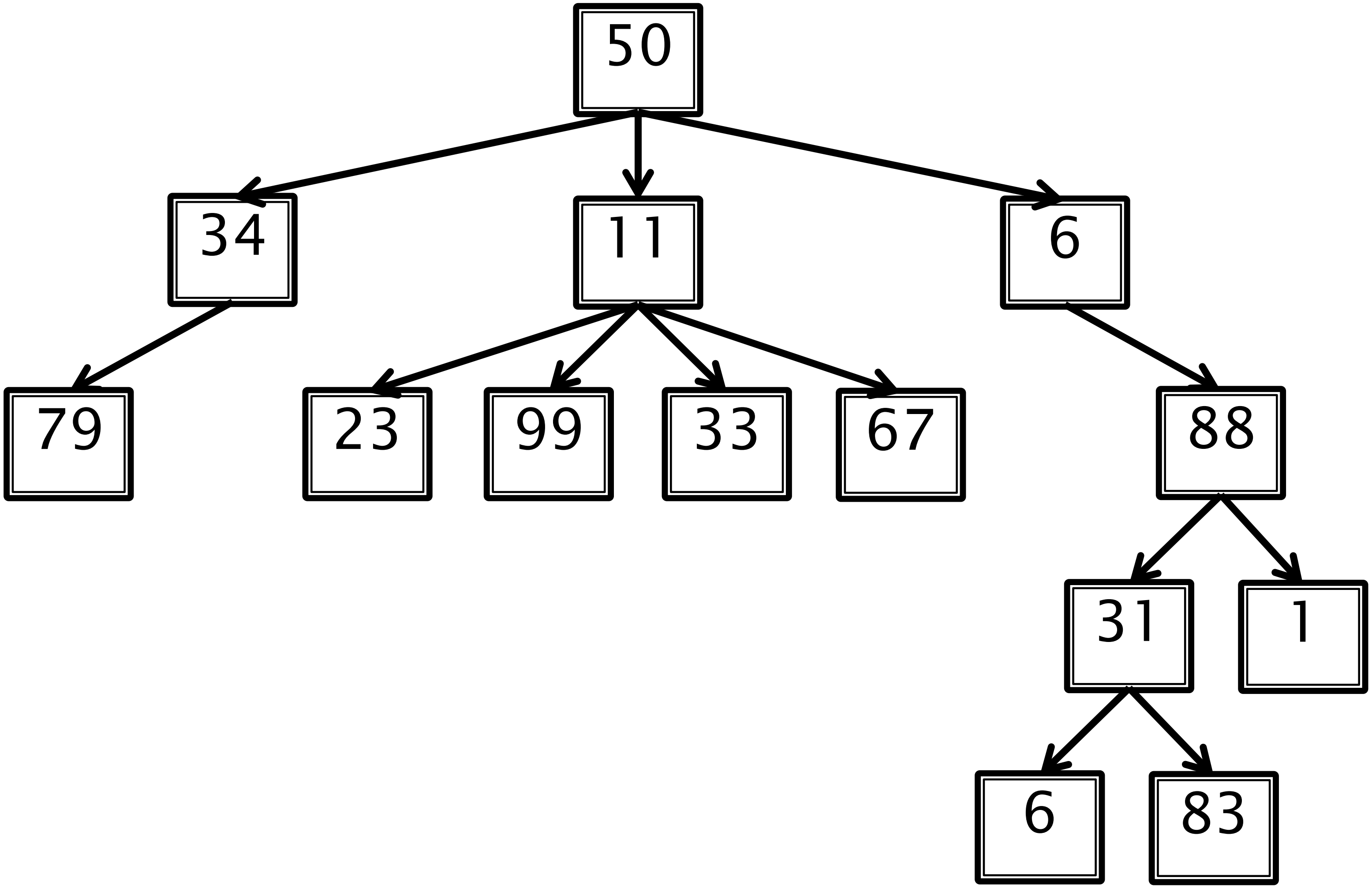
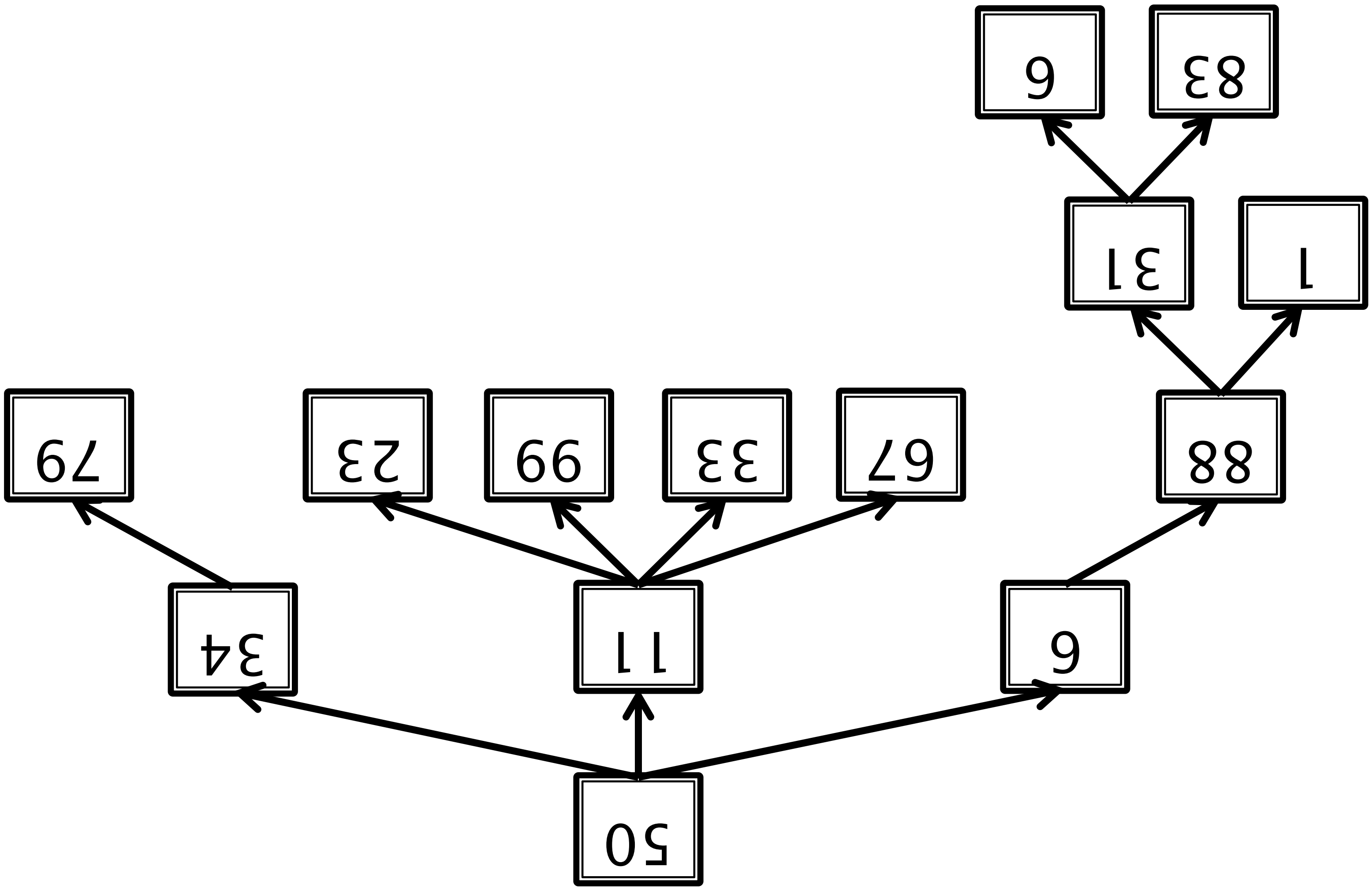
# Binary Trees

Based on slides by Prof. Burton Ma

# Trees

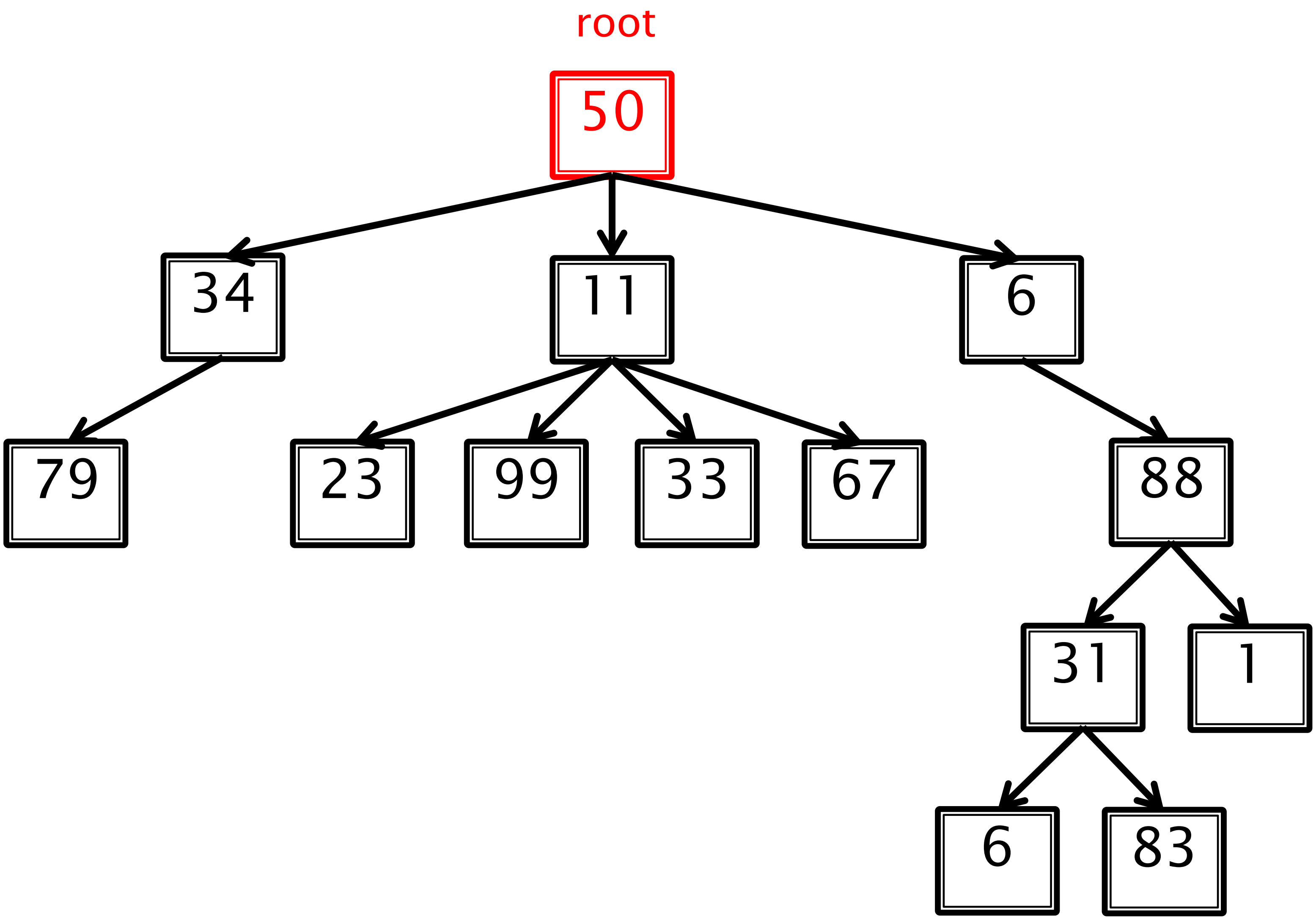
- A tree is a data structure made up of nodes
  - Each node stores data
  - Each node has links to zero or more nodes in the next level of the tree
    - Children of the node
  - Each node has exactly one parent node
    - Except for the root node


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



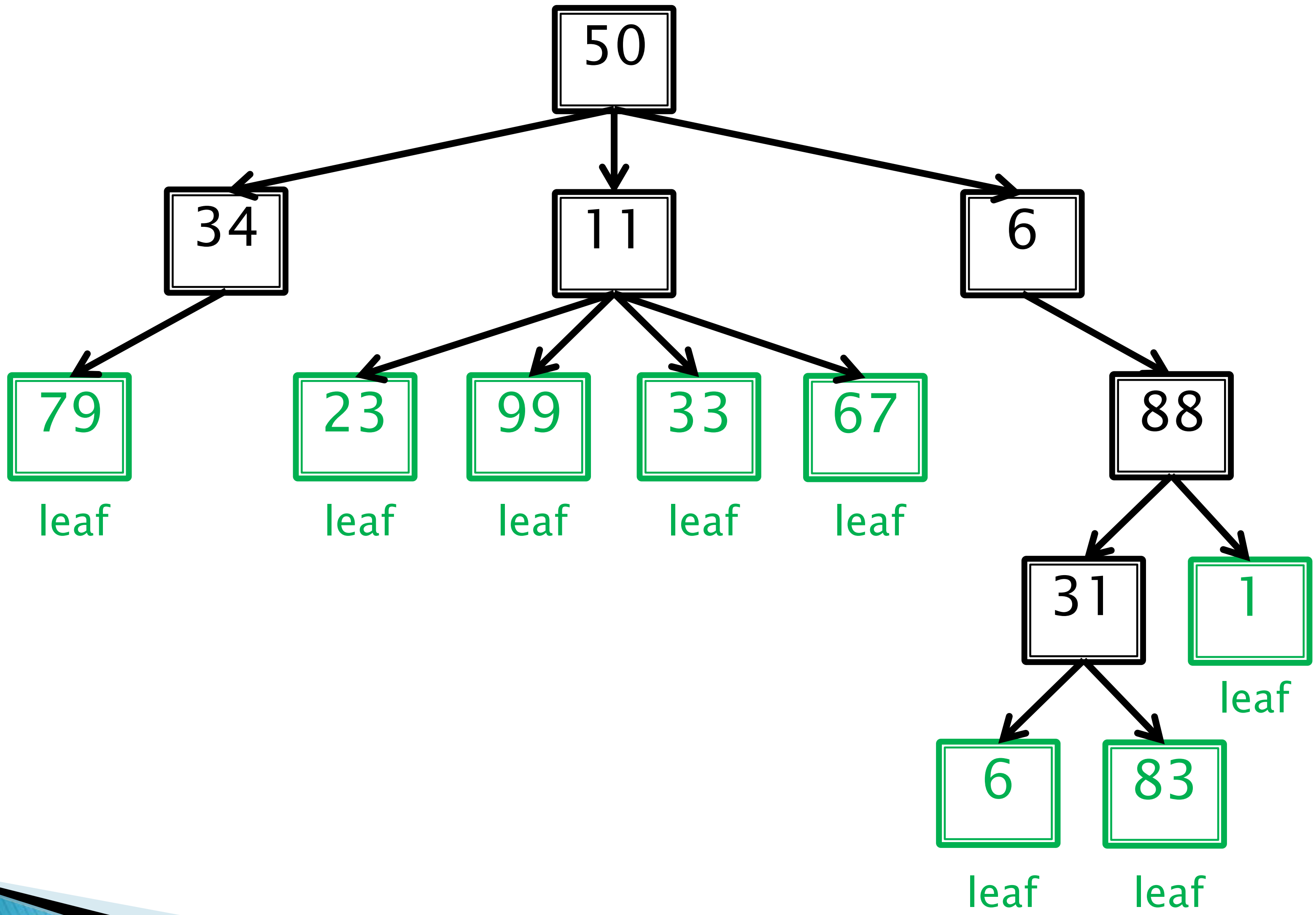
# Trees

- 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

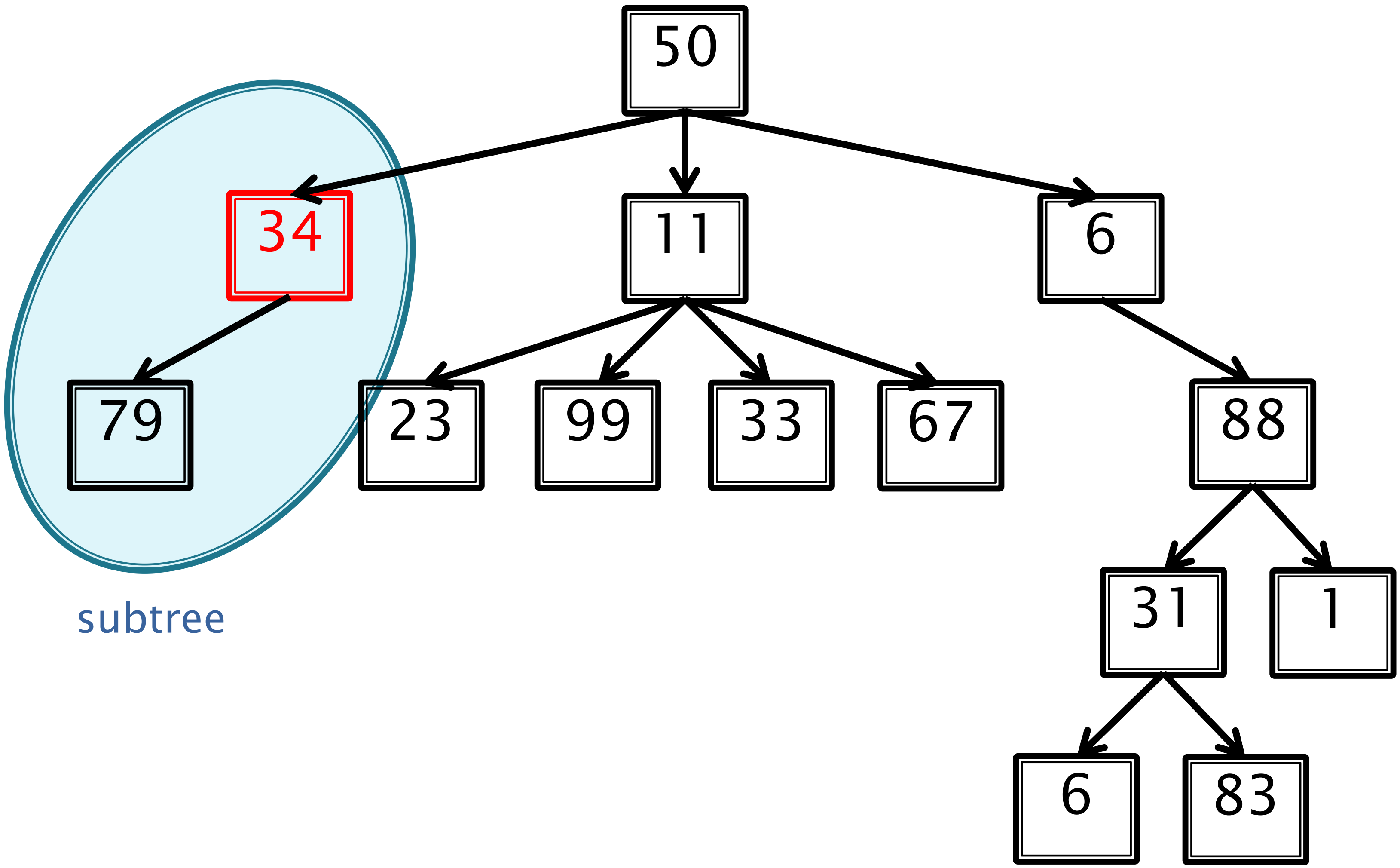
- A node without any children is called a leaf

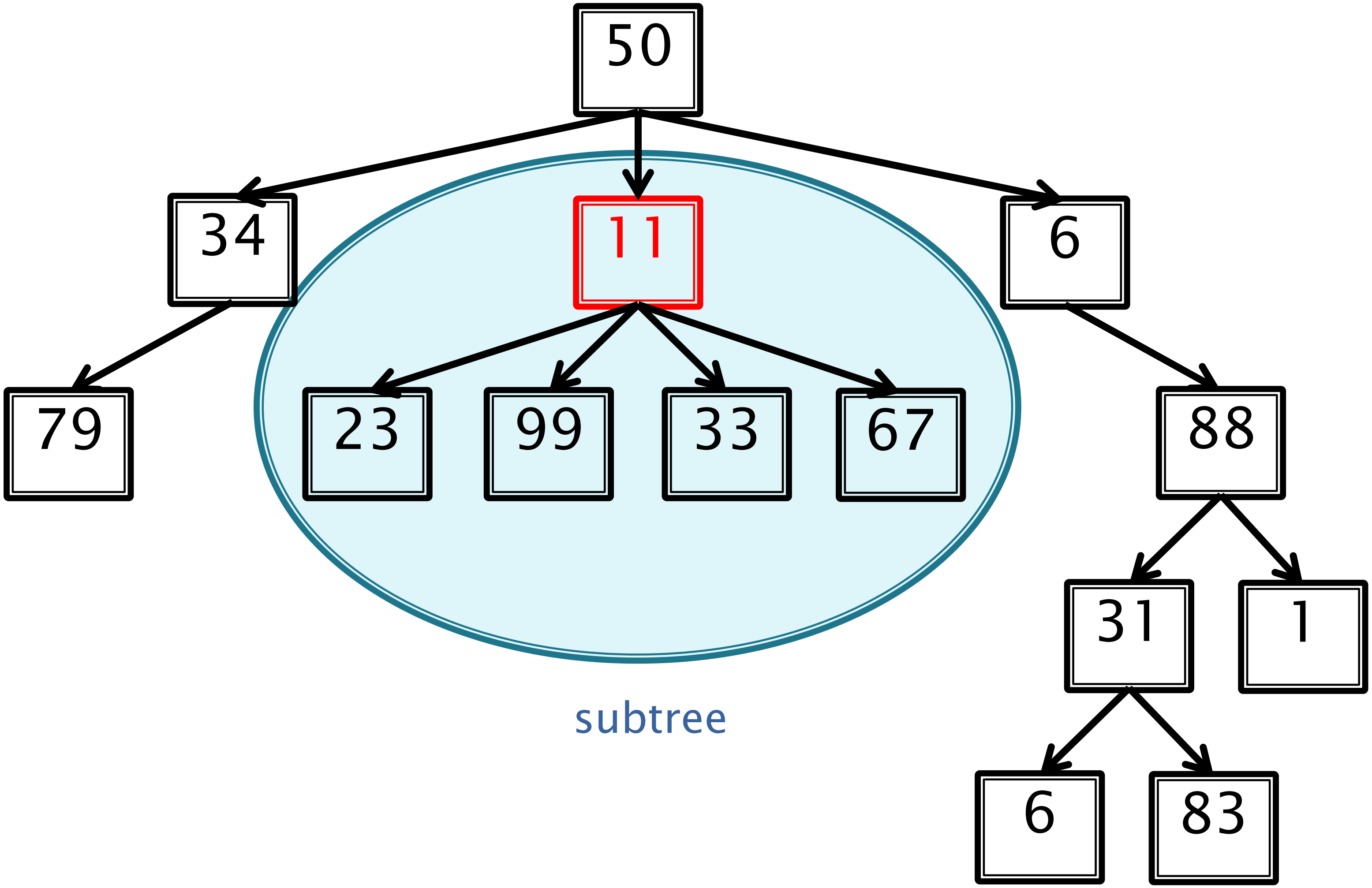


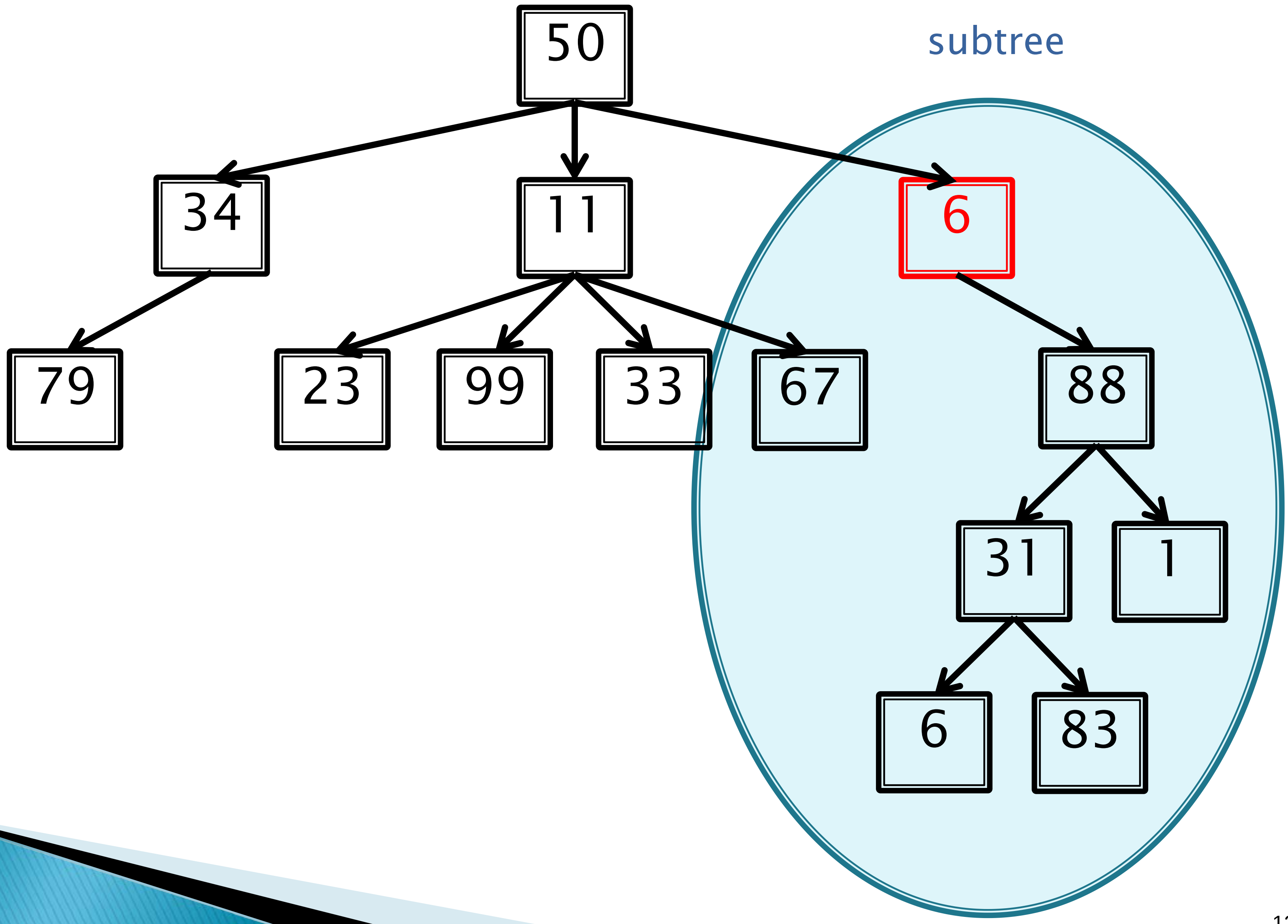
50
34
11
6
79
leaf
23
leaf
99
leaf
33
leaf
67
leaf
88
31
1
leaf
6
leaf
83
leaf

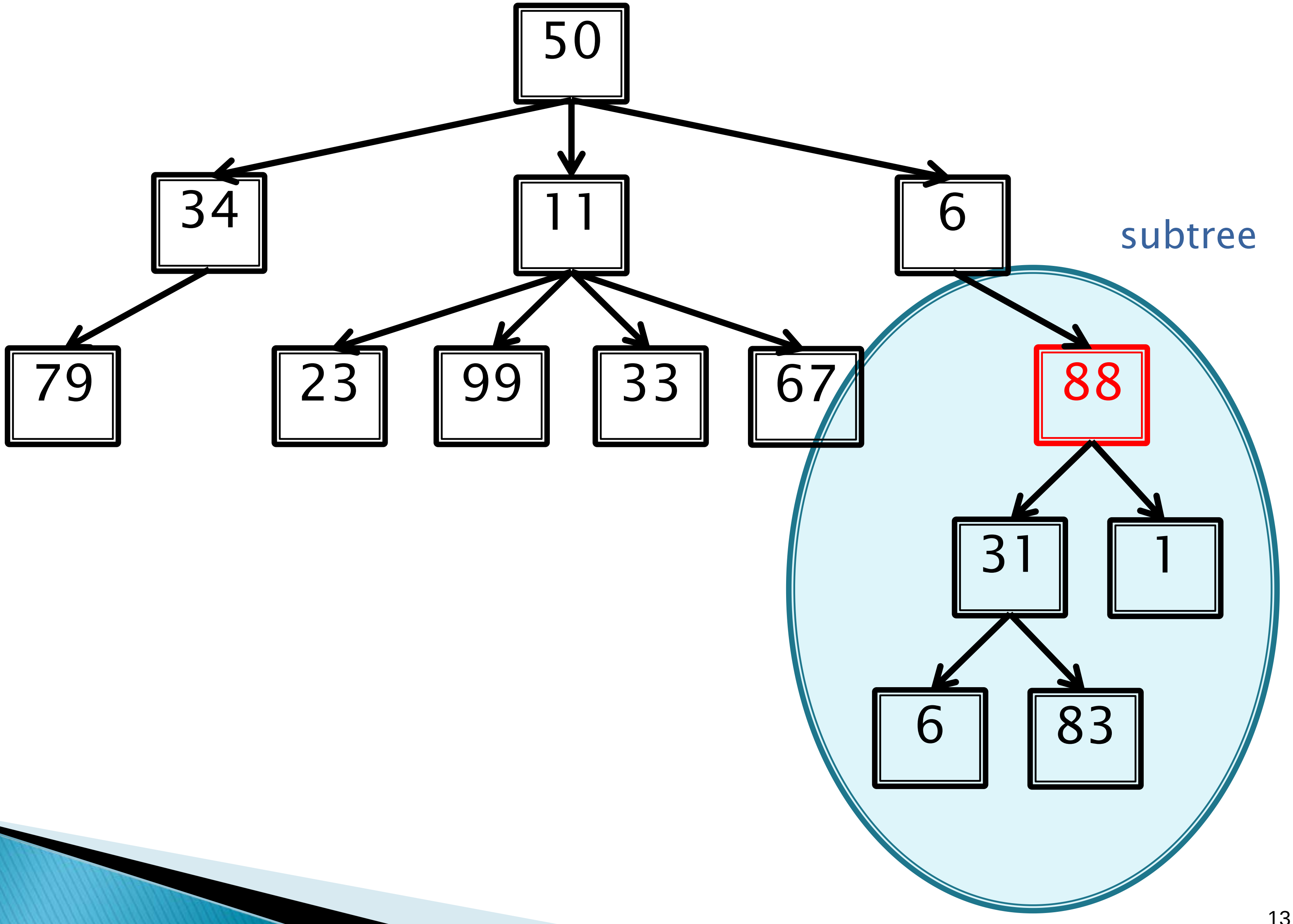
# Trees

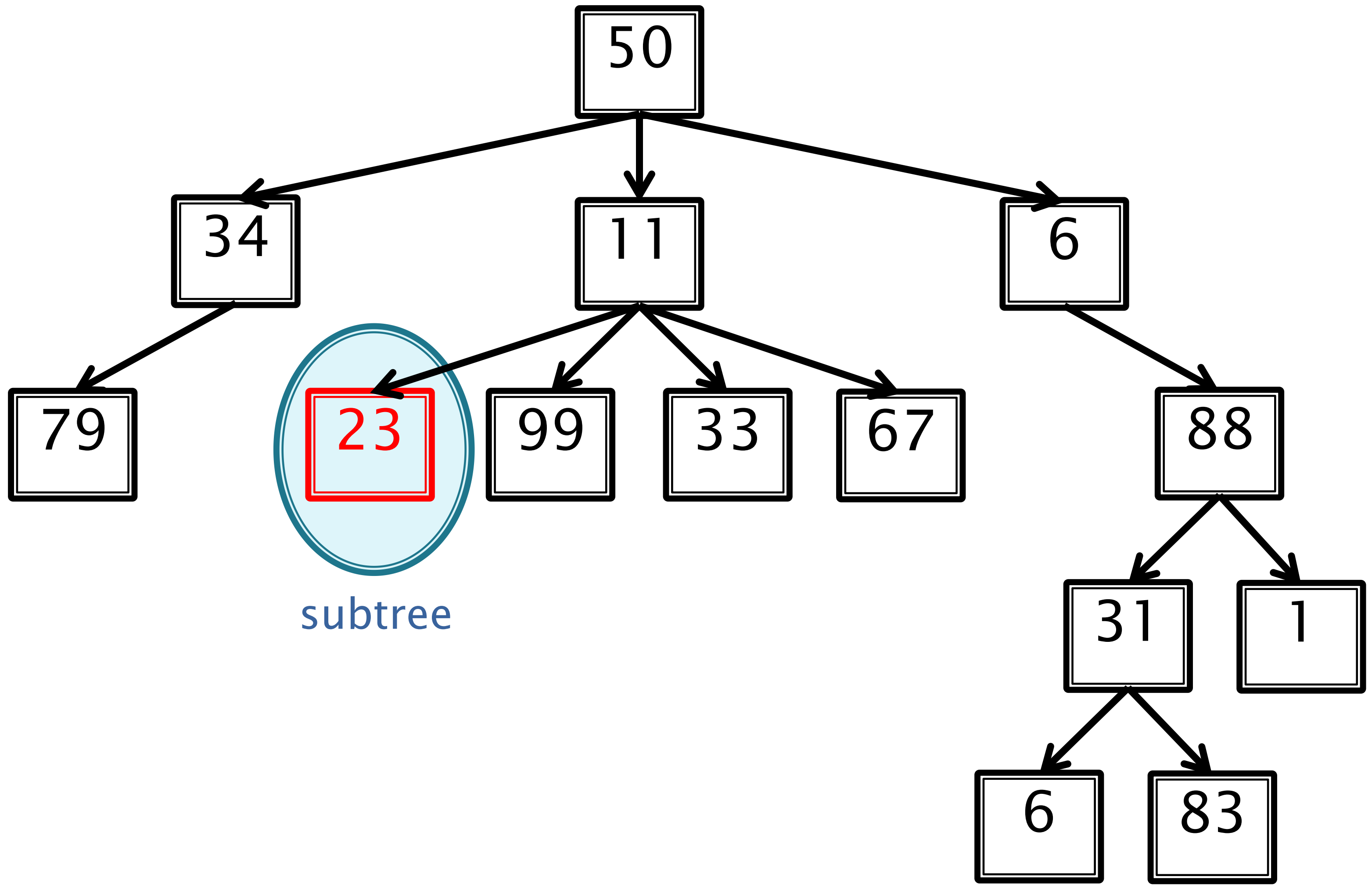
- The recursive structure of a tree means that every node is the root of a tree


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

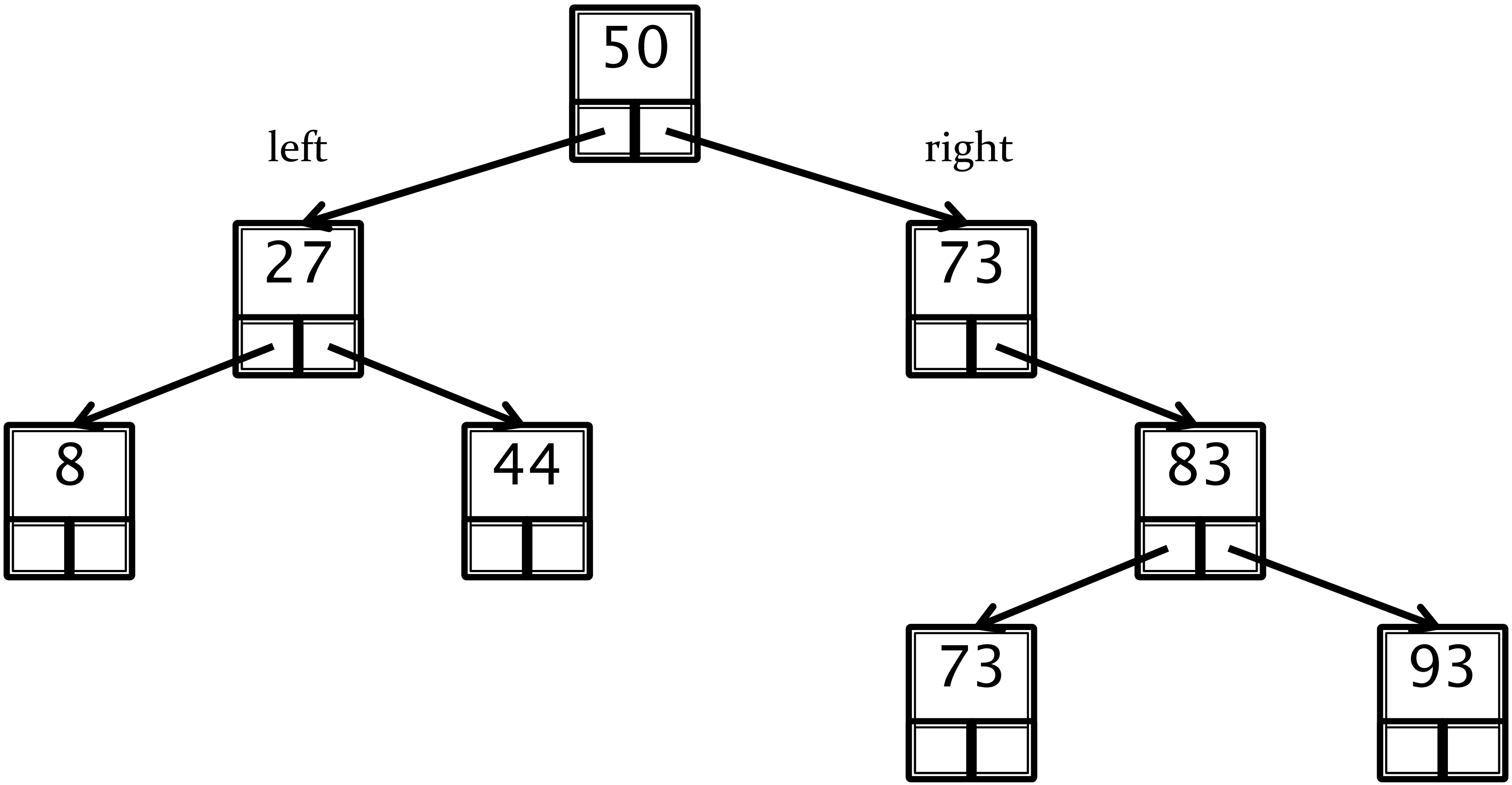

subtree
50
34
11
6
79
23
99
33
67
88
31
1
6
83

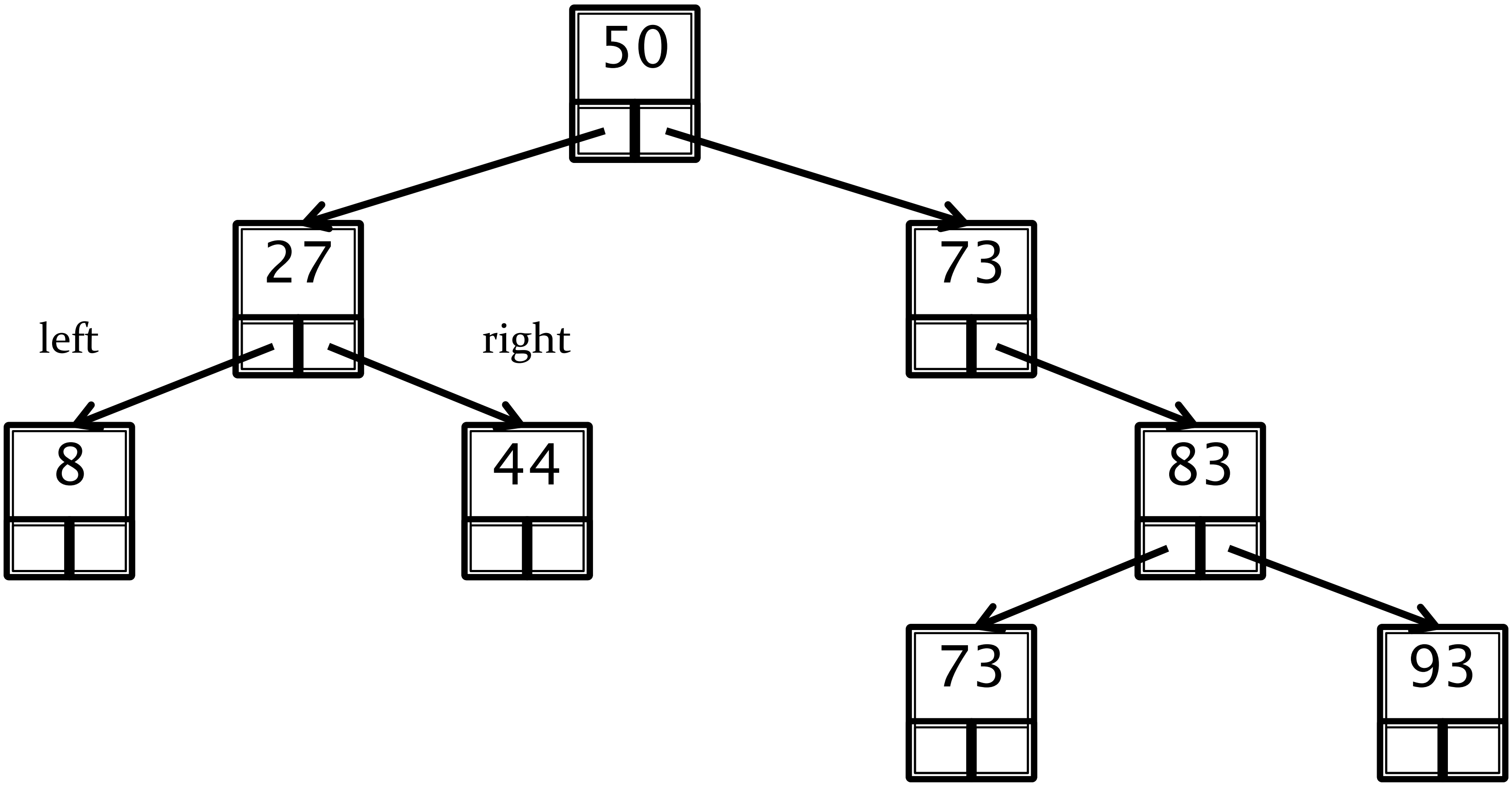

50
34
11
6
subtree
79
23
99
33
67
88
31
1
6
83

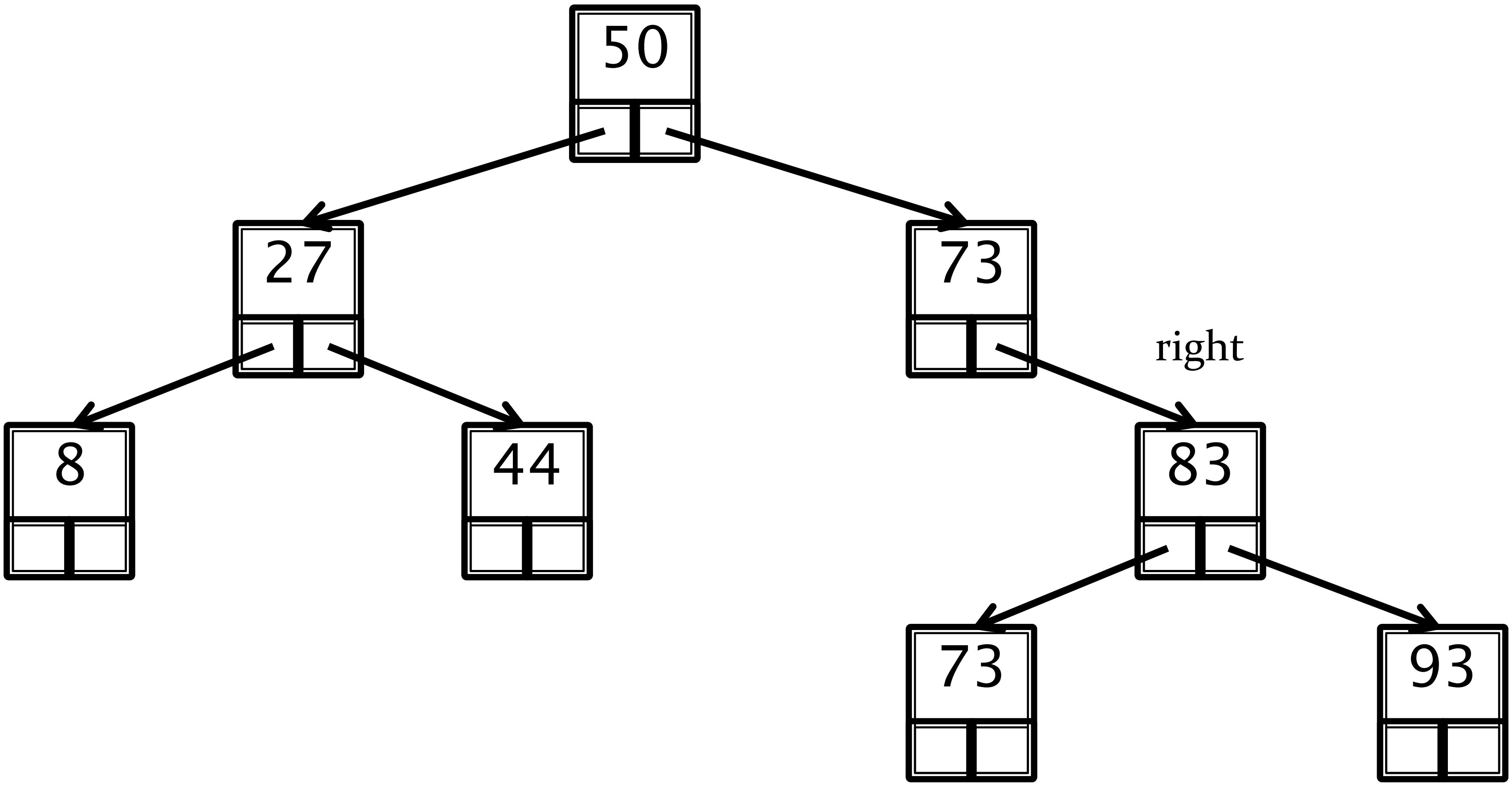

50
34
11
6
79
23
99
33
67
88
subtree
31
1
6
83

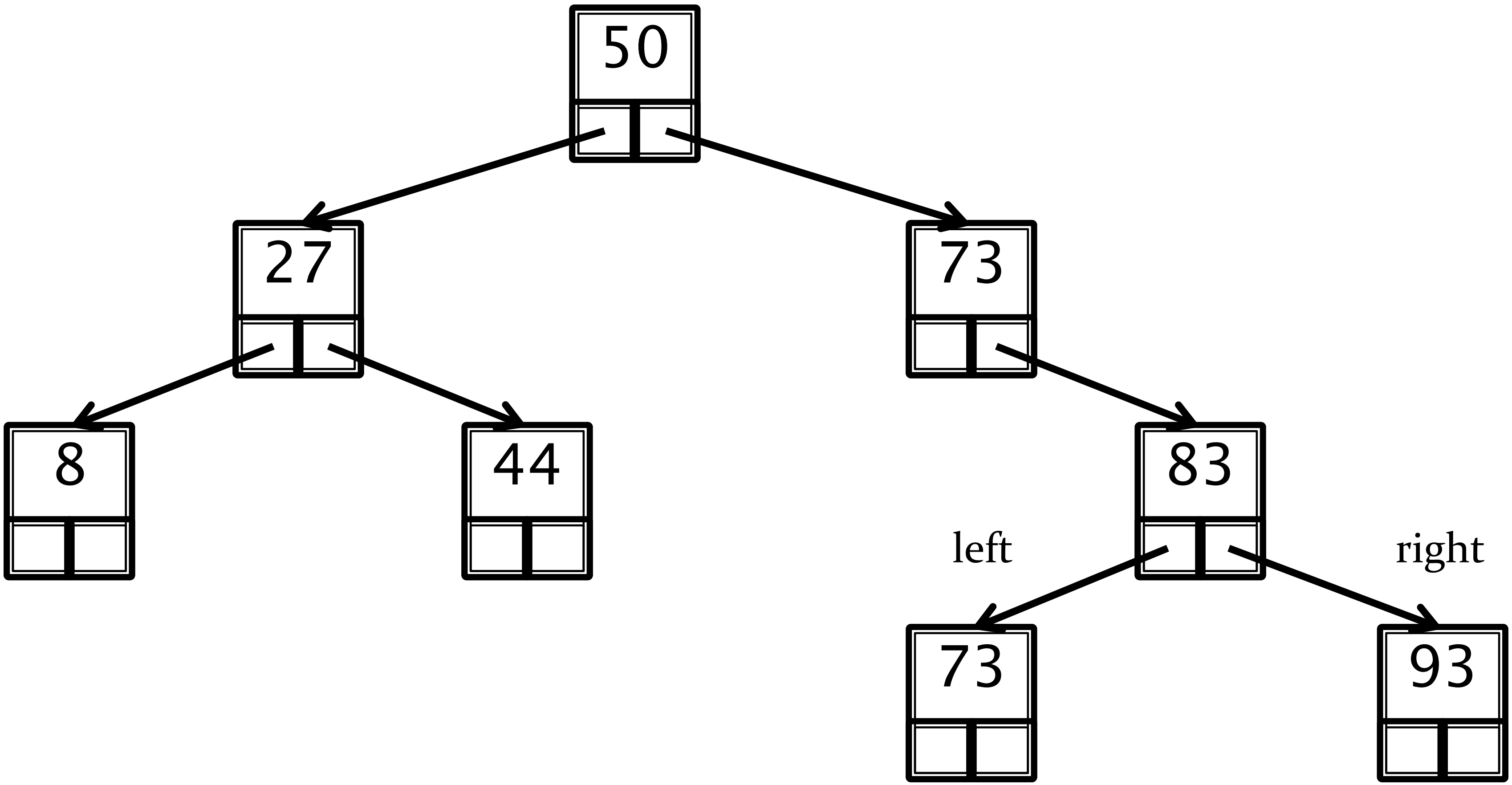
# Binary Tree

- 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
- Binary Search Tree:
  - All data in left subtree is “less than” data at root
  - All data in right subtree is “greater than” data at root
  - The designer can decide where to put “equal” data, or to have only unique values (i.e., like a set)


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

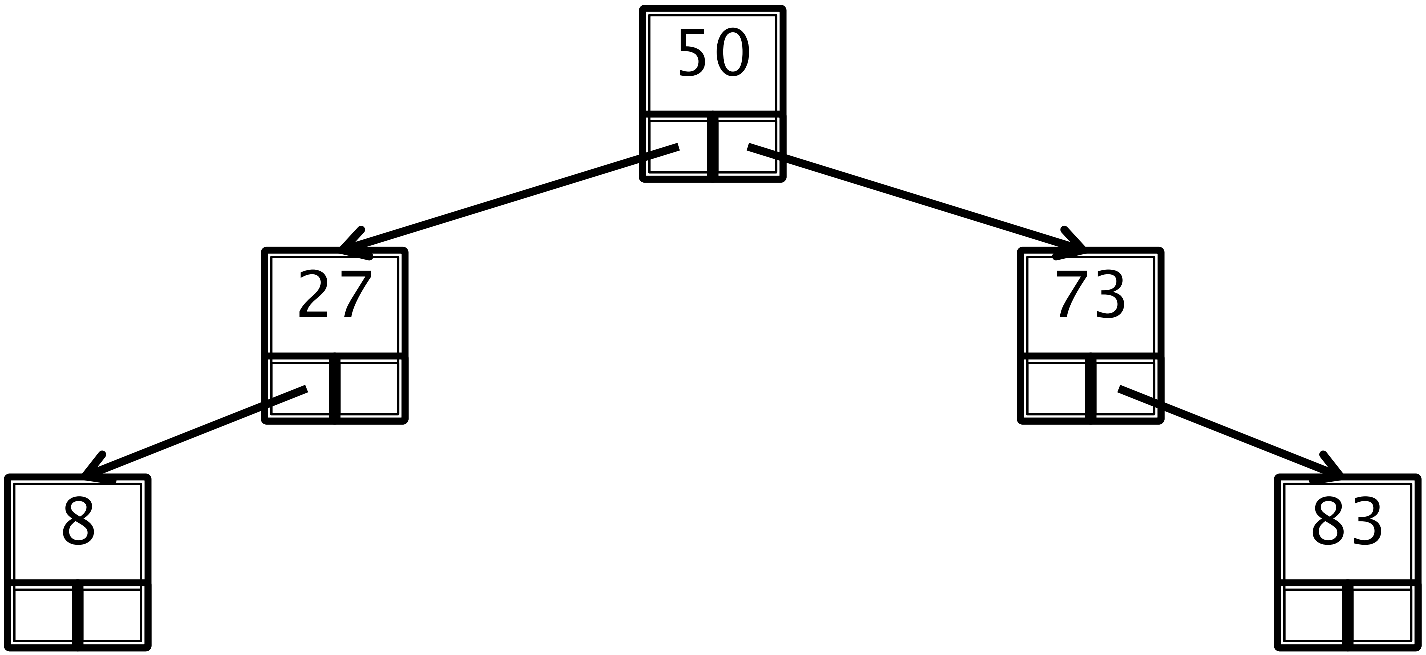
# Building a Binary Search Tree

- Need an inner class representing a node
- We will cover adding and deleting nodes, implementing other methods is left as exercises
- Pseudo code presented in slides, Java code available on course website

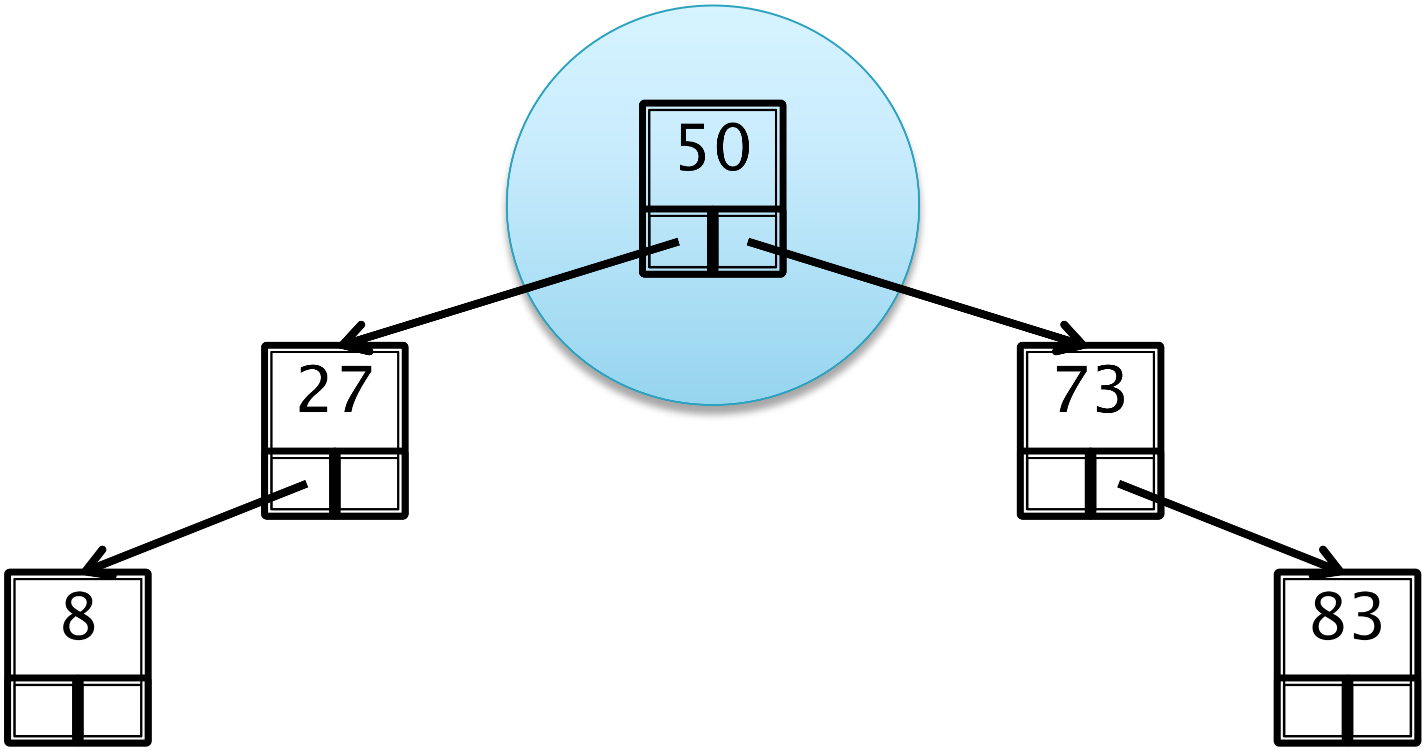
# Adding Nodes

- Step 1:
  - If the tree is empty, make it the root

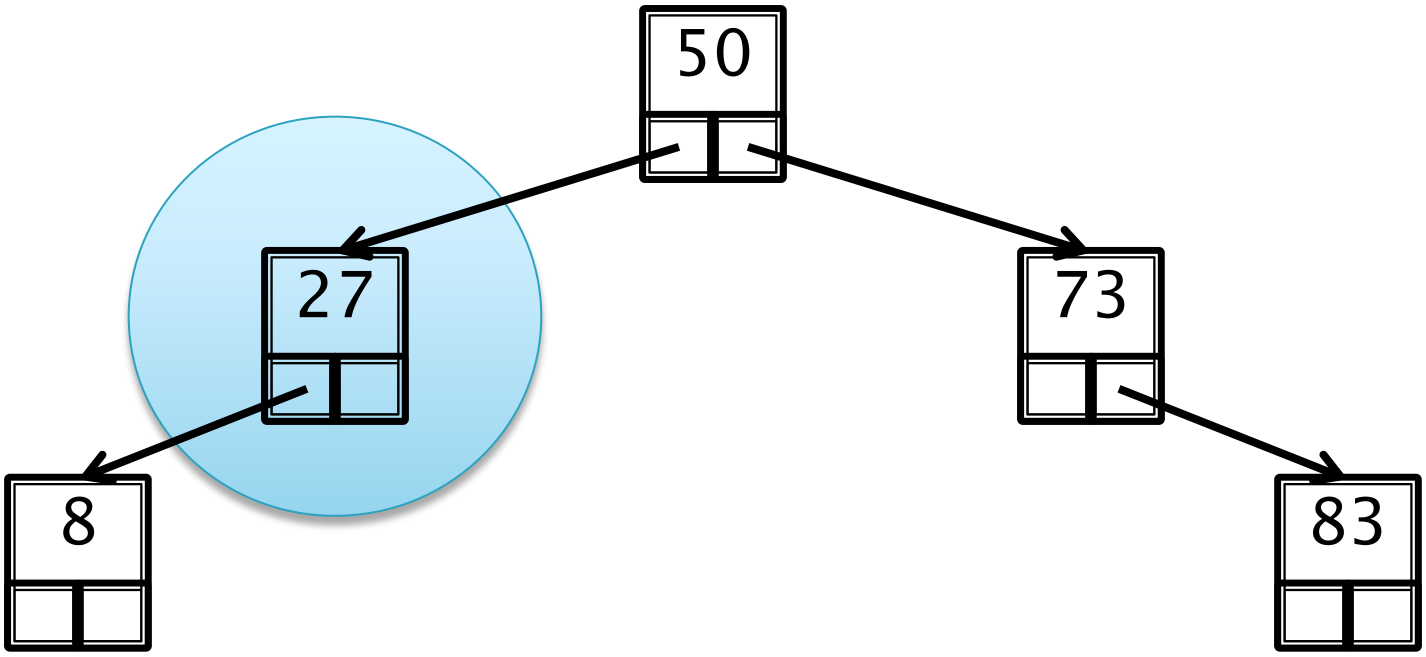
- Step 2:
  - If the tree is not empty, traverse to the left or right child (depending if data is larger than root or smaller than root, respectively) and repeat Step 1



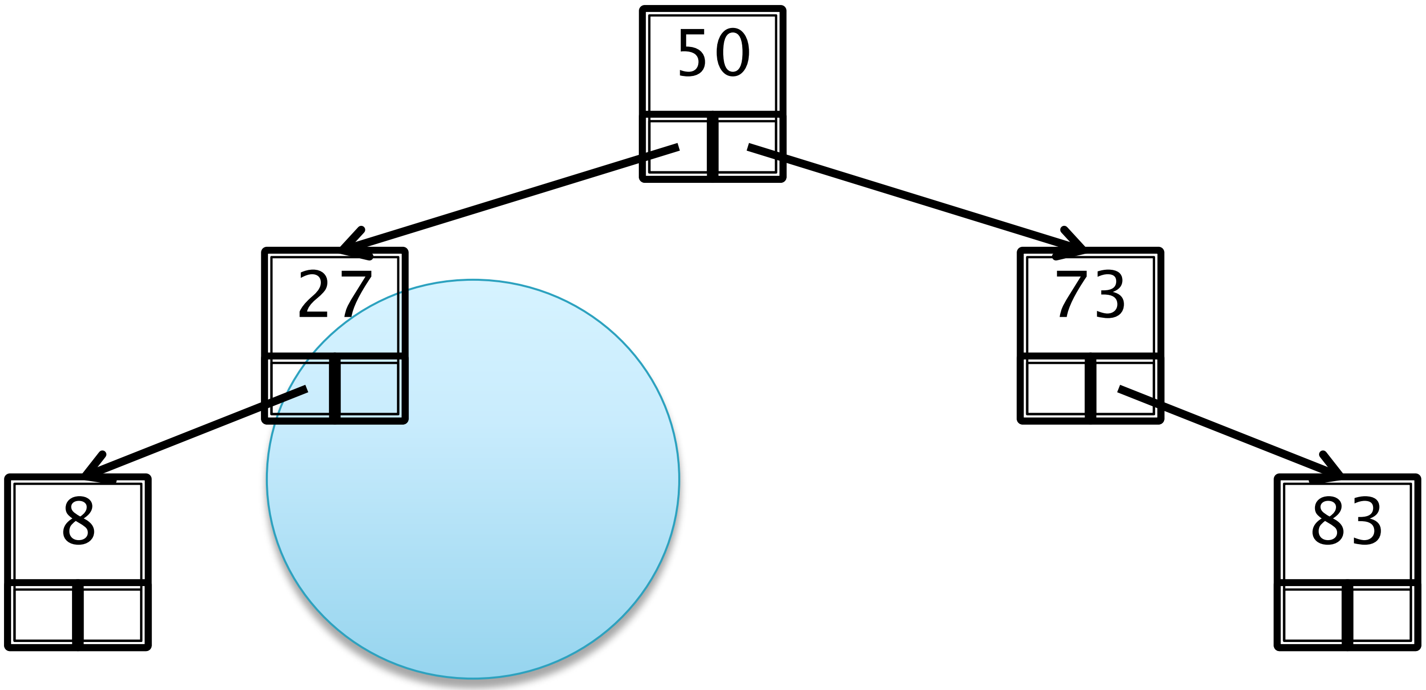

Insert the integer 44.



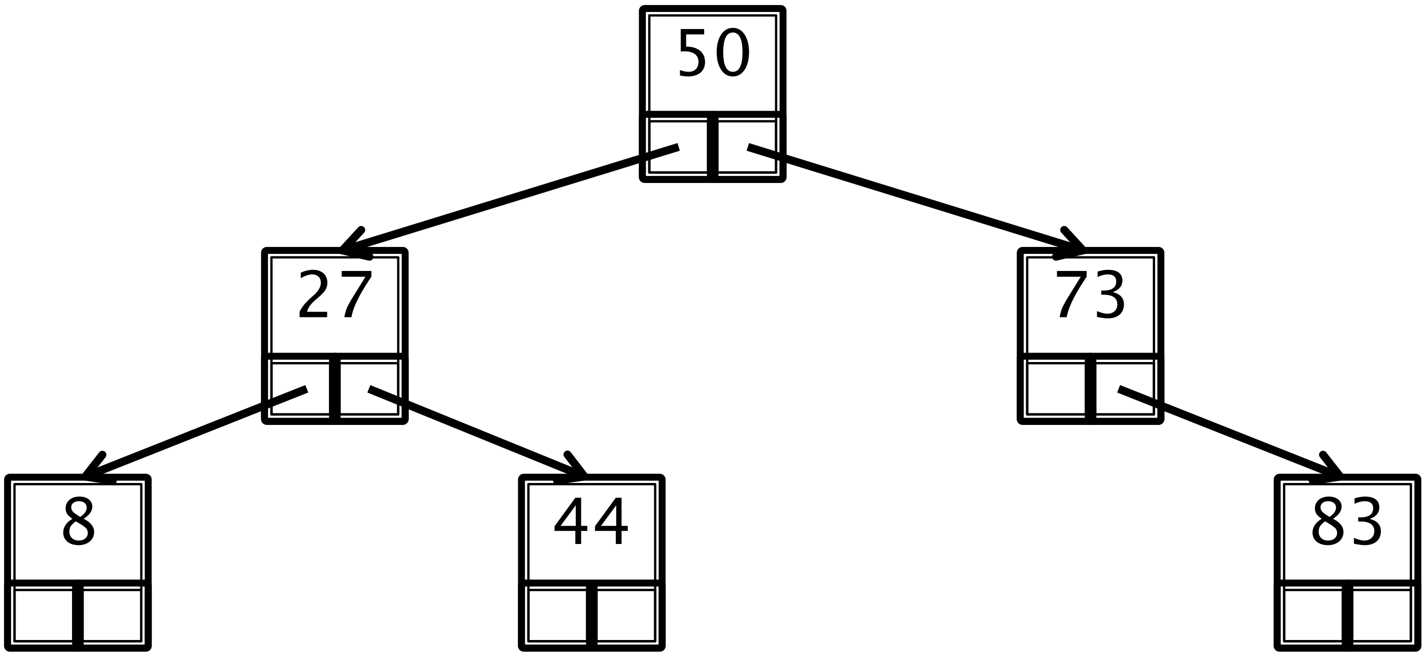

44 is less than 50 → go left




44 is greater than 27→ go right




The right subtree is empty → insert here

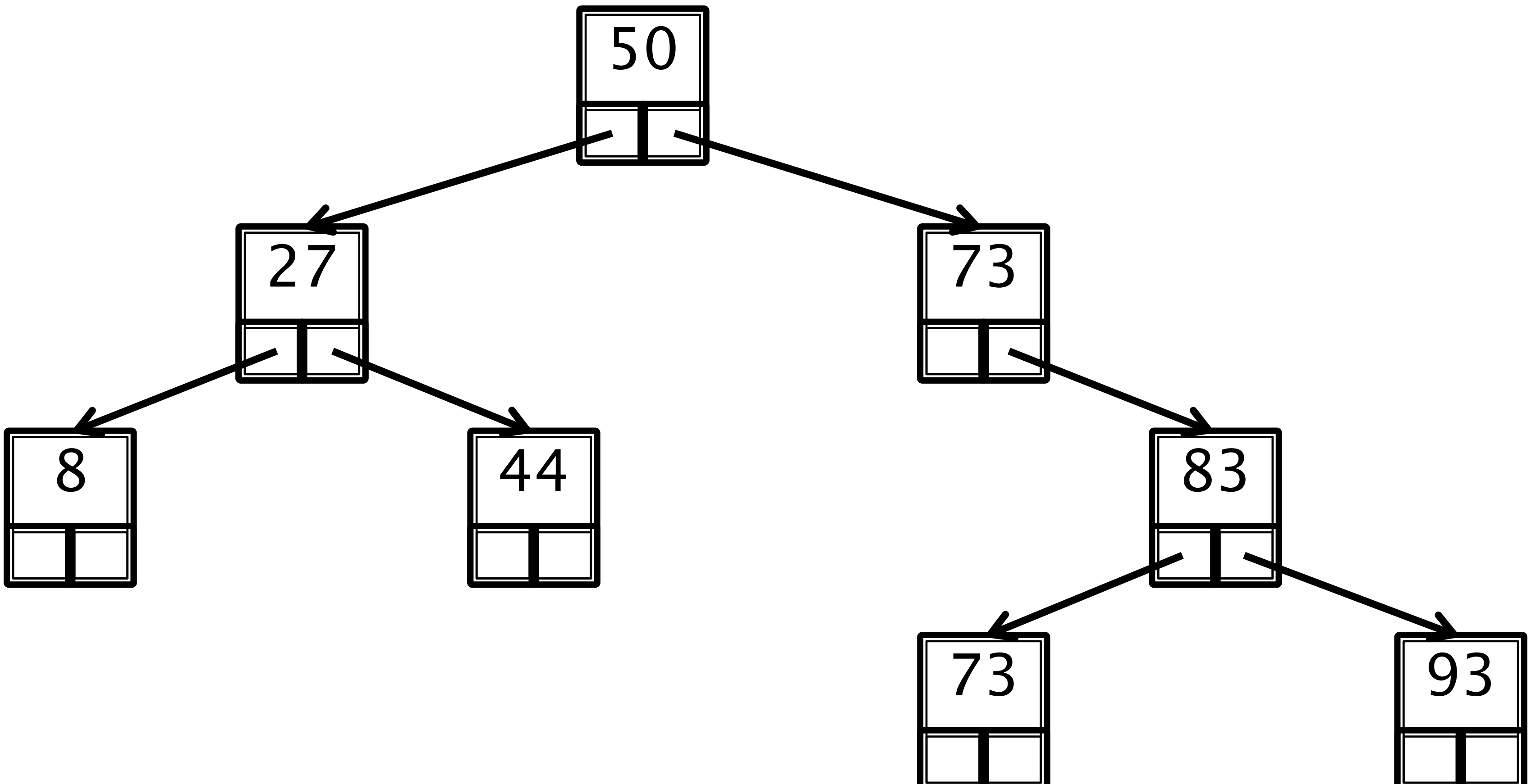


50
27
73
8
44
83

# Removing Nodes

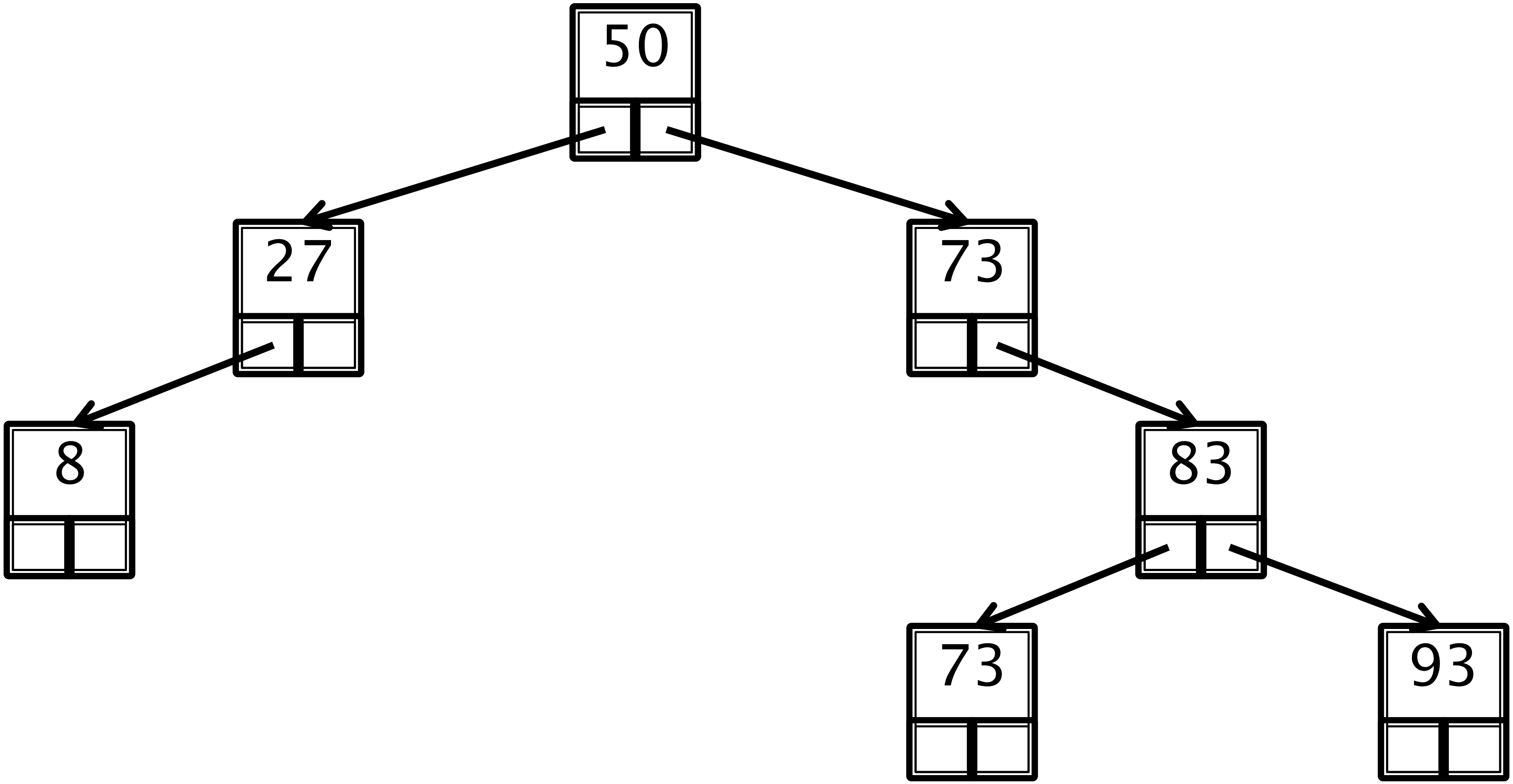
- Case 1: Node is a leaf
  - Easiest case, as there are no children to handle
- Case 2: Node has 1 child
  - Also easy, as the child replaces the removed node
- Case 3: Node has 2 children
  - Which descendant will replace removed node?
  - Largest descendant in left subtree

# Removing Nodes: Case 1

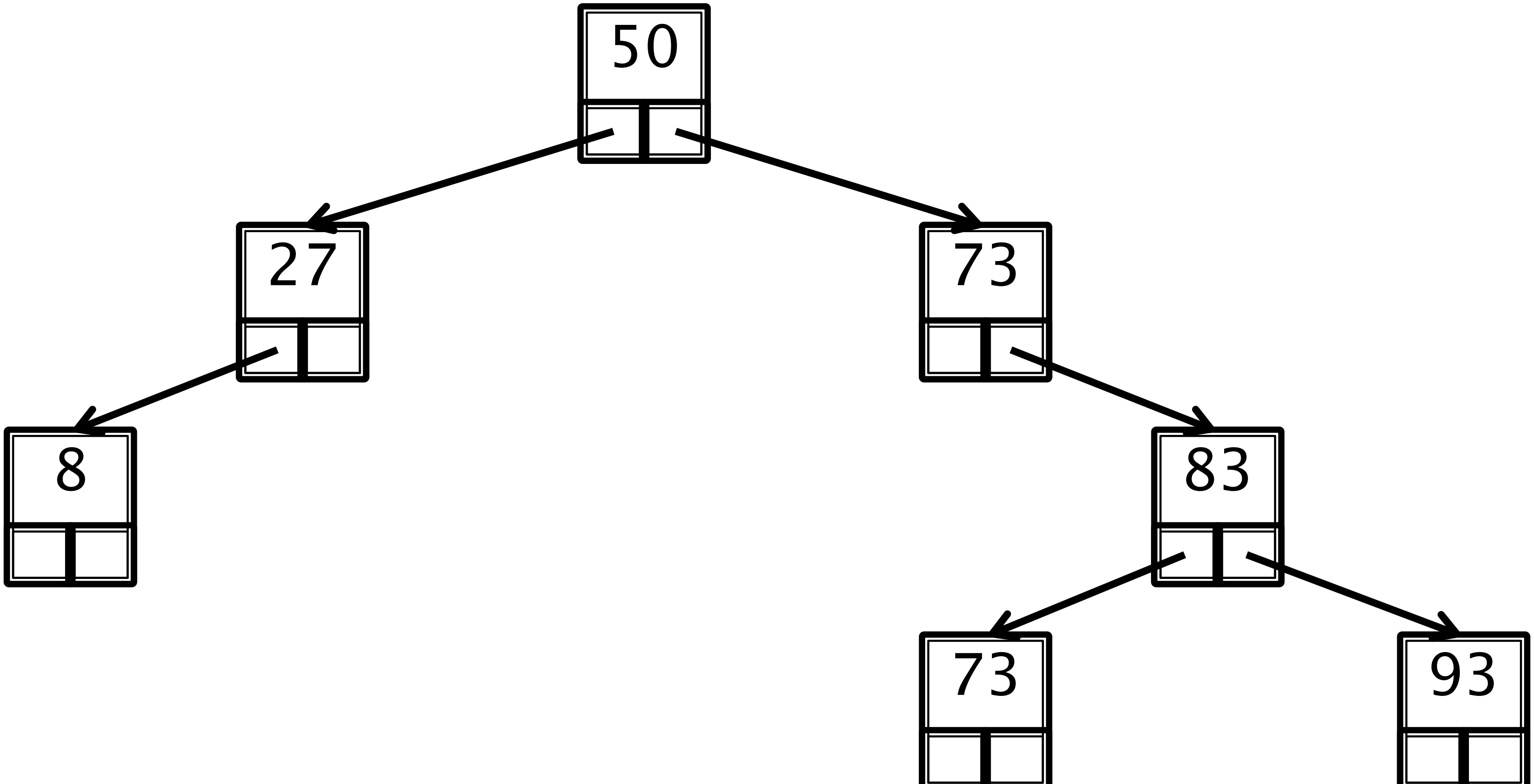



Remove 44

# Removing Nodes: Case 1

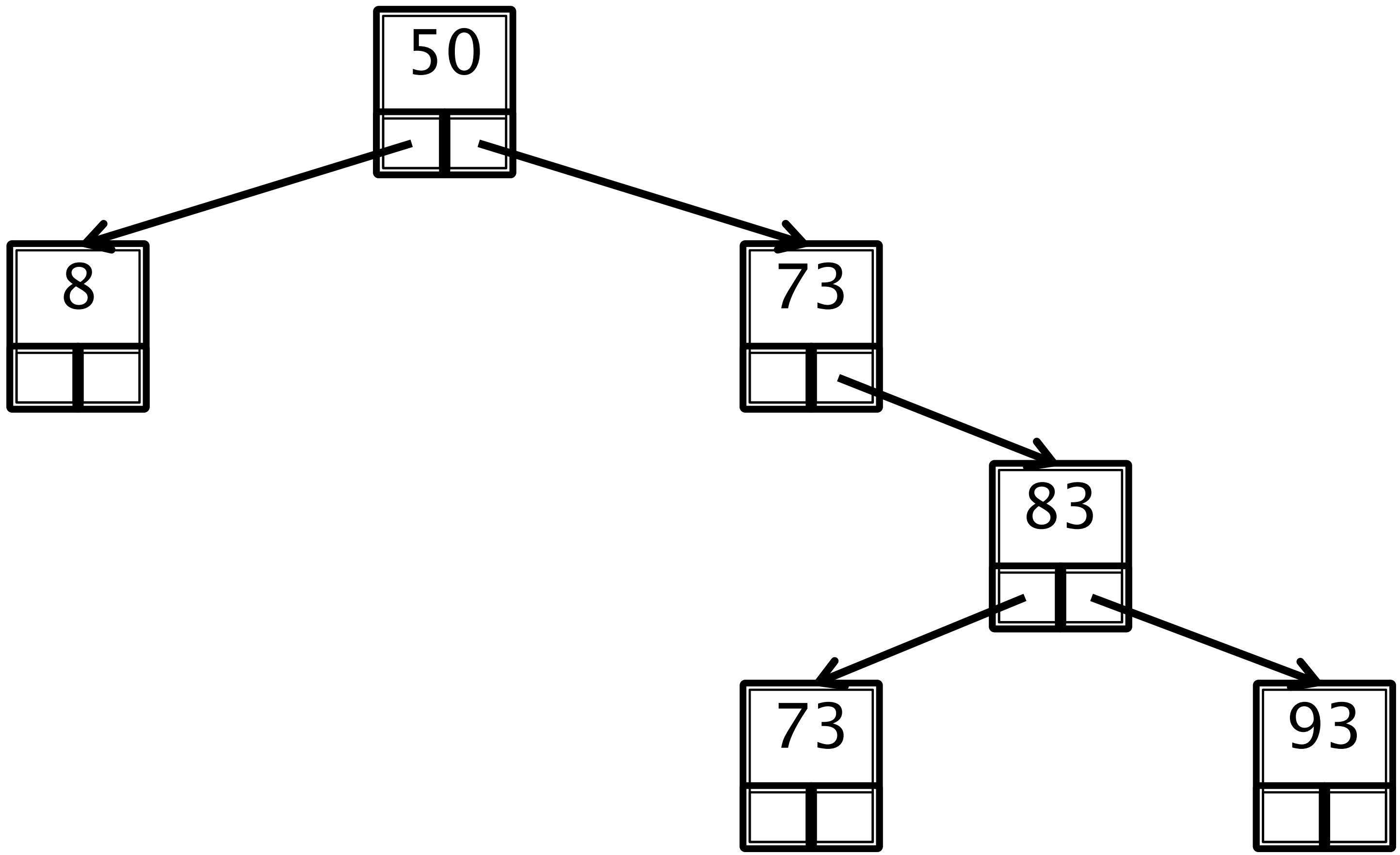


# Removing Nodes: Case 2

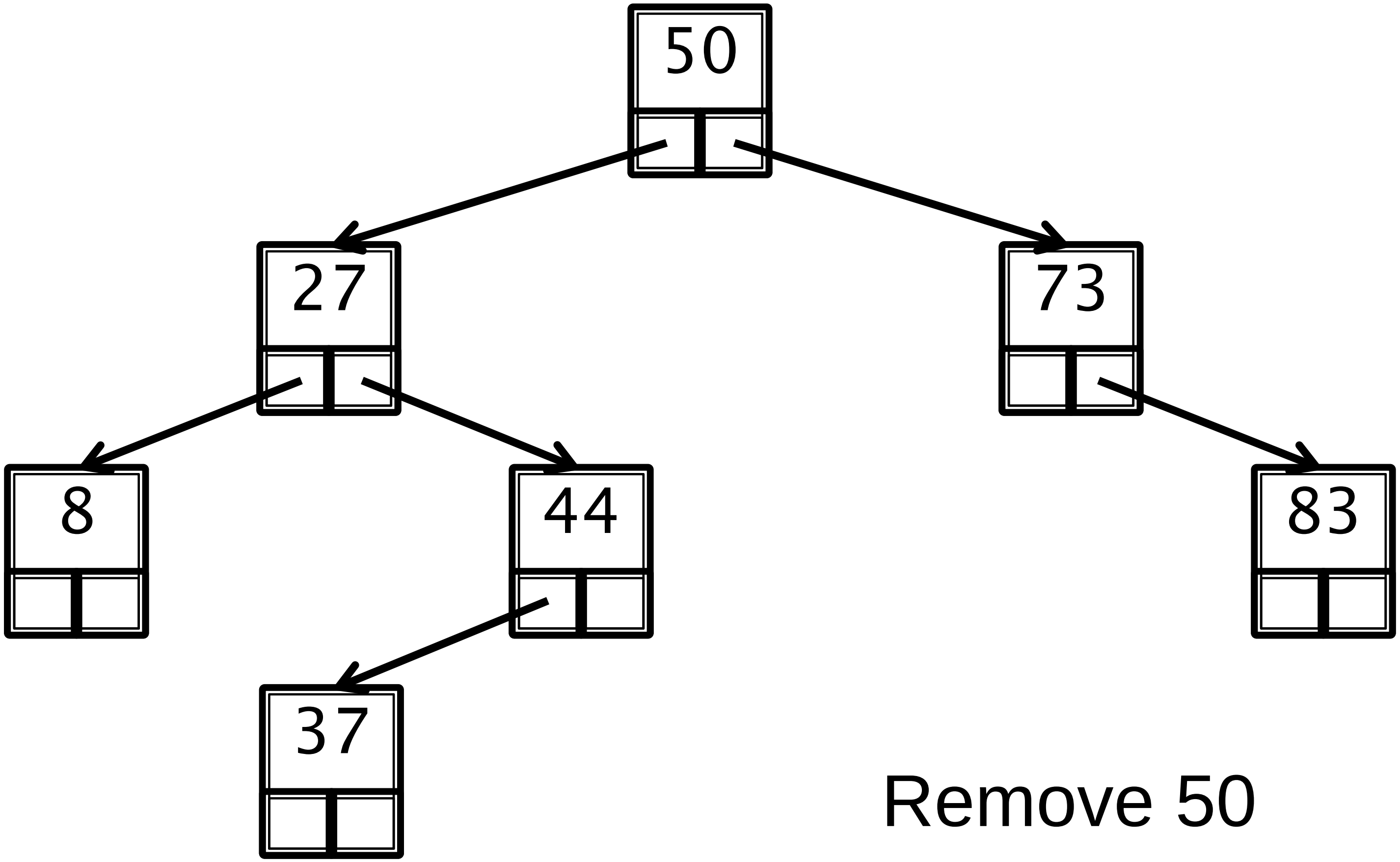



Remove 27

# Removing Nodes: Case 2

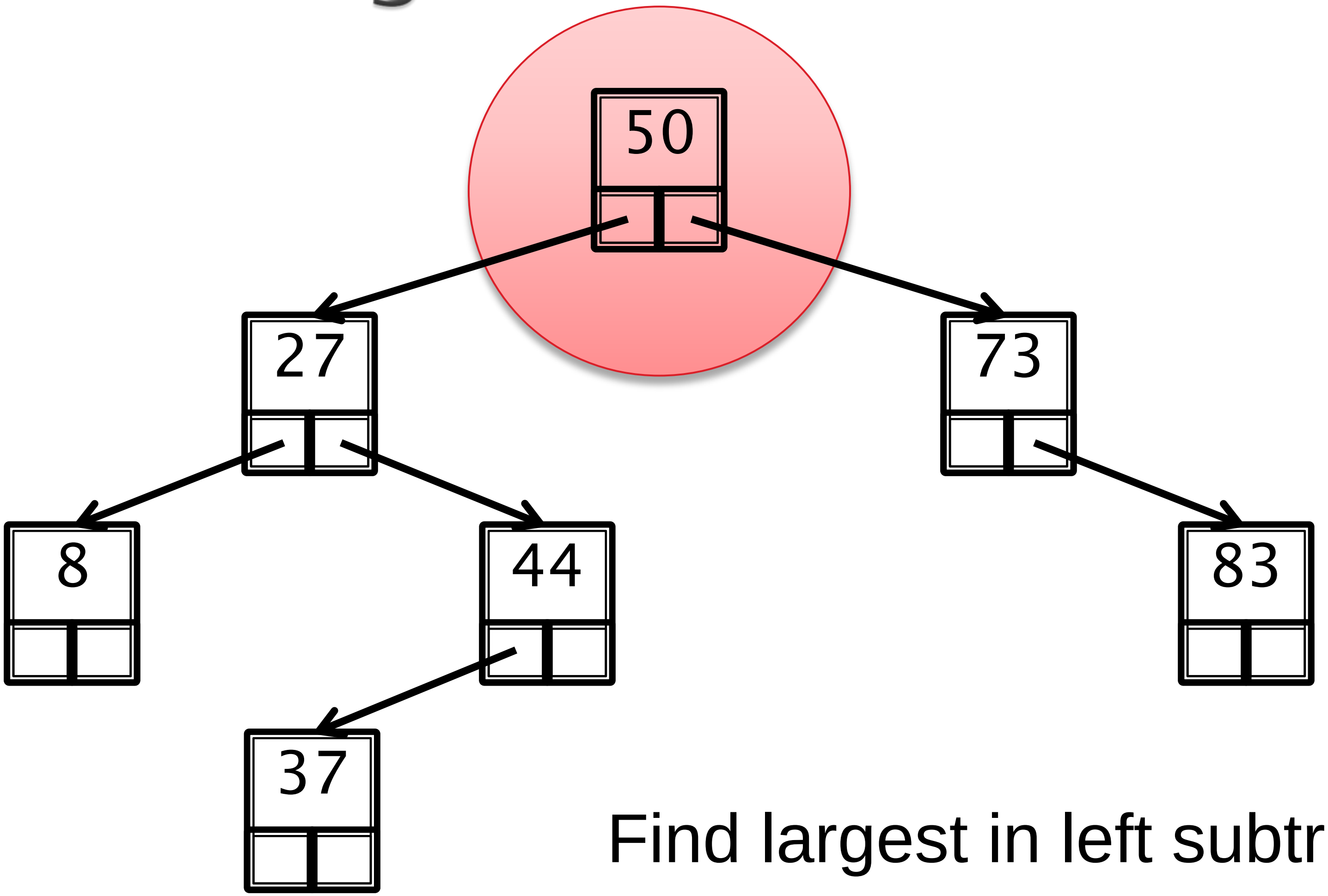


# Removing Nodes: Case 3



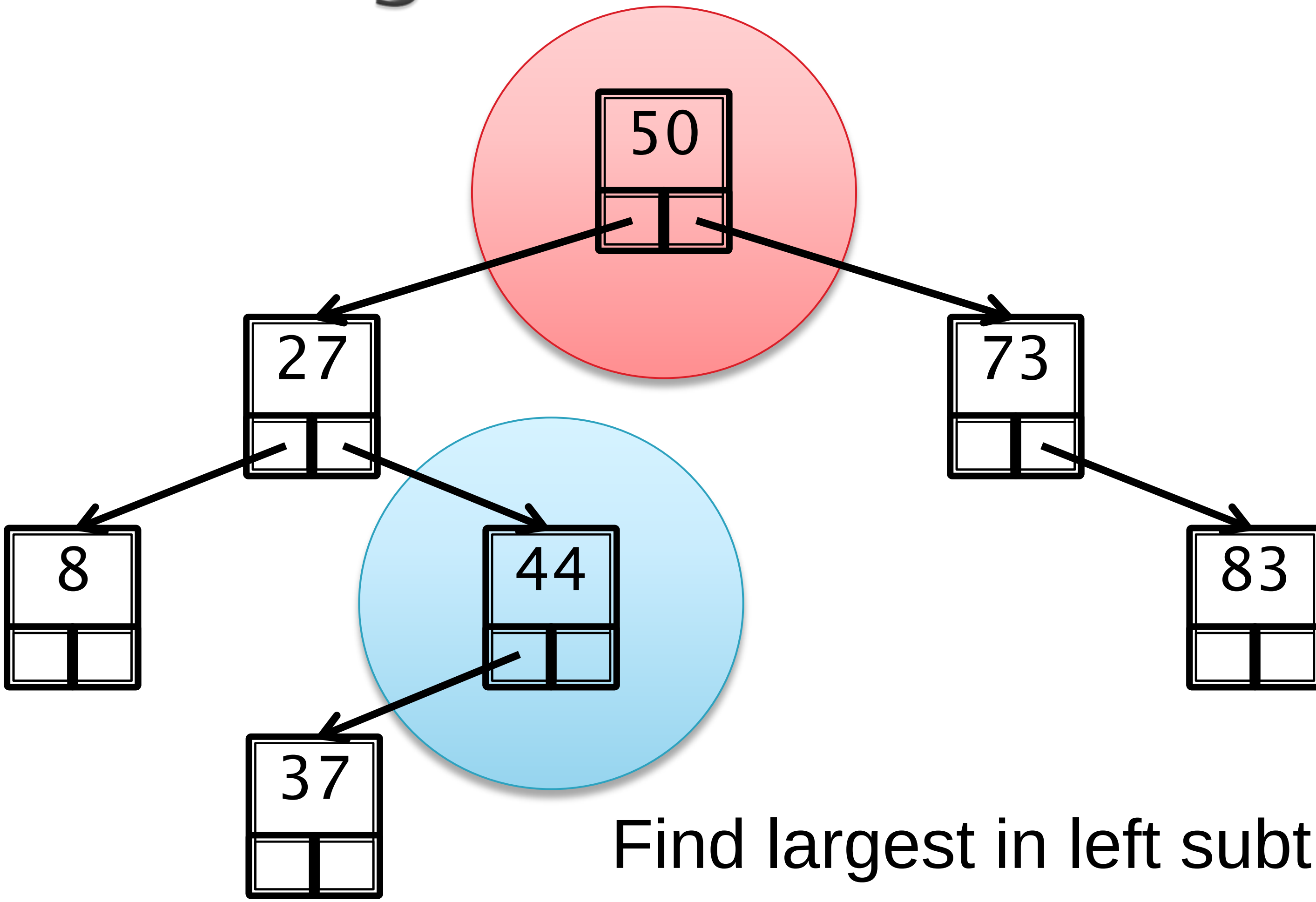

Remove 50

# Removing Nodes: Case 3



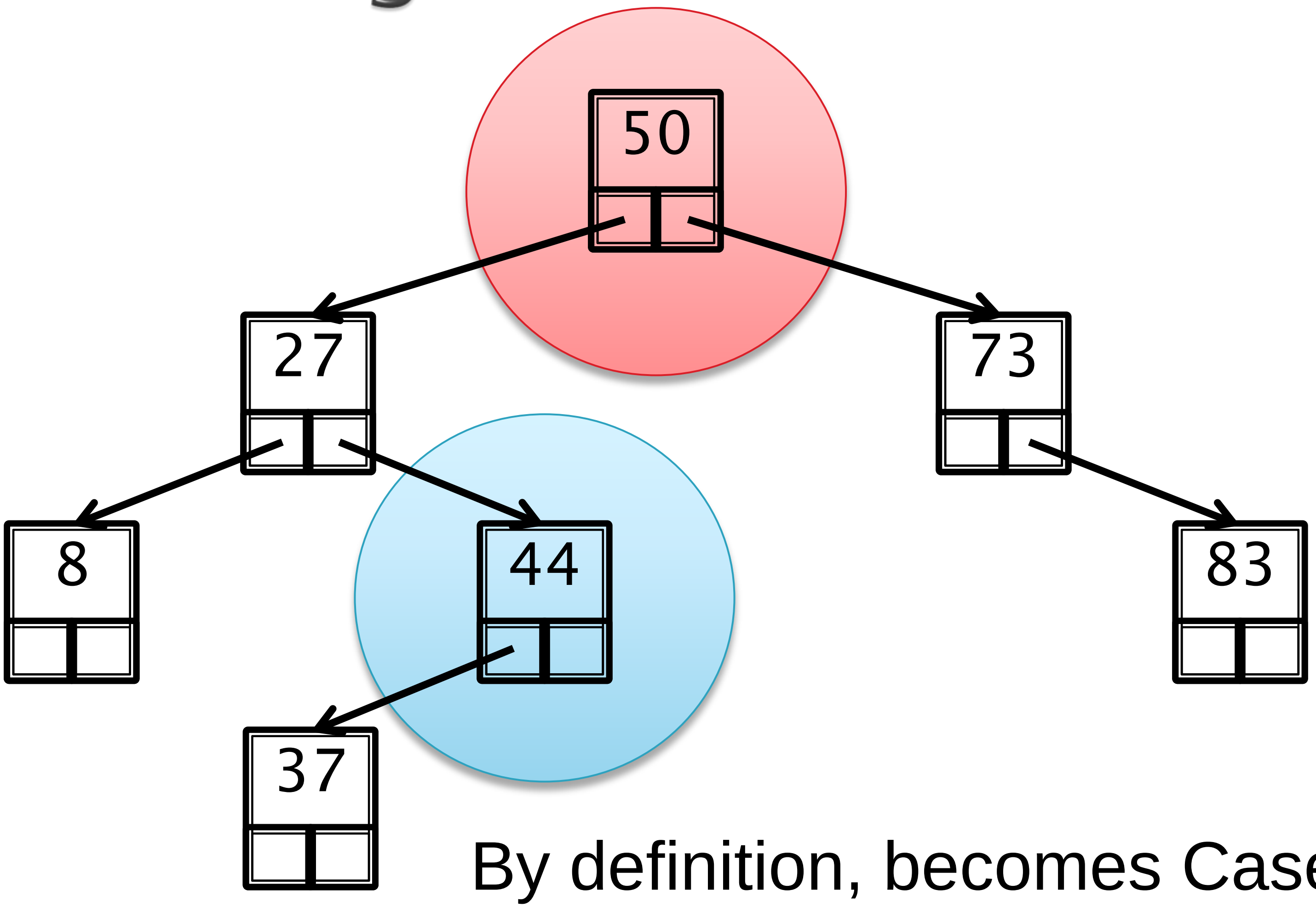

Find largest in left subtree

# Removing Nodes: Case 3



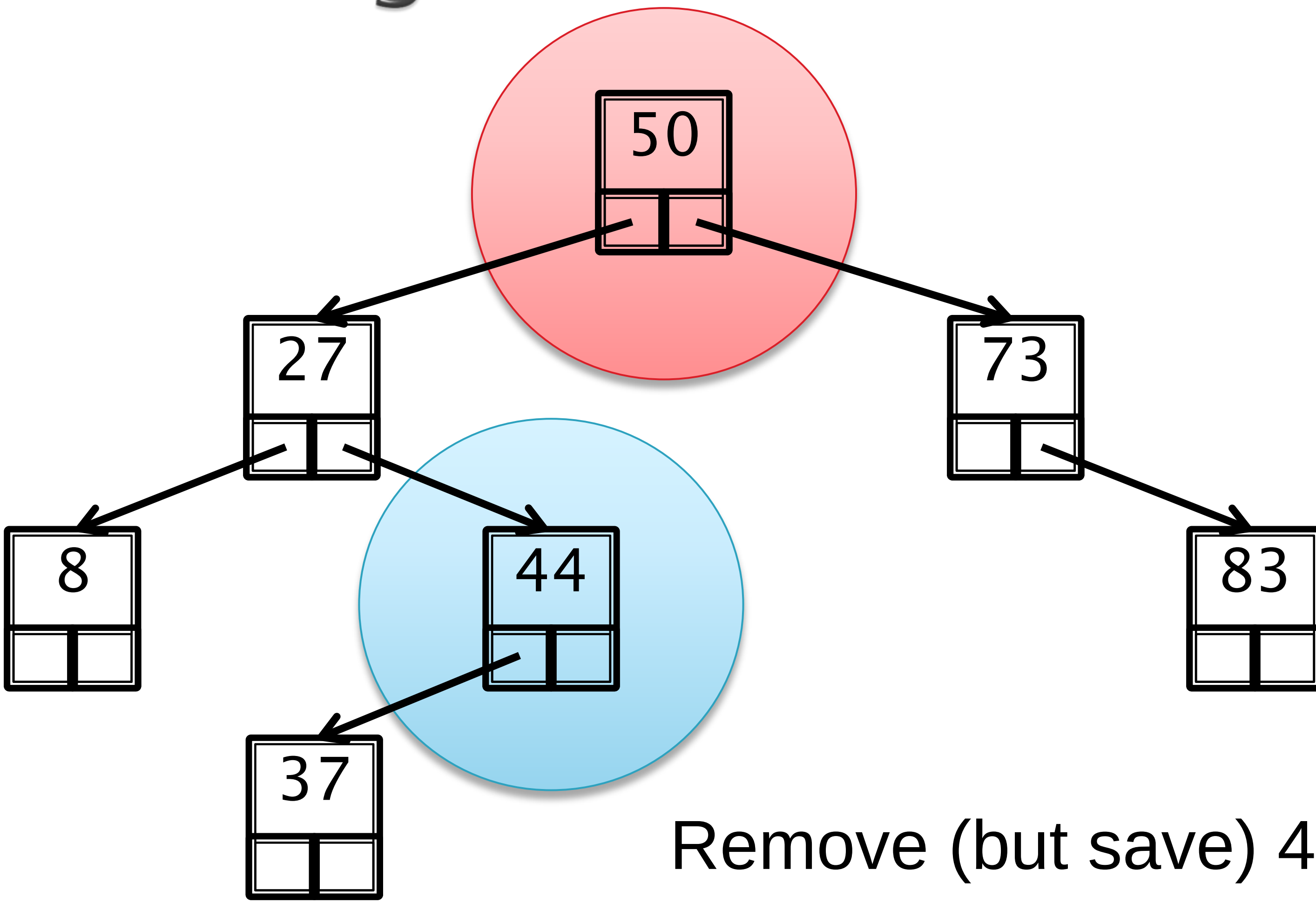

Find largest in left subtree

# Removing Nodes: Case 3



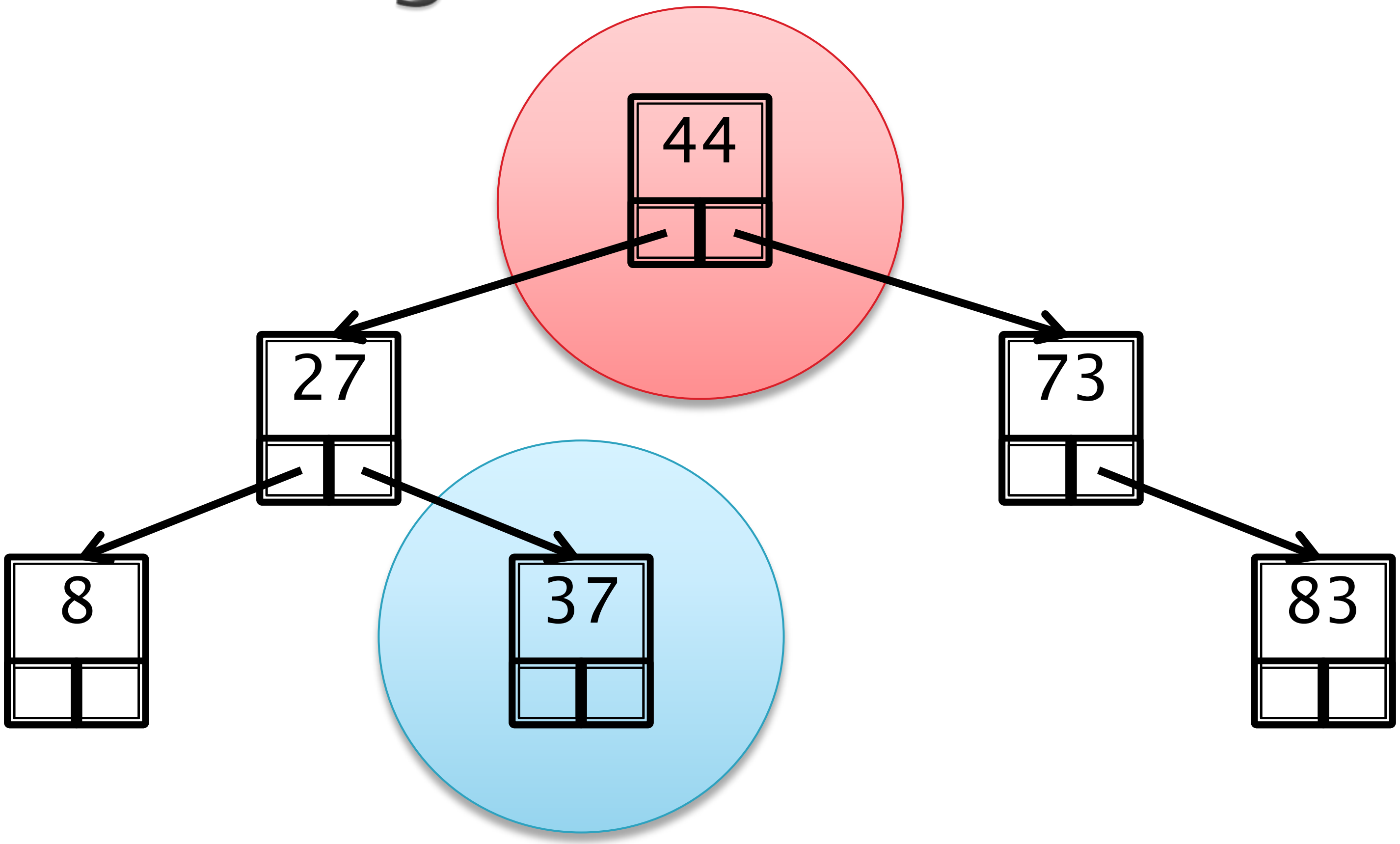

By definition, becomes Case 1 or 2

# Removing Nodes: Case 3




Remove (but save) 44

# Removing Nodes: Case 3




50 is removed from tree

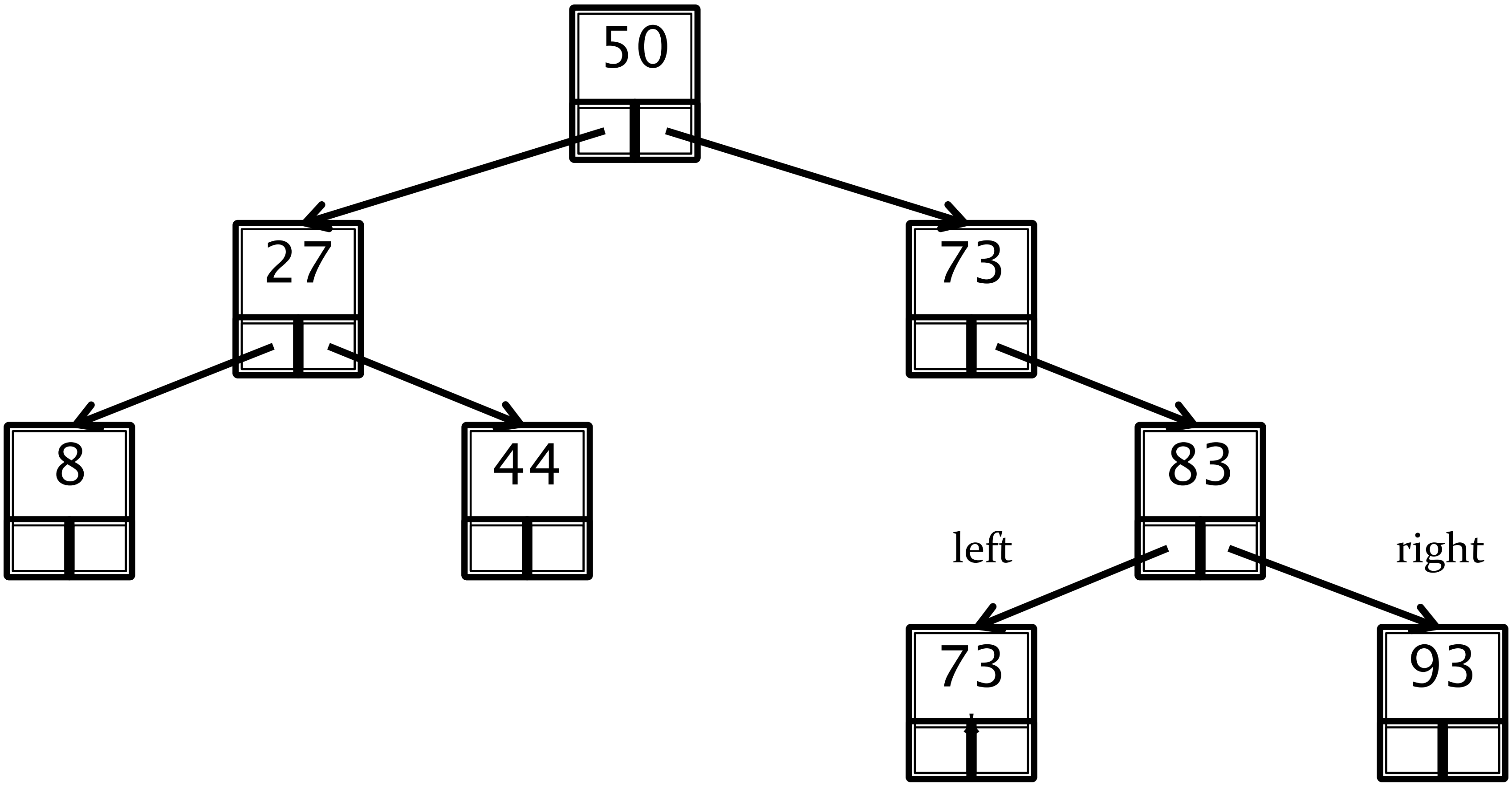
# Binary Tree Algorithms

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

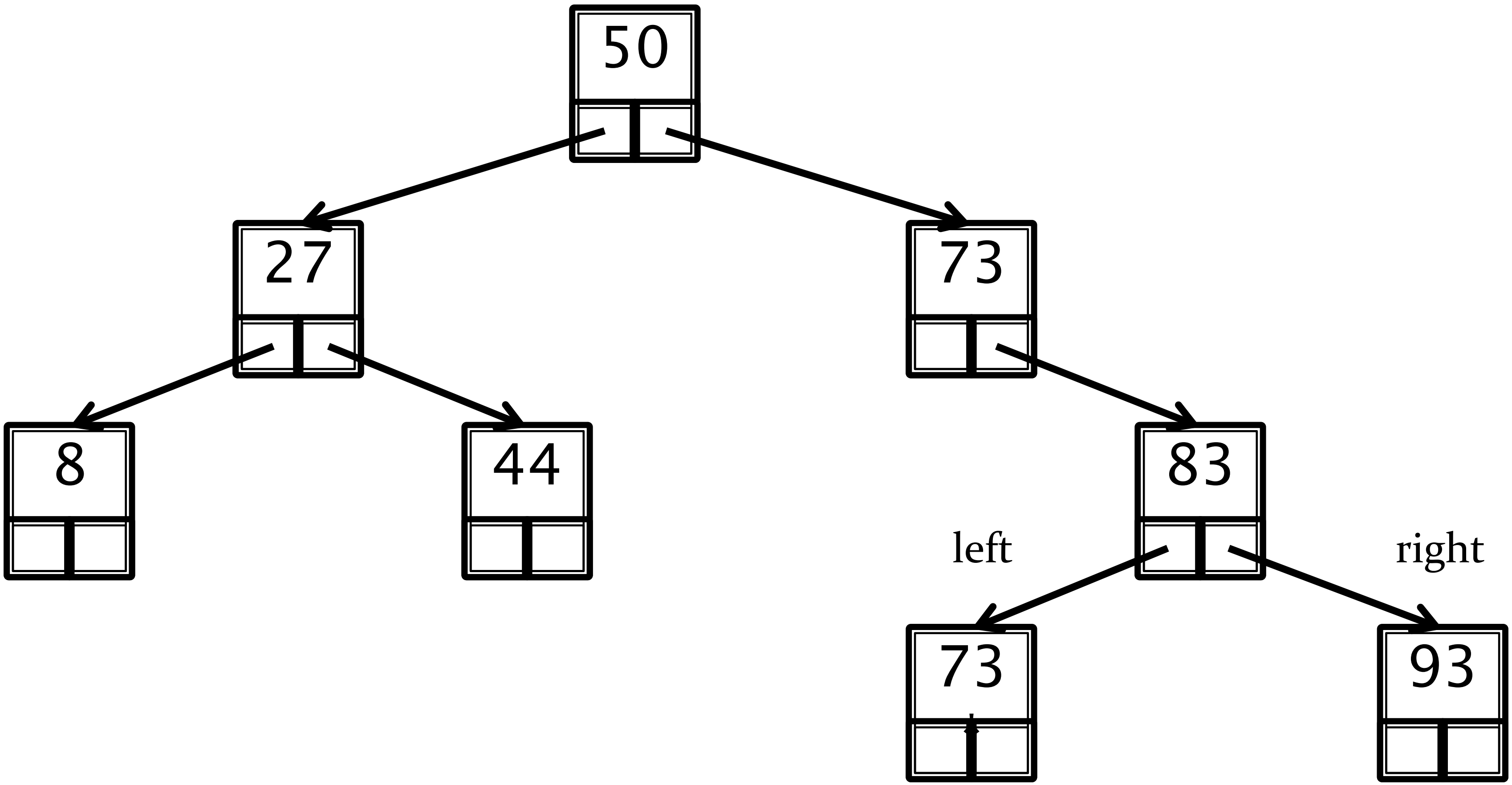
```
public <E> boolean contains(E e, Node<E> node)
 {
        Base cases:
         - node is null
         - node's data is equal to e

        Recursive cases:
         - if e < node's data, search left subtree
         - if e > node's data, search right subtree
}
```

# Iteration

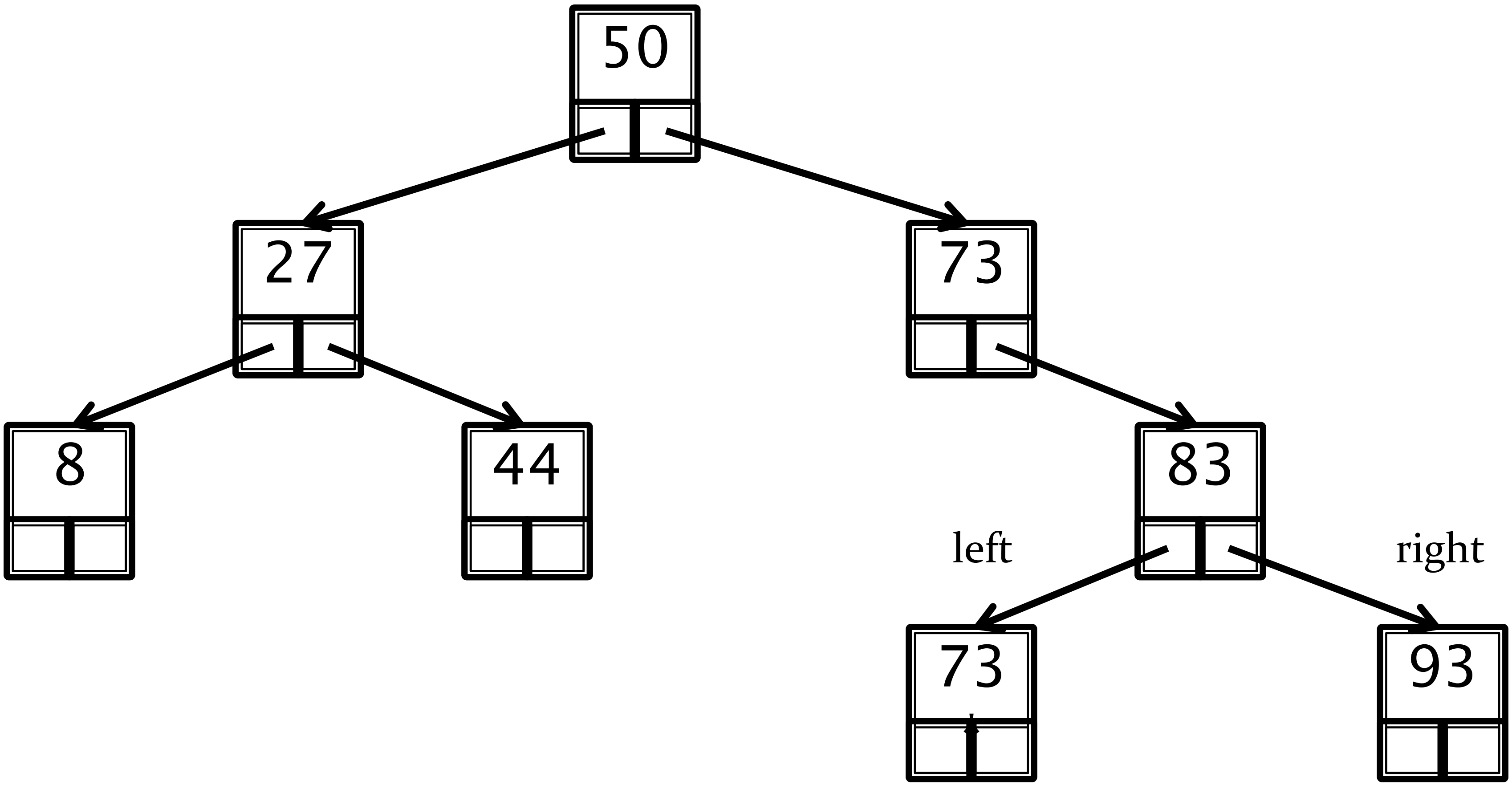
- Visiting every element of the tree can also be done recursively
- 3 possibilities based on when the root is visited
  - Inorder
    - Visit left child, then root, then right child
  - Preorder
    - Visit root, then left child, then right child
  - Postorder
    - Visit left child, then right child,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




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



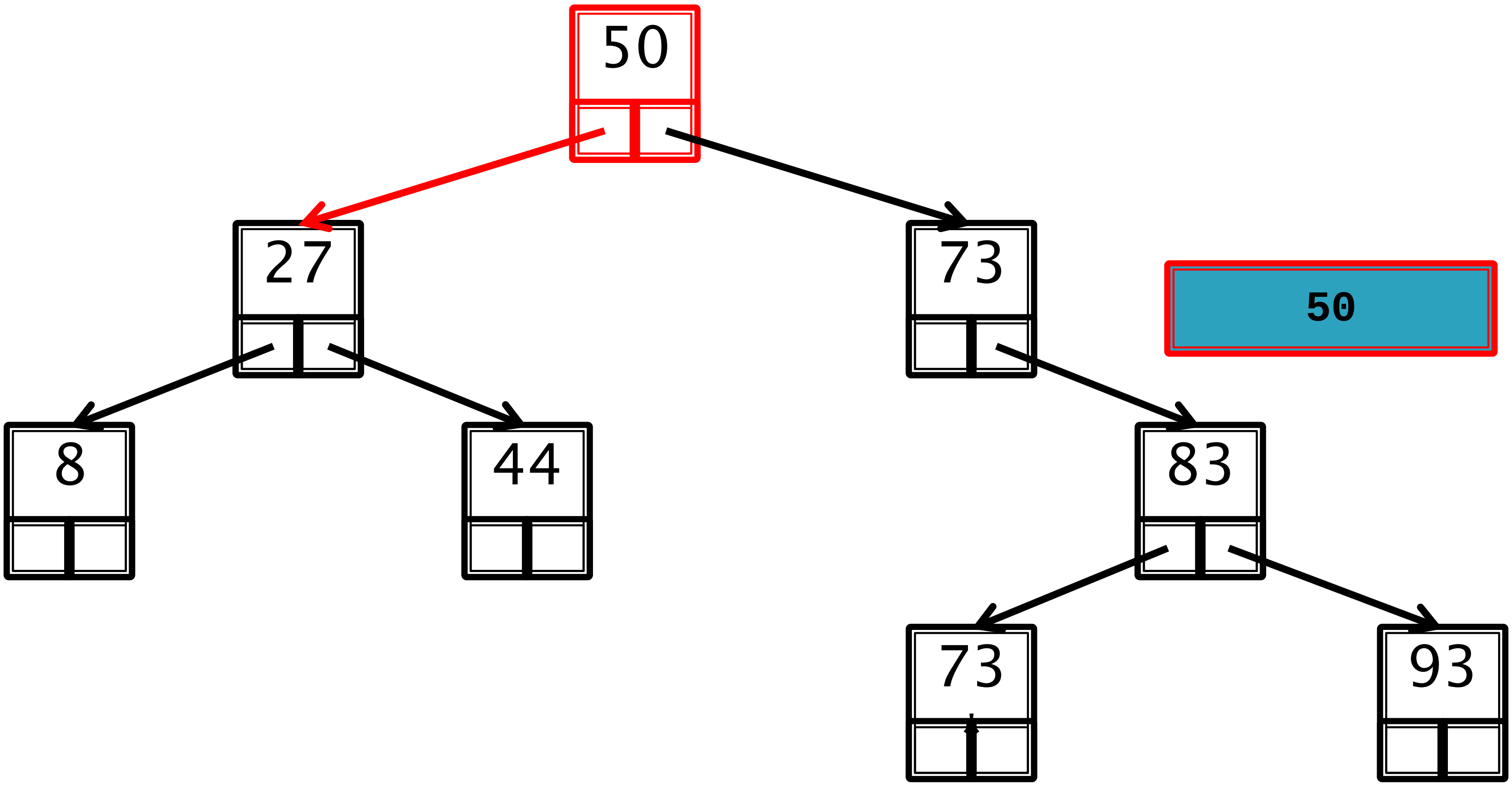

Preorder: 50, 27, 8, 44, 73, 83, 73*, 93



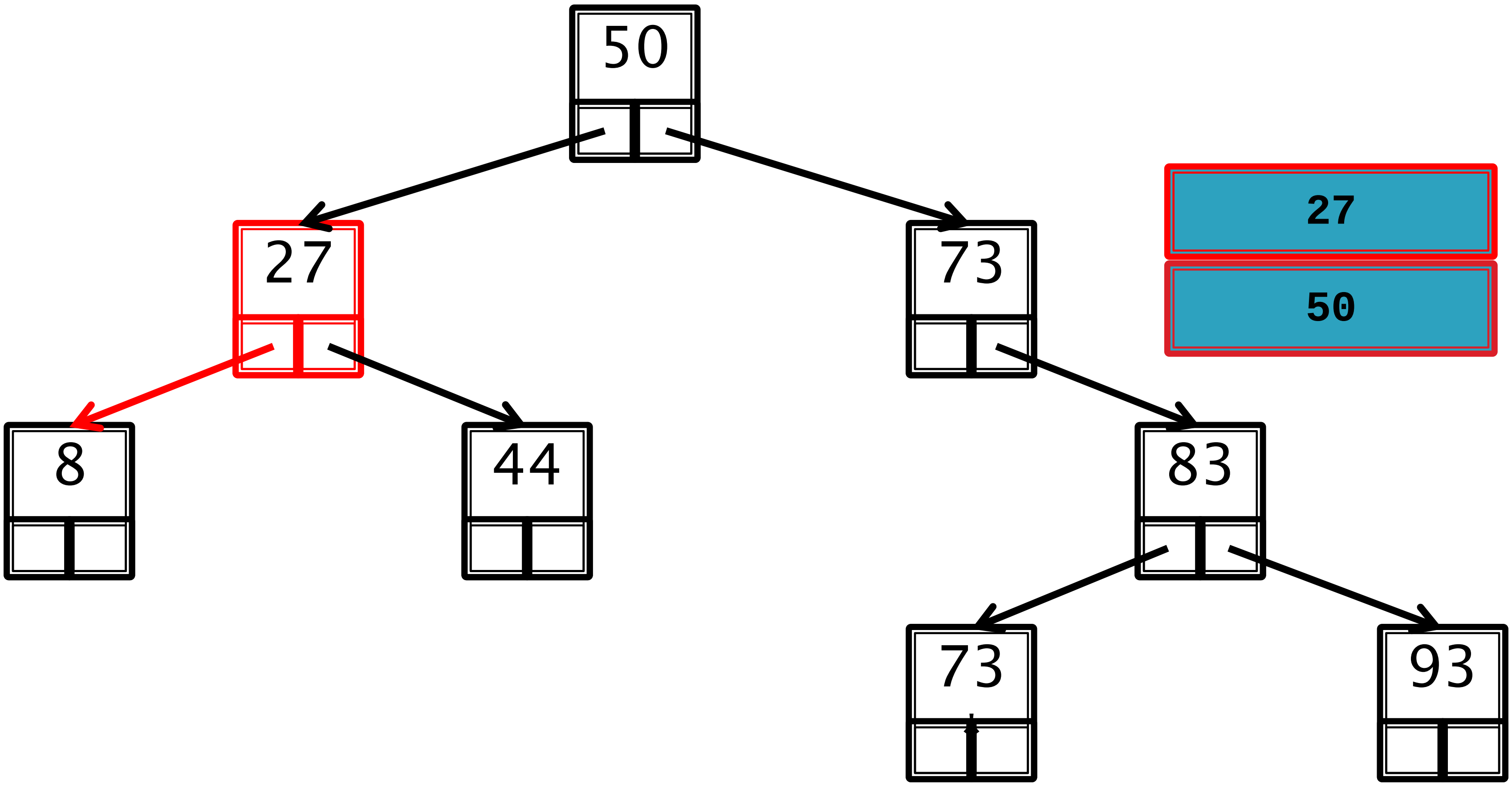

Postorder: 8, 44, 27, 73*, 93, 83, 73, 50

# Example: Tree traversal

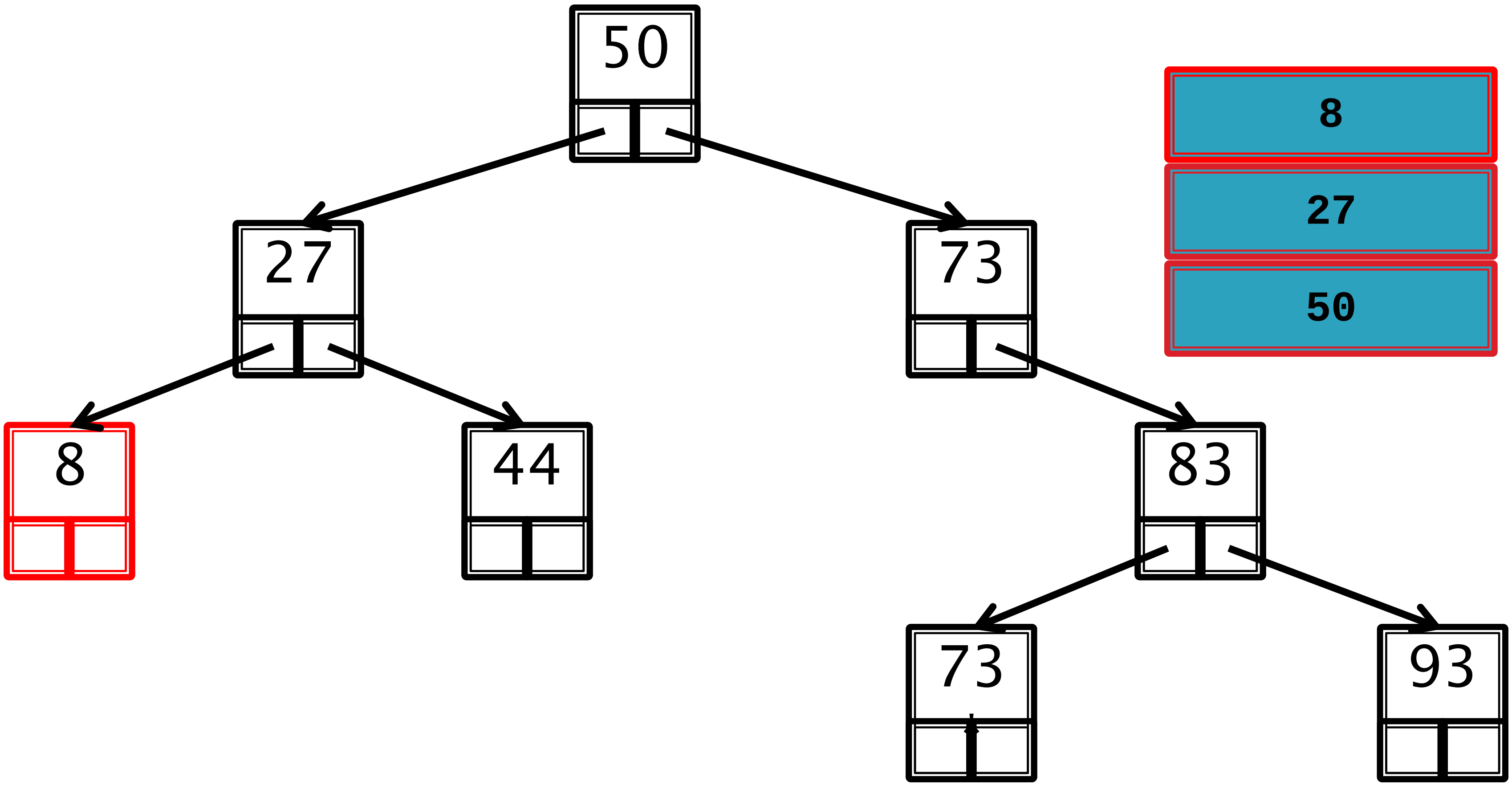
- A stack can be used in place of recursion for visiting all of the nodes of a tree
  - Basic idea is to push nodes onto the stack as you traverse the tree
  - Pushing the node onto the stack allows you to remember that you have to visit the other branch of the tree rooted at the node



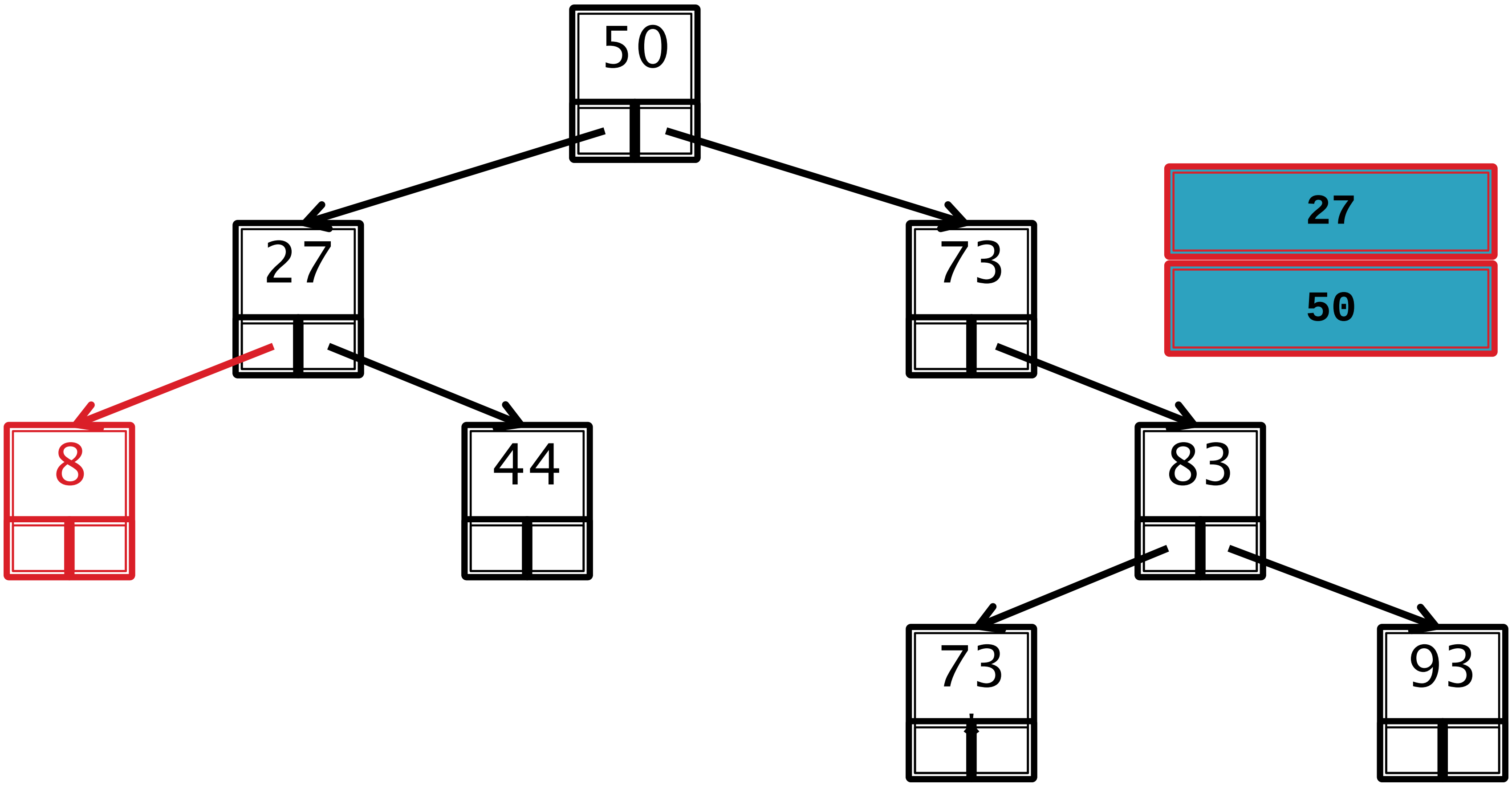

Inorder: 8, 27, 44, 50, 73, 73*, 83, 93



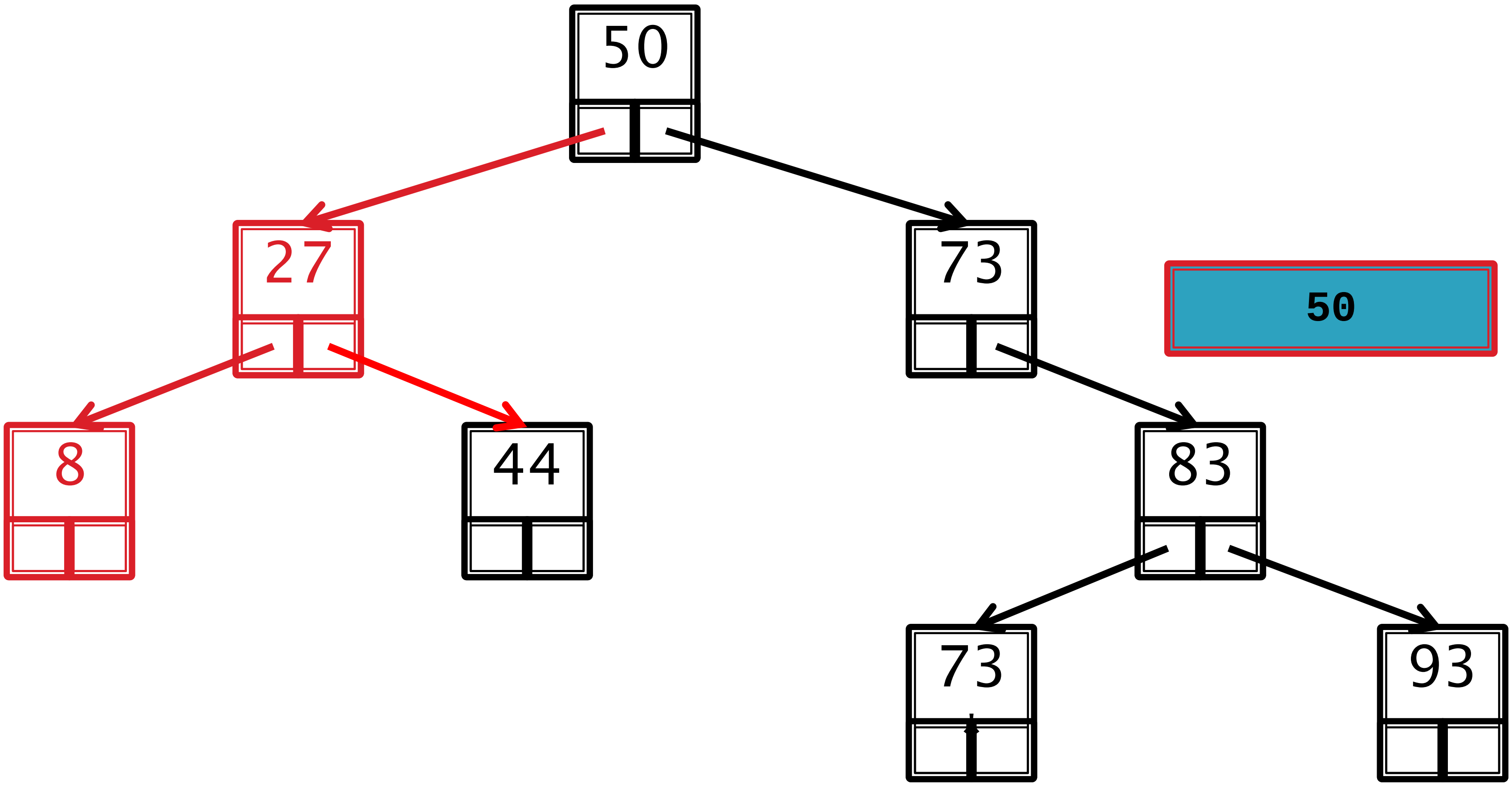

Inorder: 8, 27, 44, 50, 73, 73*, 83, 93



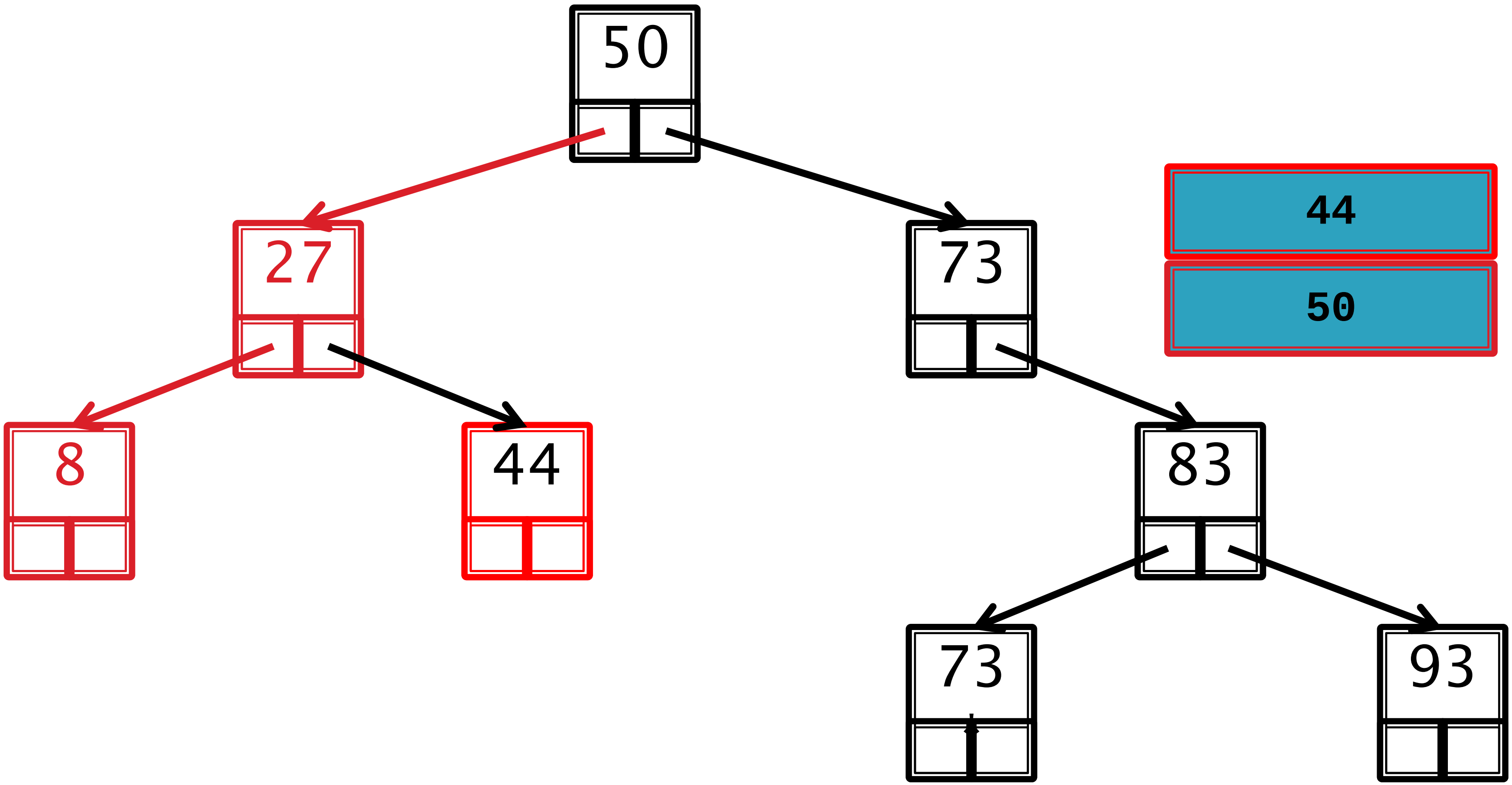

Inorder: 8, 27, 44, 50, 73, 73*, 83, 93



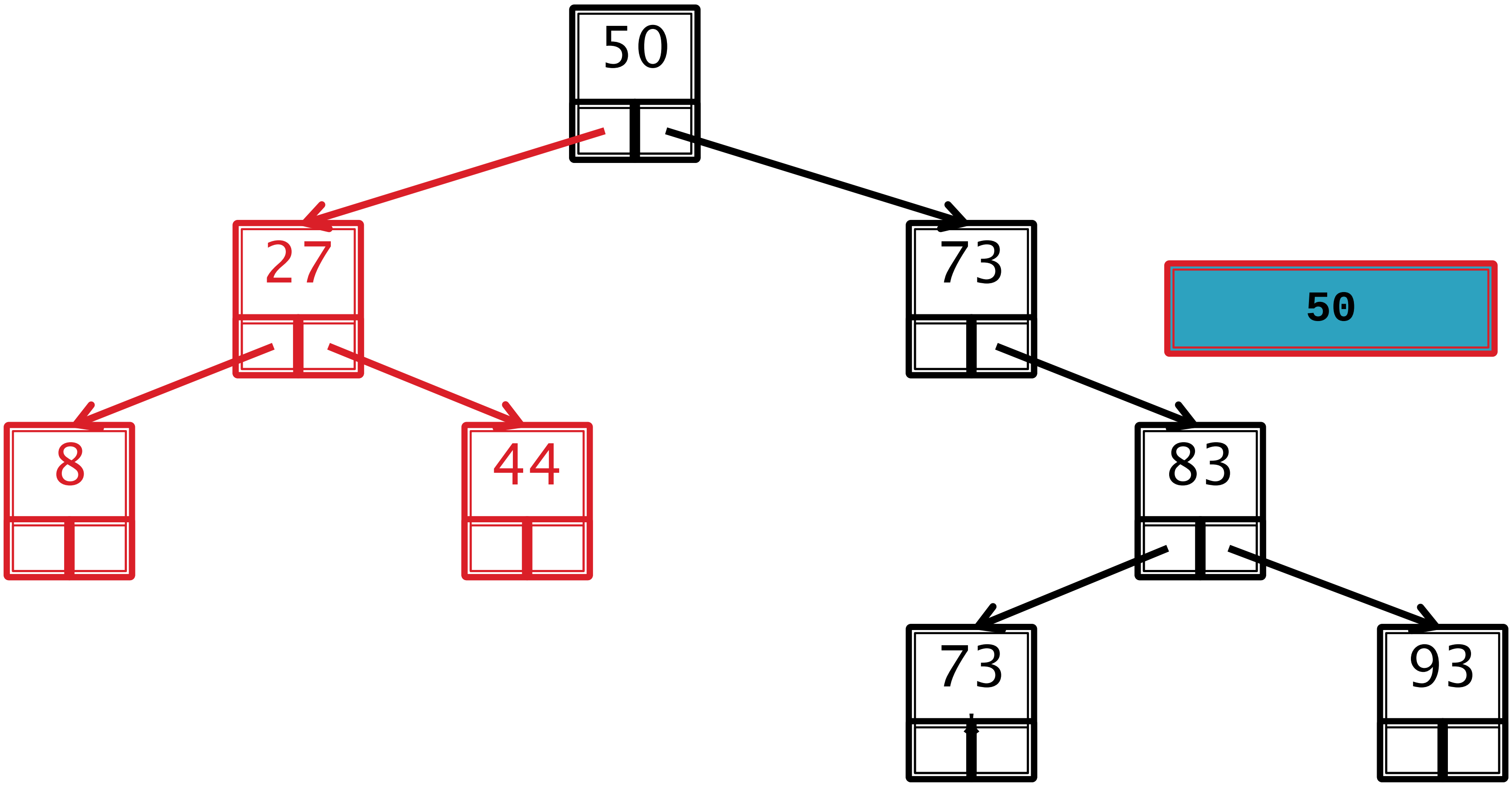

Inorder: 8, 27, 44, 50, 73, 73*, 83, 93



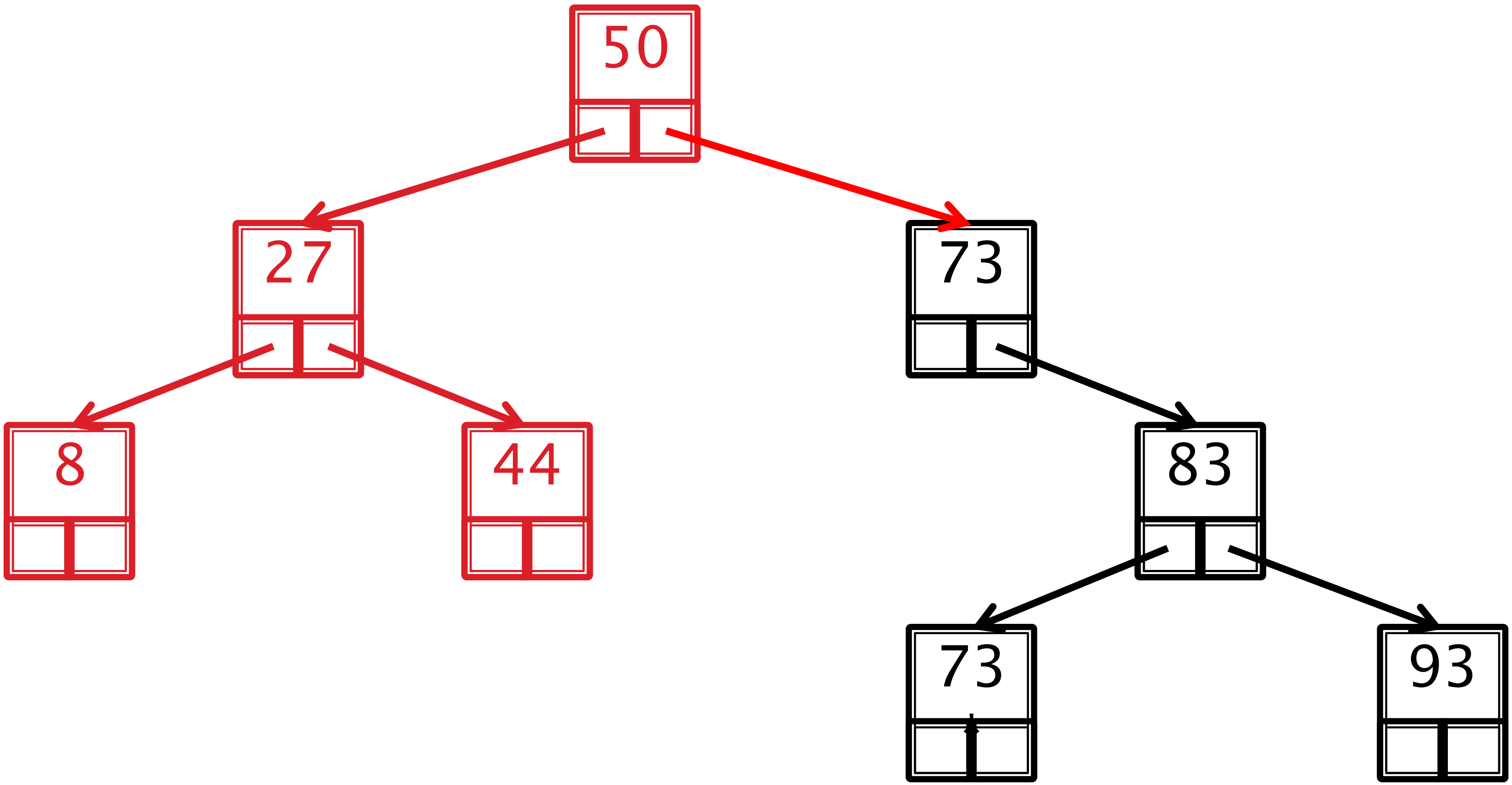

Inorder: 8, 27, 44, 50, 73, 73*, 83, 93



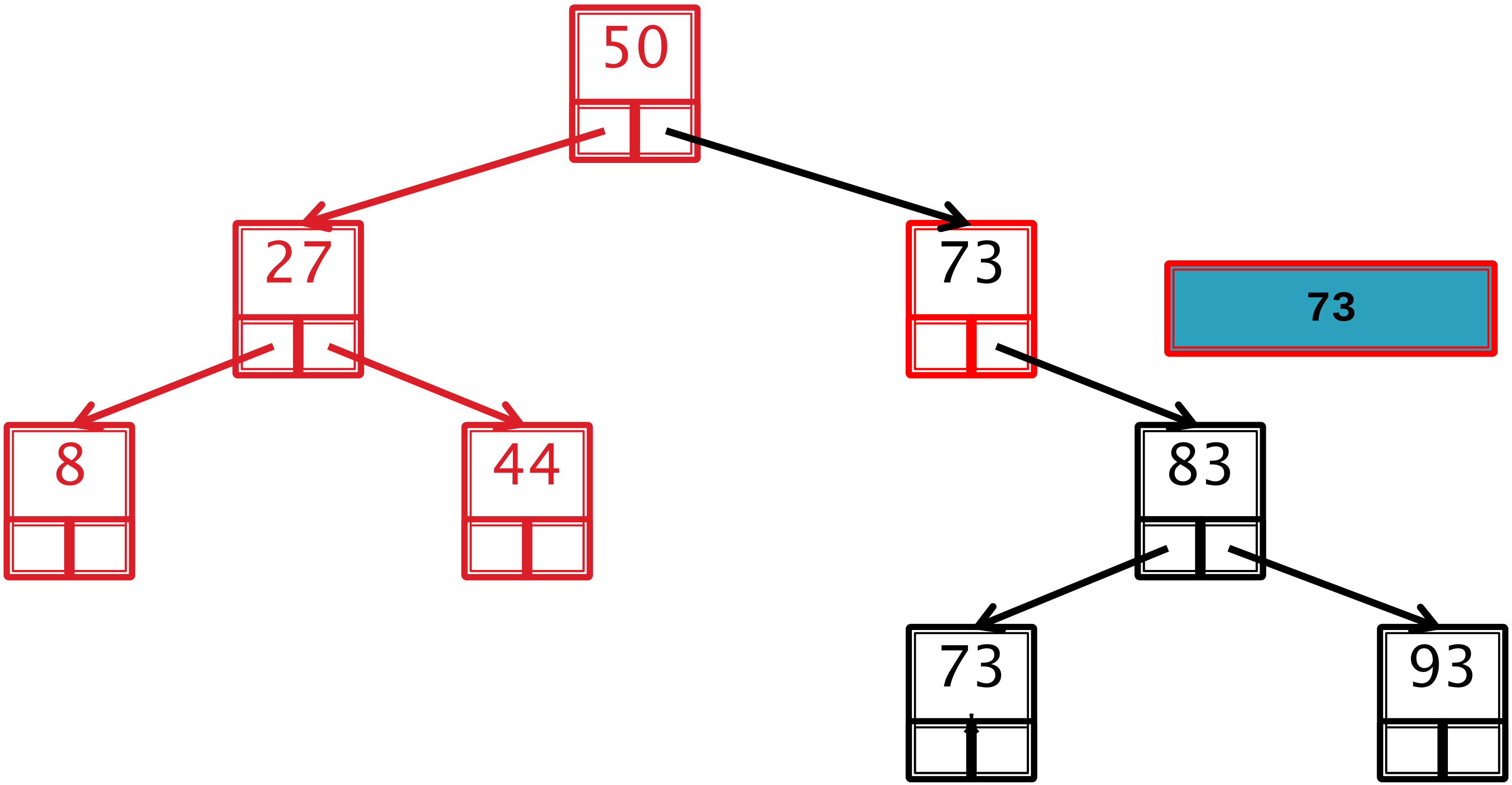

Inorder: 8, 27, 44, 50, 73, 73*, 83, 93



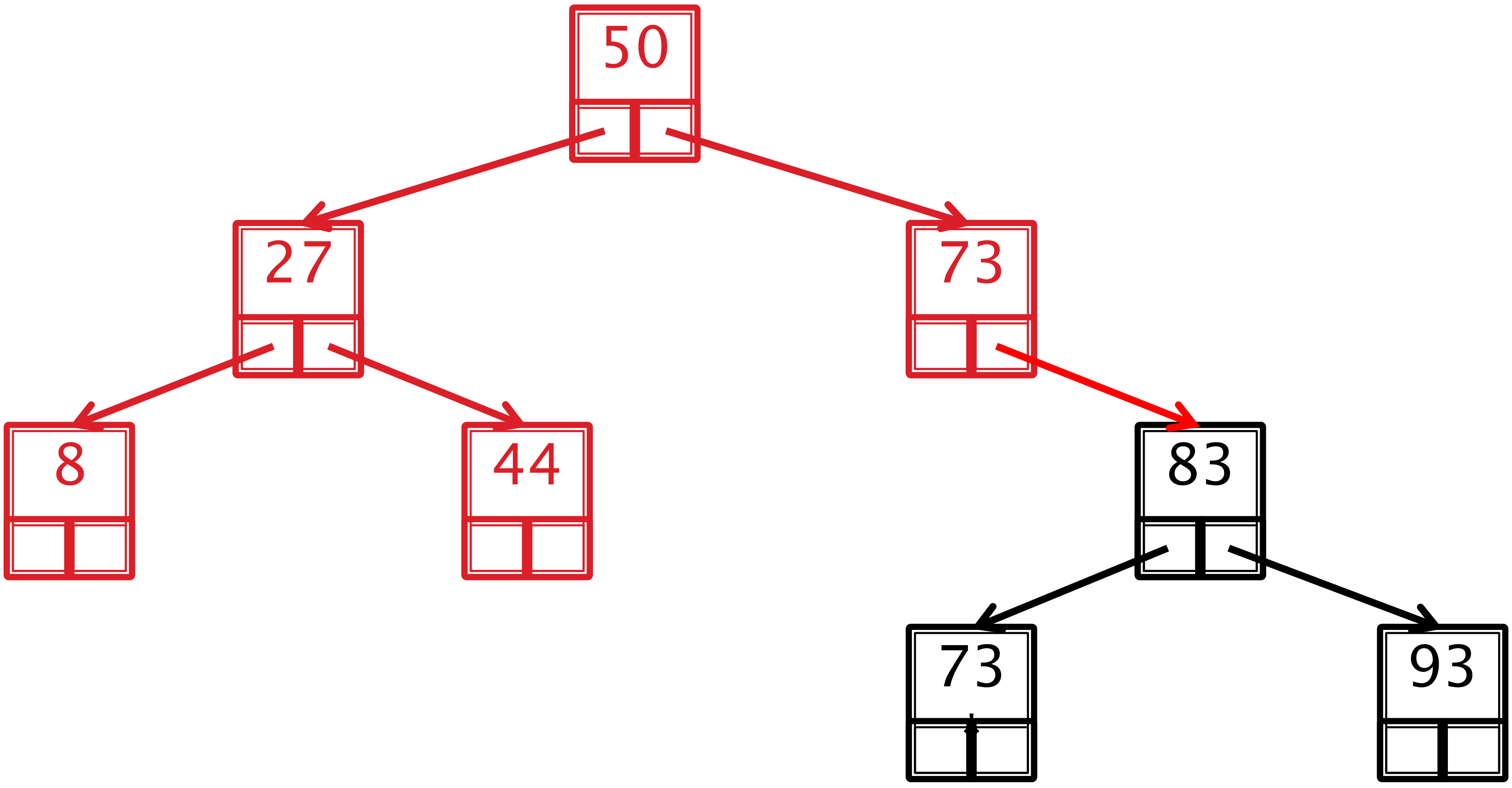

Inorder: 8, 27, 44, 50, 73, 73*, 83, 93



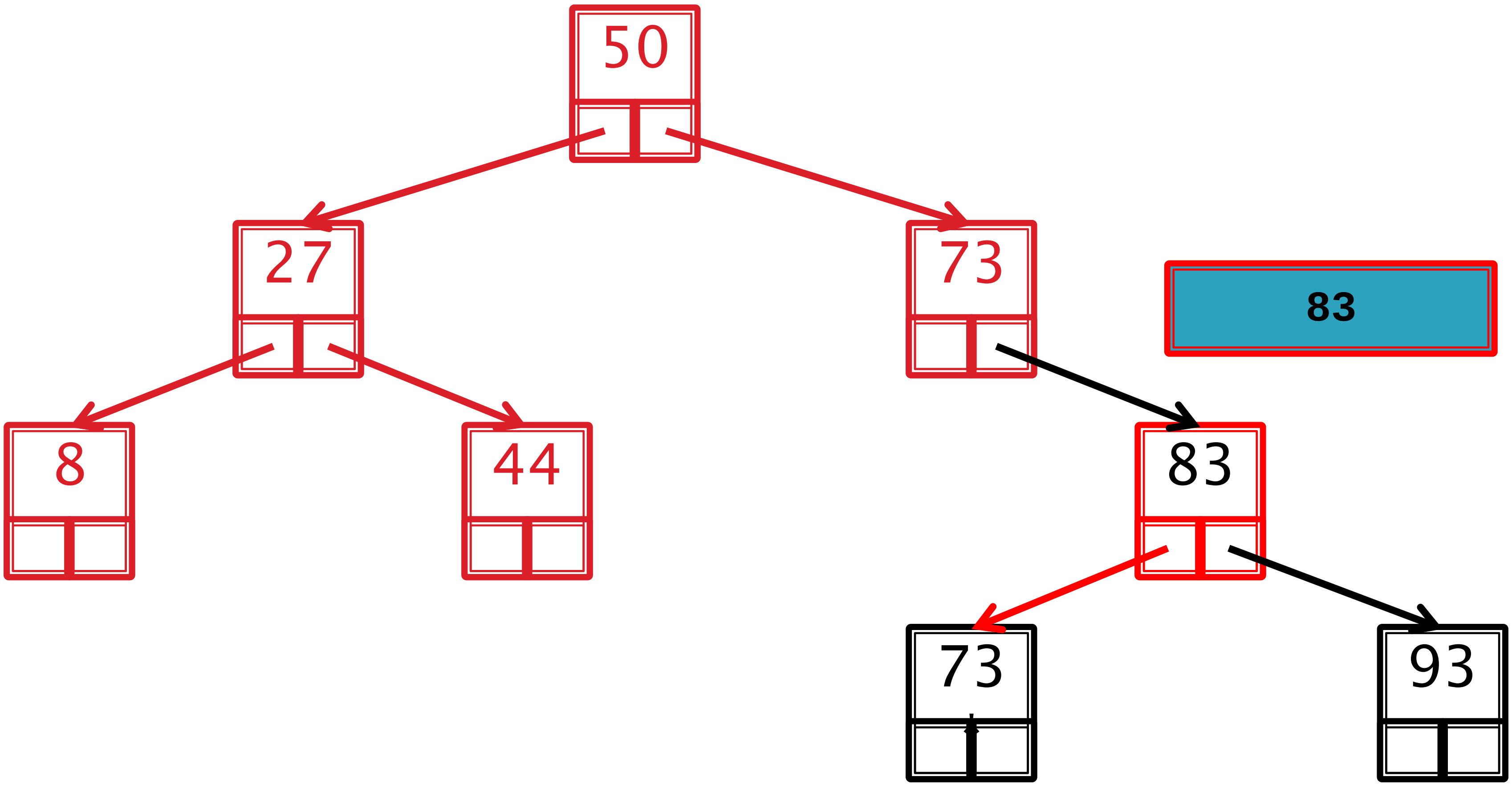

Inorder: 8, 27, 44, 50, 73, 73*, 83, 93



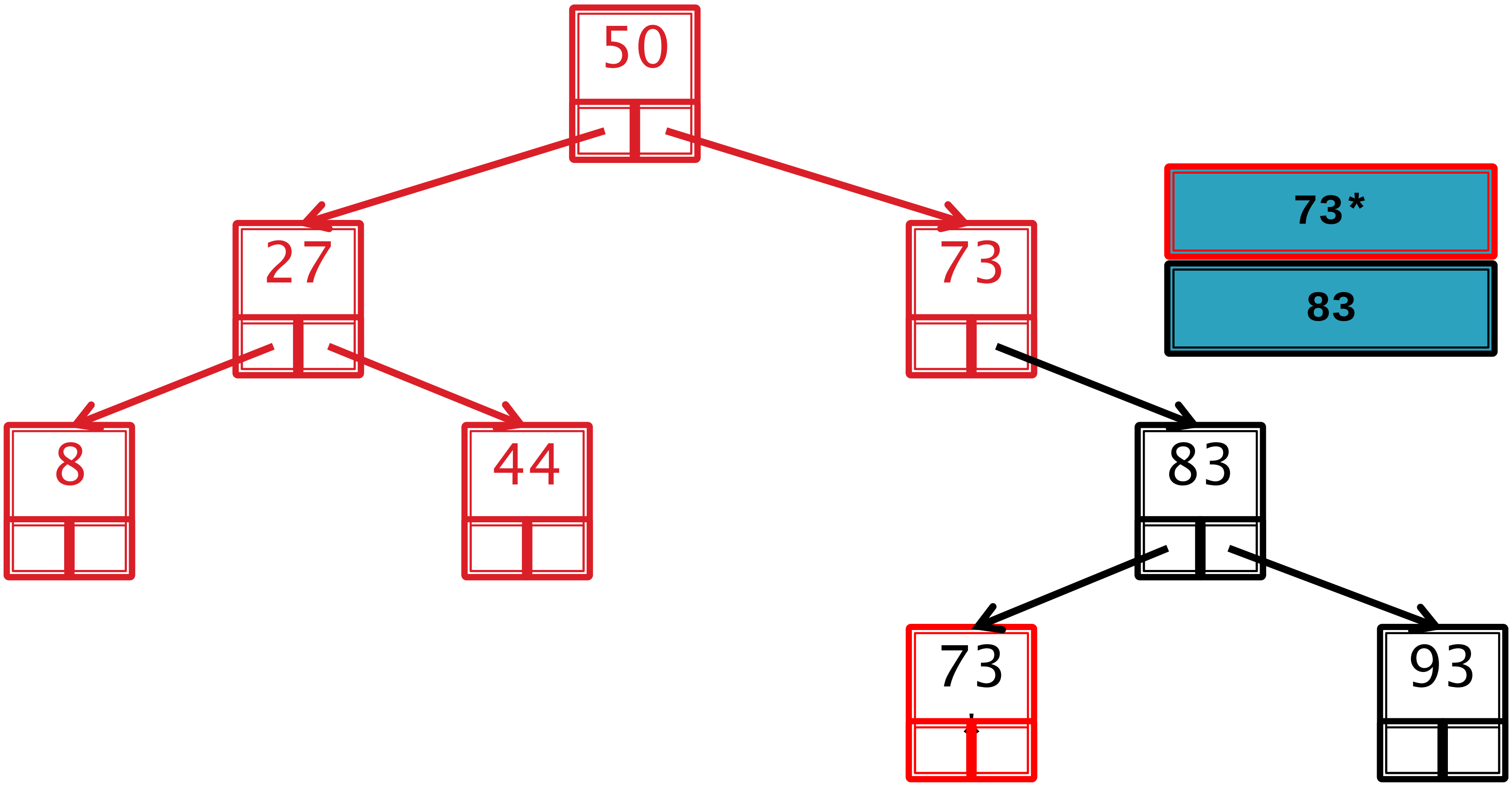

Inorder: 8, 27, 44, 50, 73, 73*, 83, 93



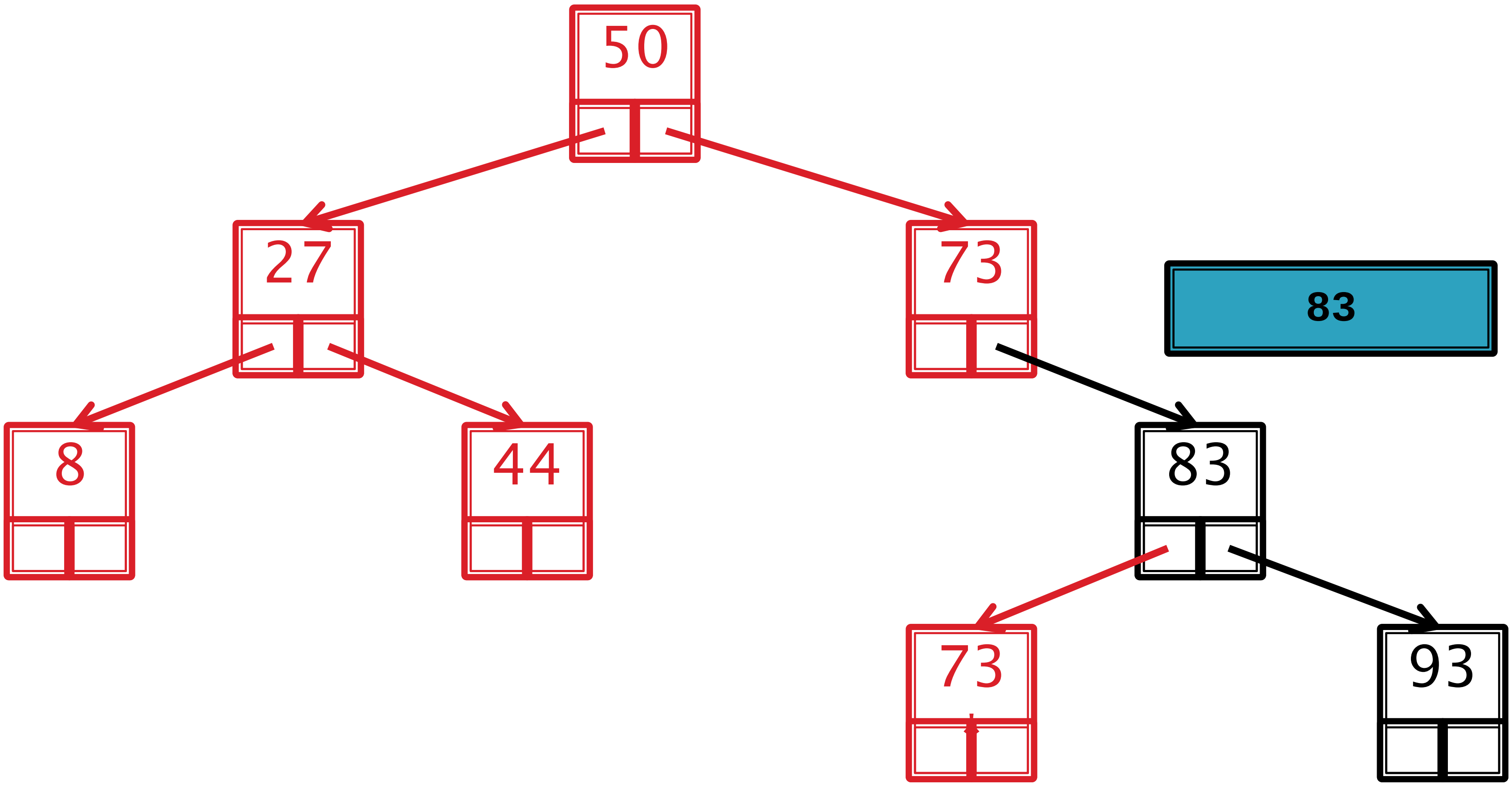

Inorder: 8, 27, 44, 50, 73, 73*, 83, 93



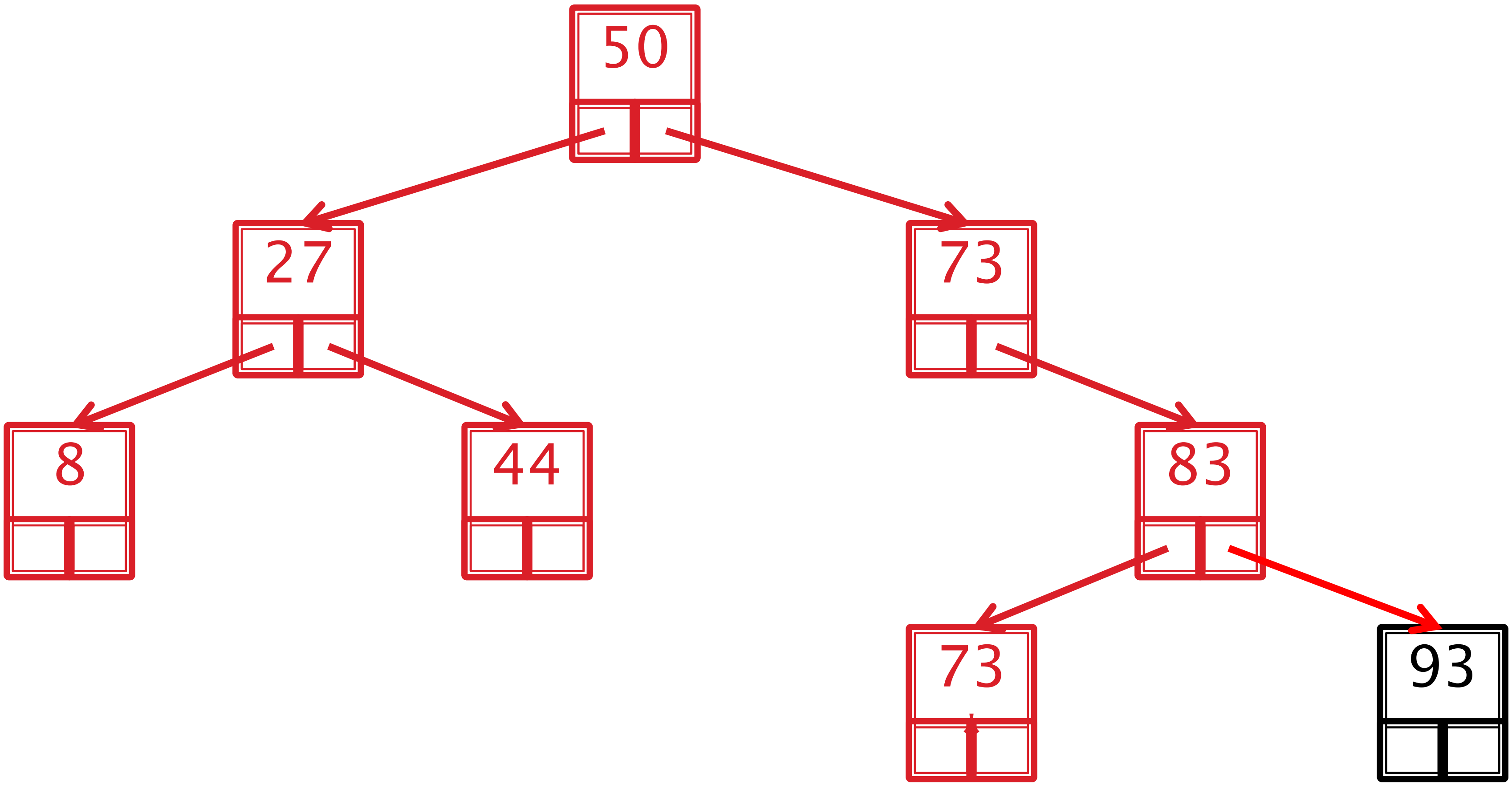

Inorder: 8, 27, 44, 50, 73, 73*, 83, 93



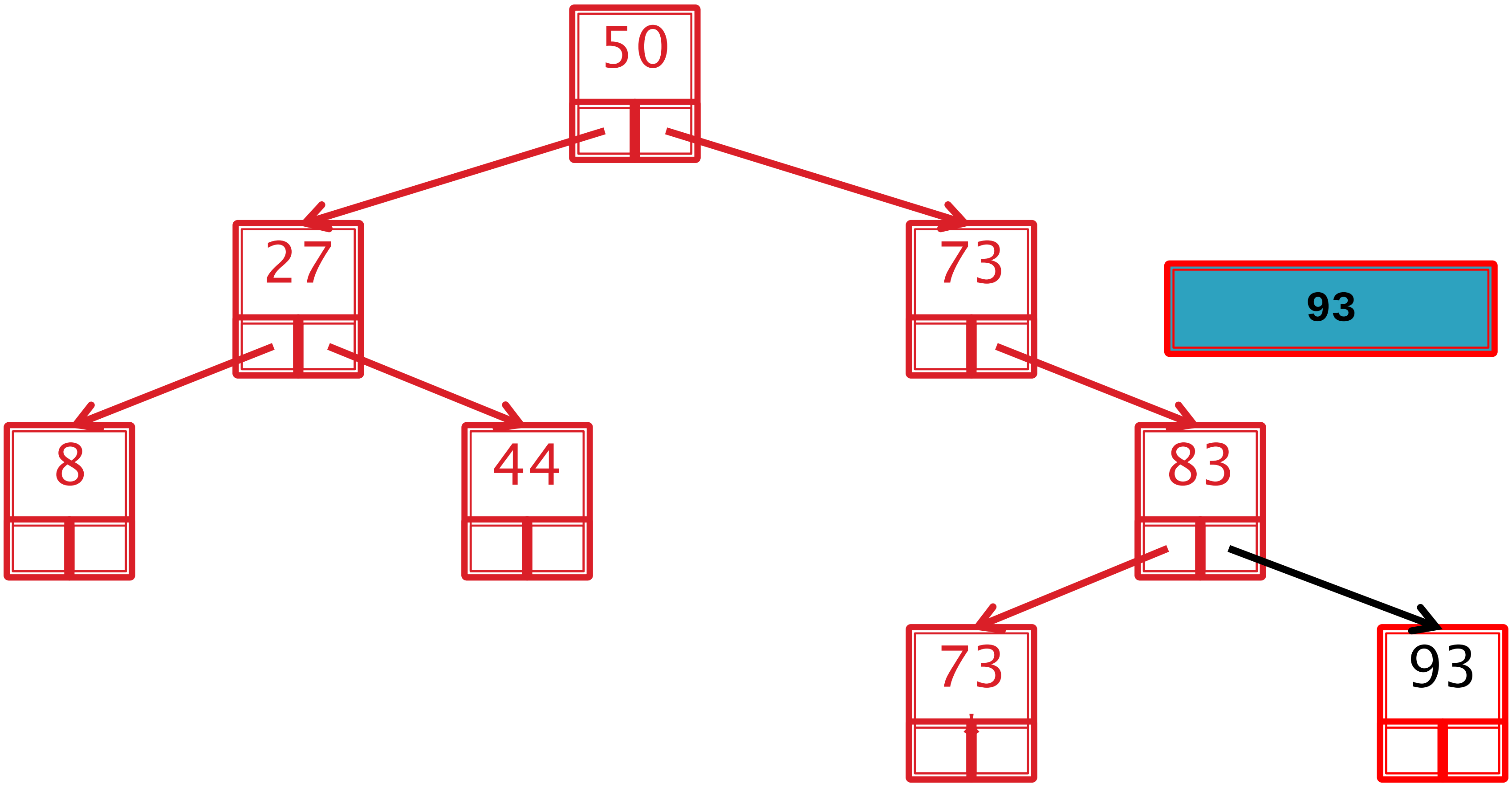

Inorder: 8, 27, 44, 50, 73, 73*, 83, 93



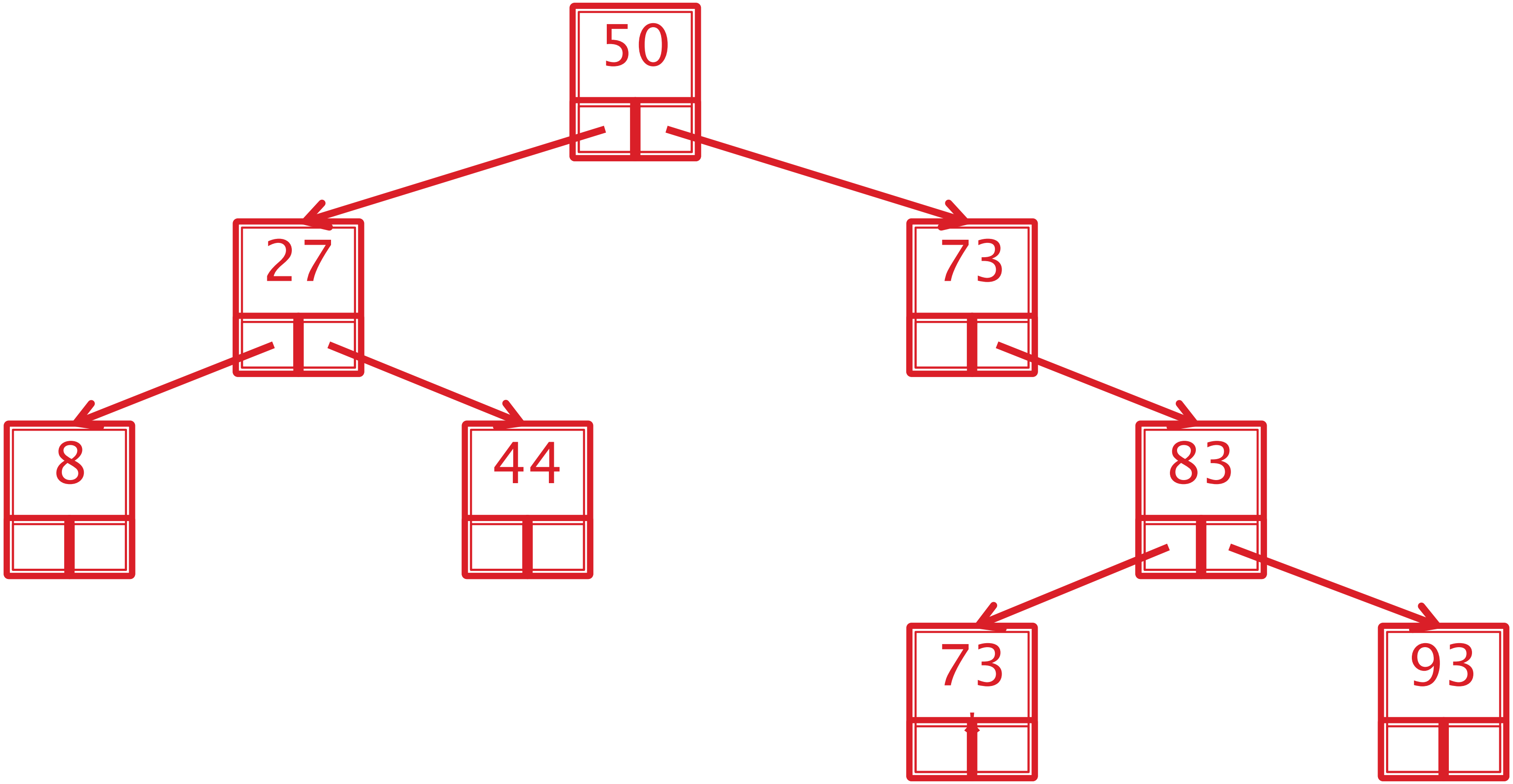

Inorder: 8, 27, 44, 50, 73, 73*, 83, 93




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



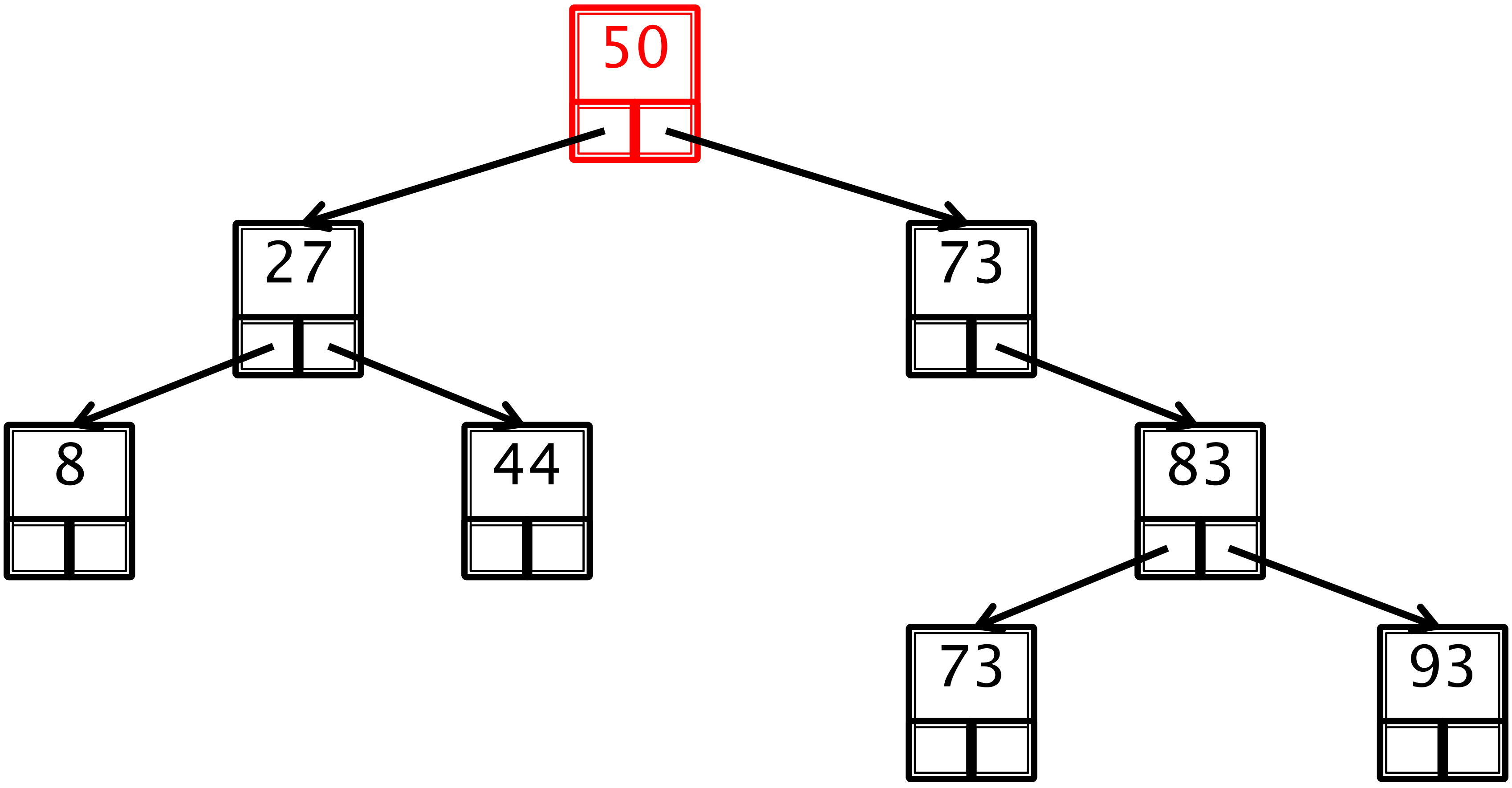

Inorder: 8, 27, 44, 50, 73, 73*, 83, 93



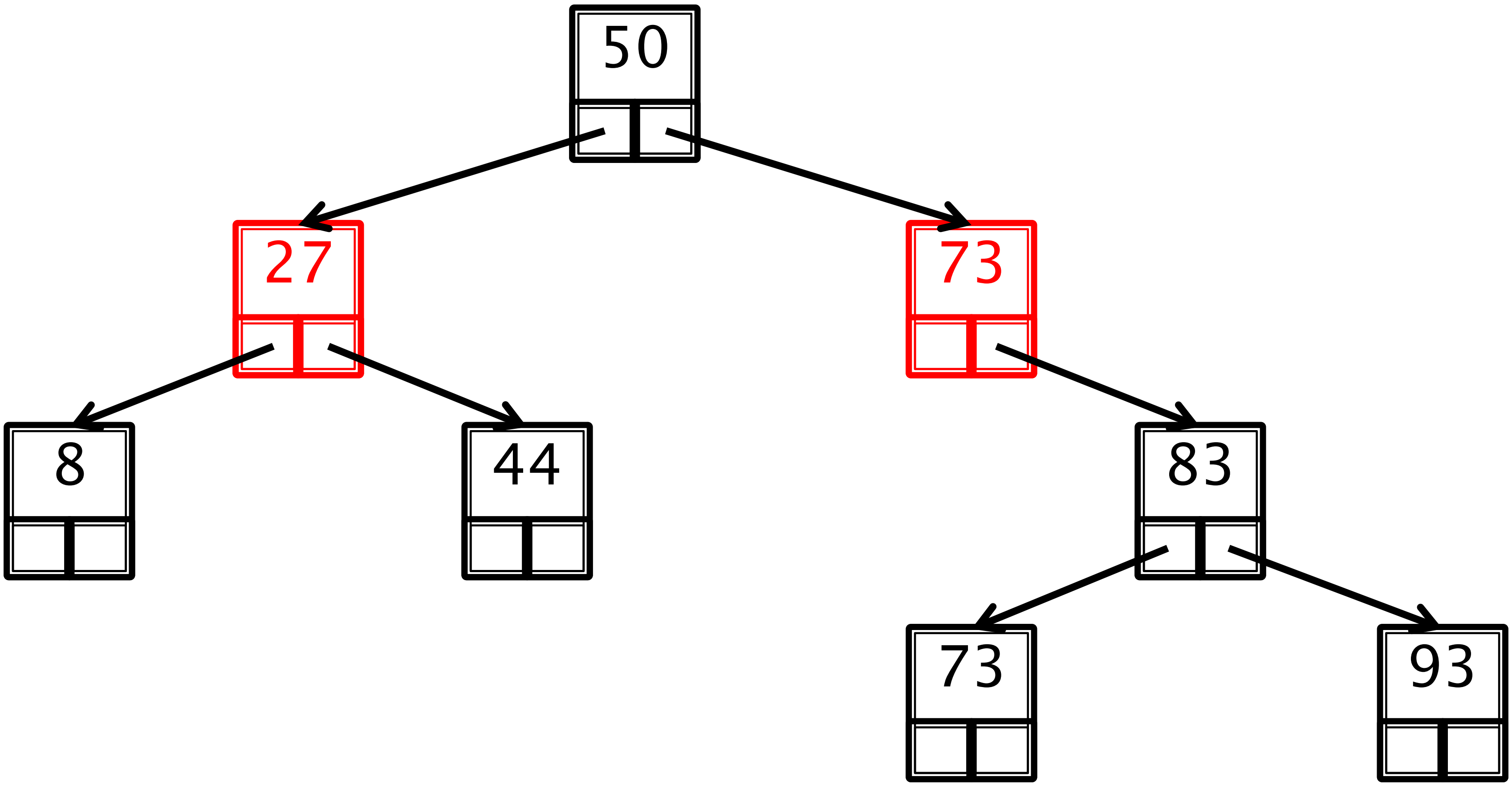

Inorder: 8, 27, 44, 50, 73, 73*, 83, 93

# Breadth-first search

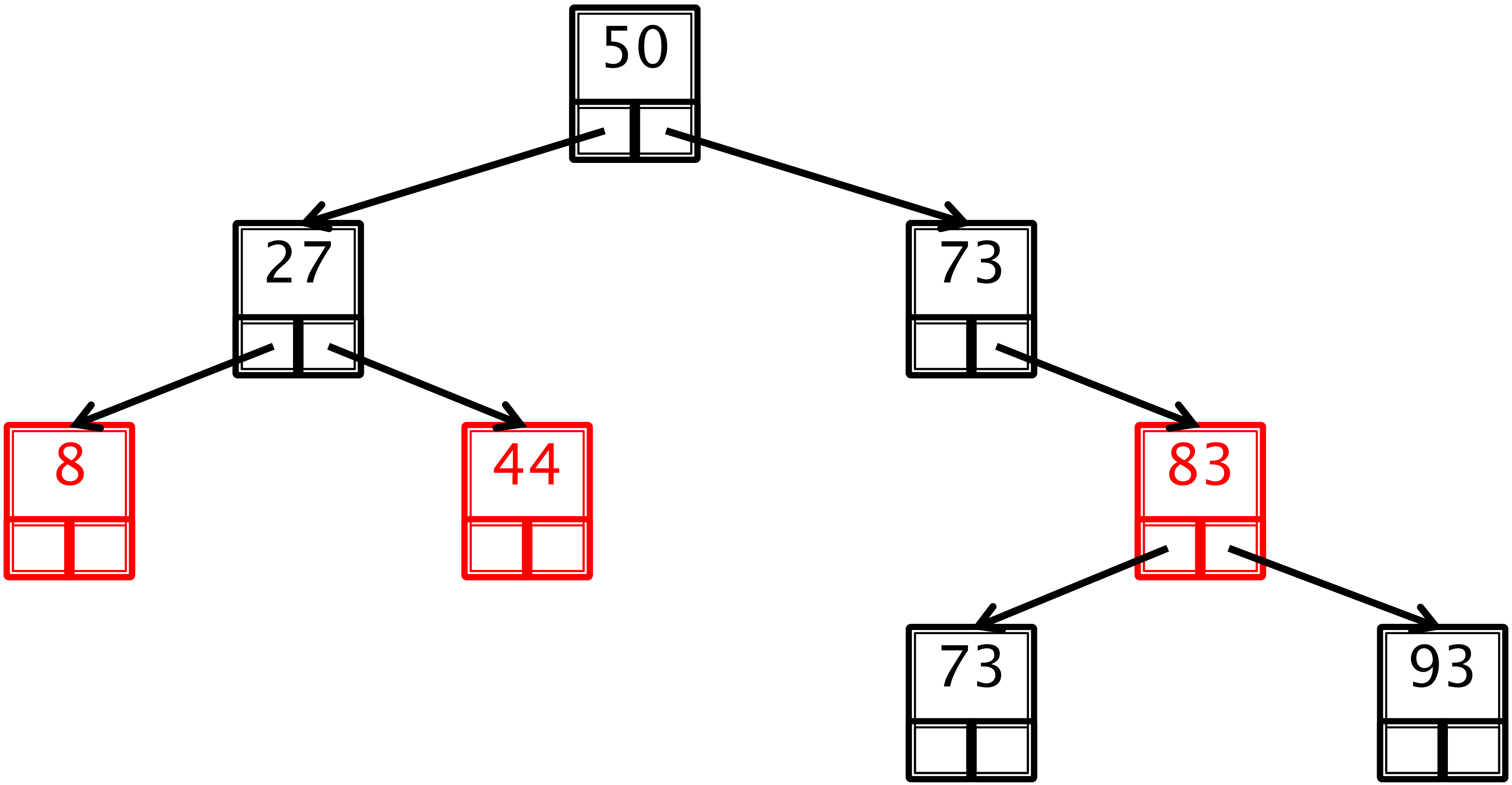
- Visiting every node of a tree using breadth-first search results in visiting nodes in order of their level in the tree



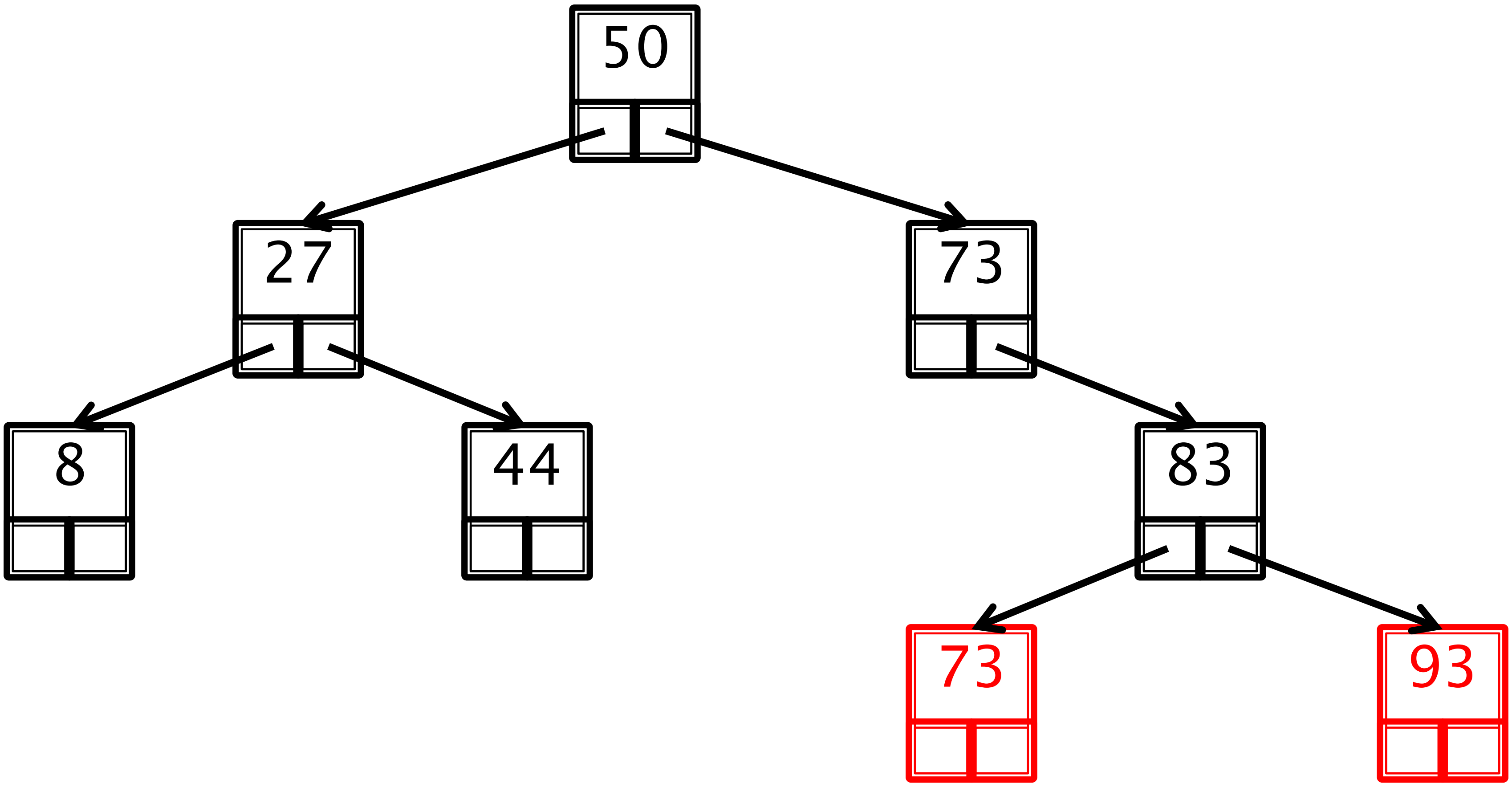

BFS: 50




BFS: 50, 27, 73



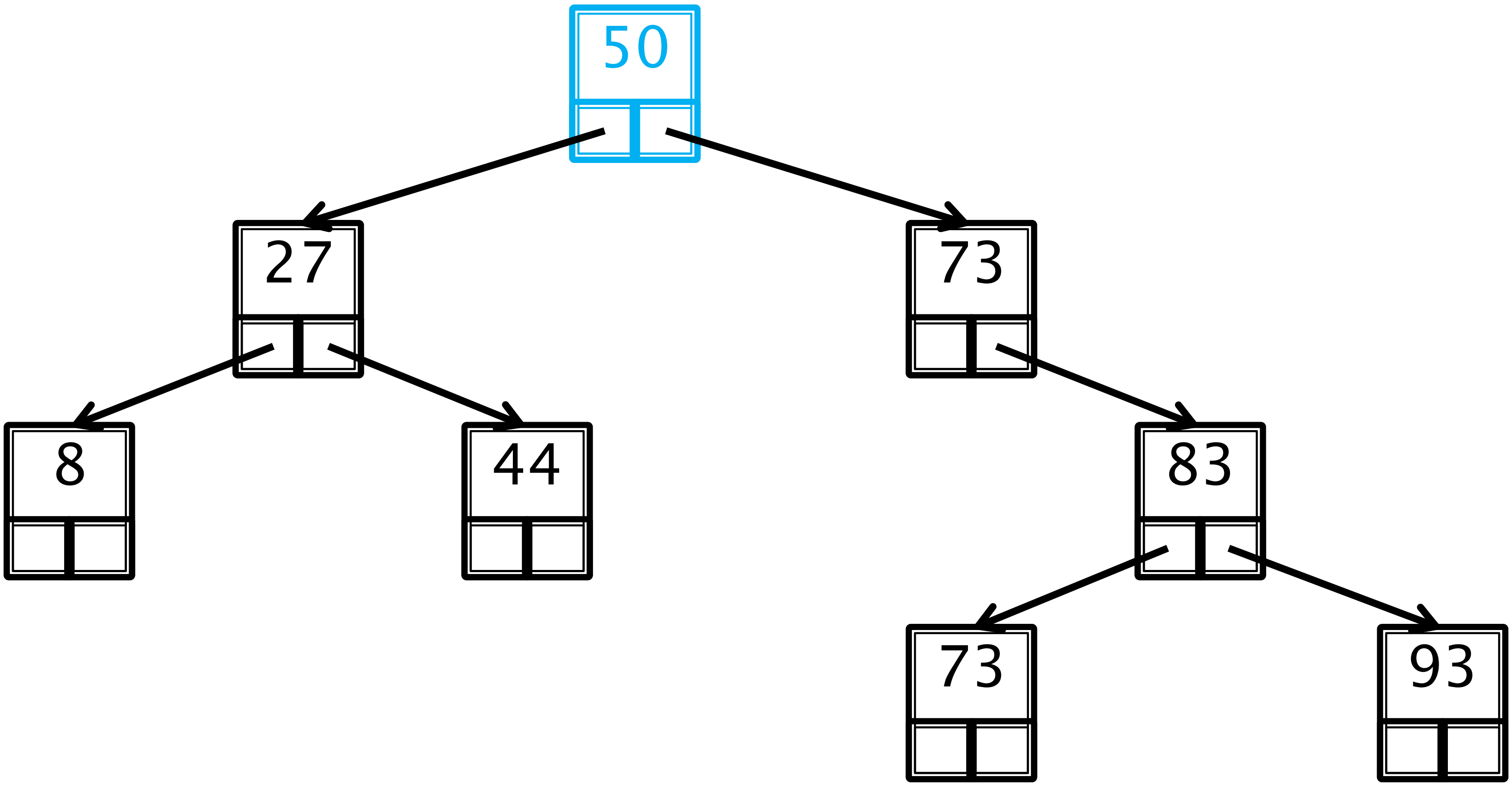

BFS: 50, 27, 73, 8, 44, 83



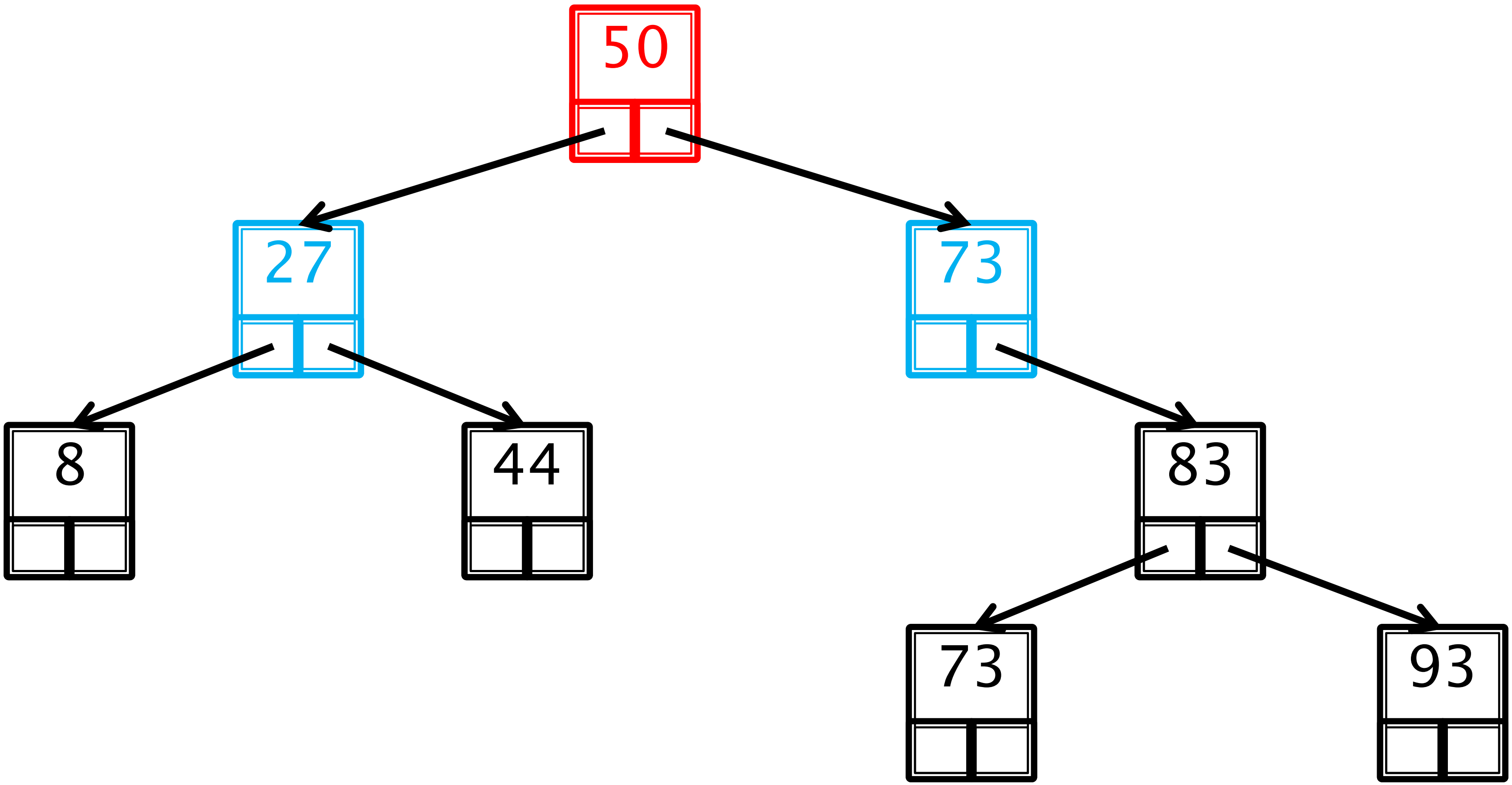

BFS: 50, 27, 73, 8, 44, 83, 73, 93

# 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)
  }
}
```

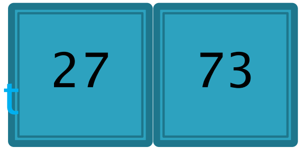


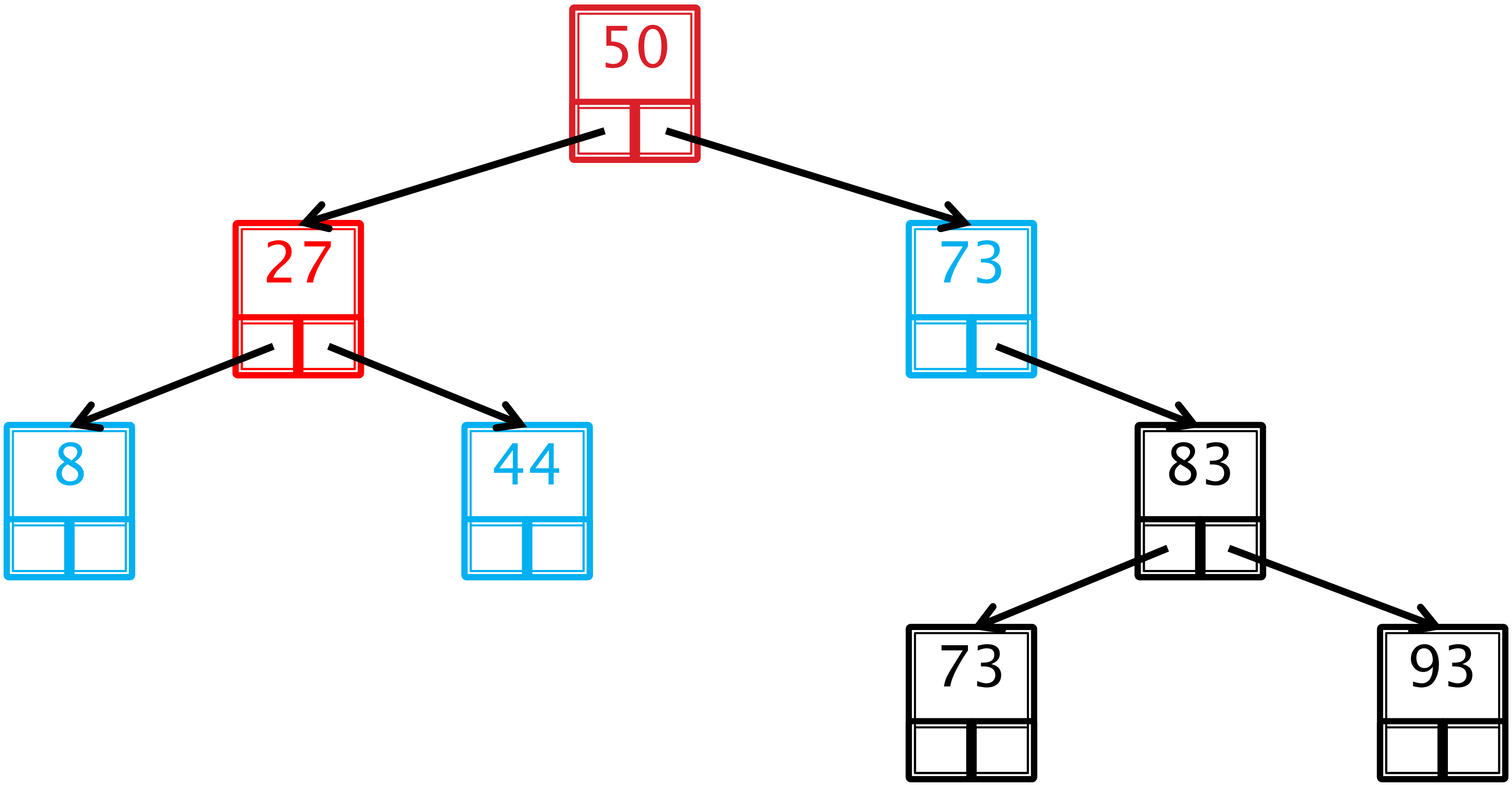

BFS:




BFS: 50

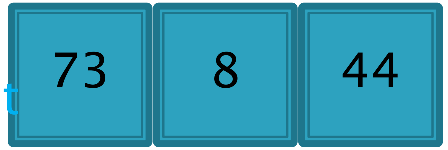
dequeue 50,
enqueue left and right

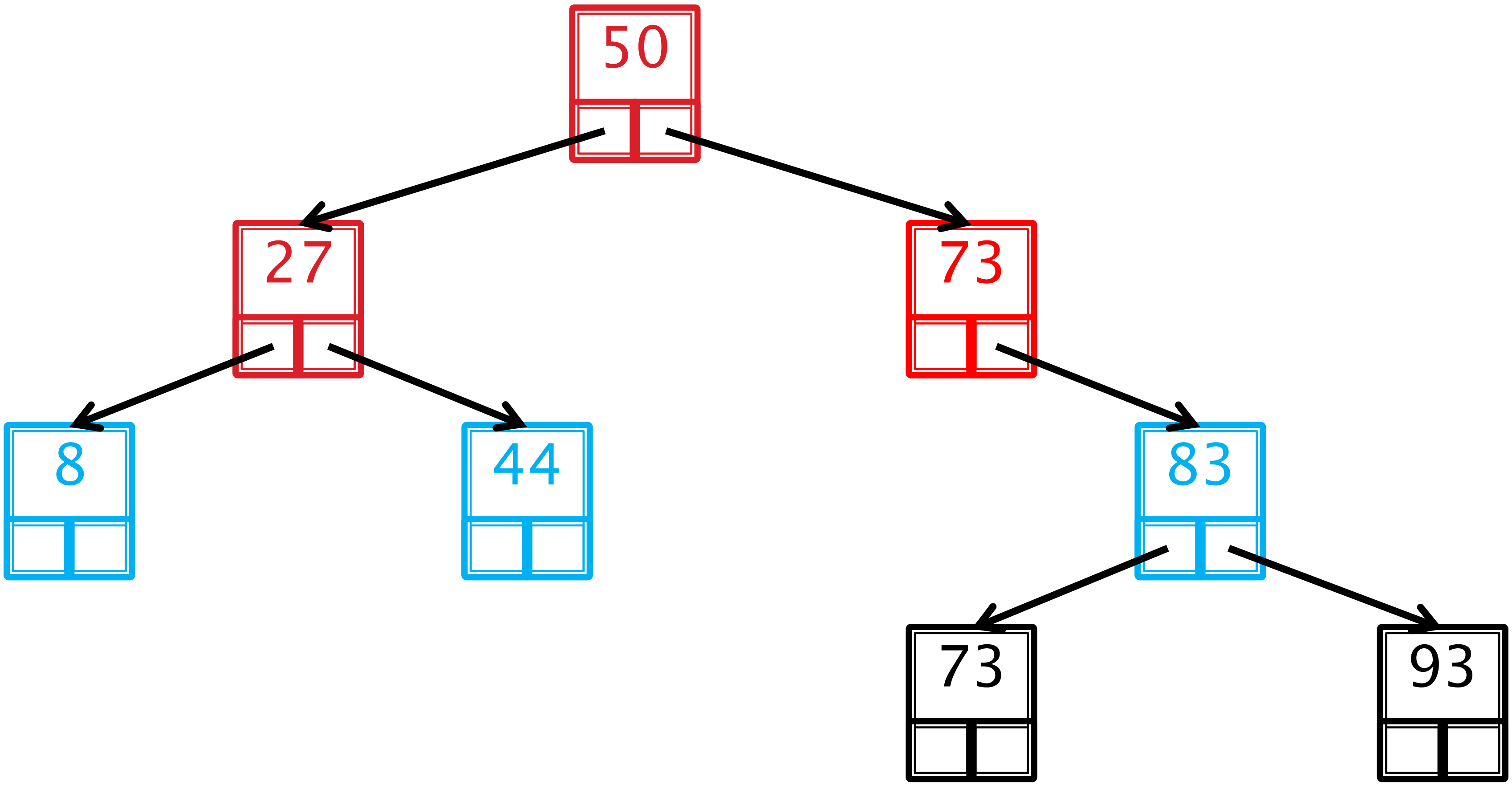





BFS: 50, 27

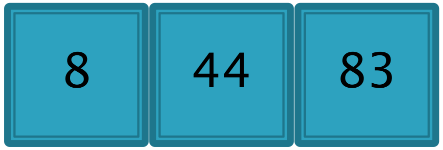
dequeue 27,
enqueue left and right

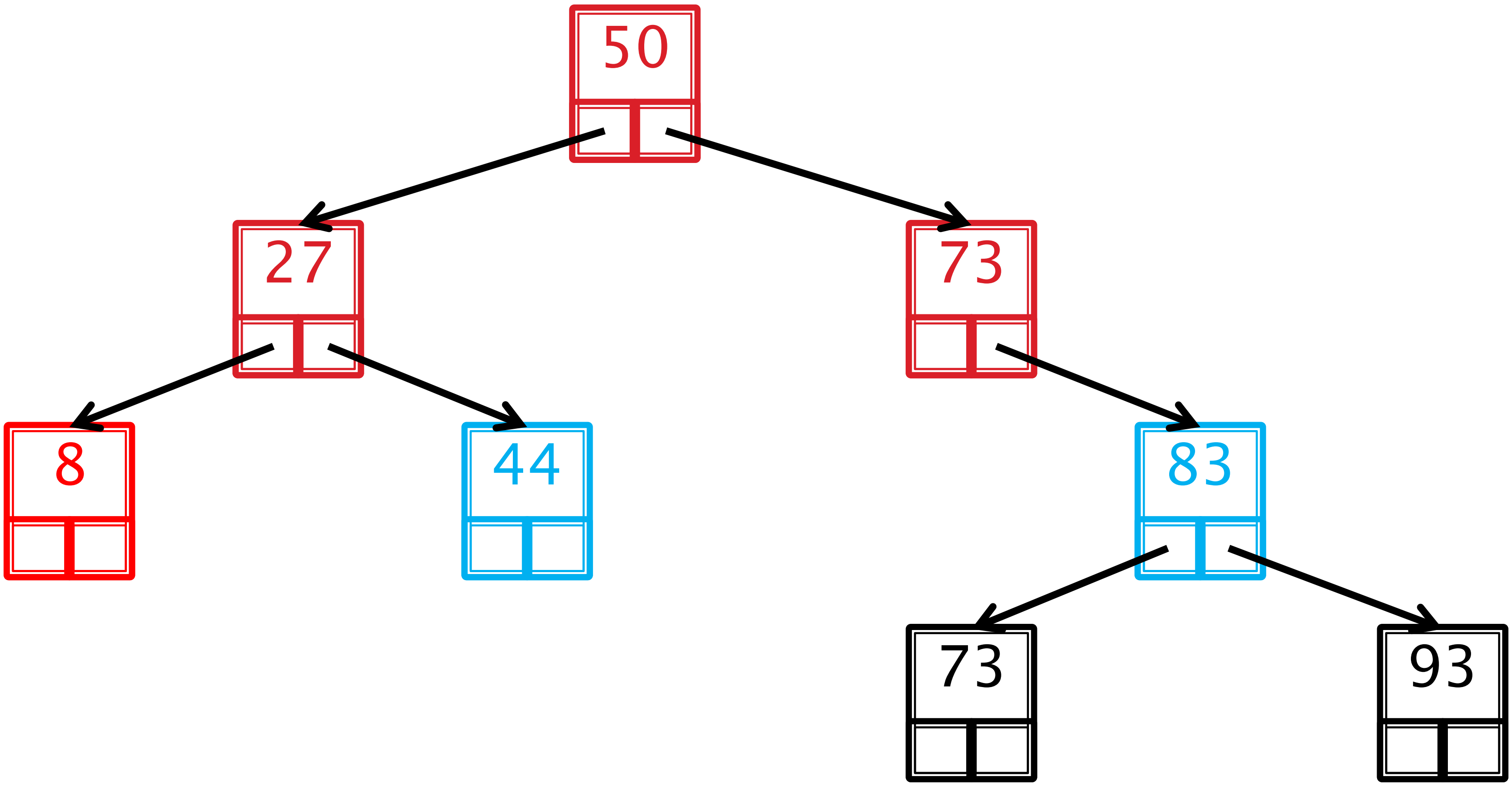





BFS: 50, 27, 73

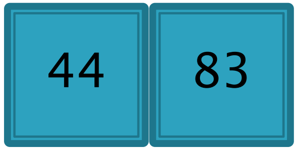
dequeue 73,
enqueue right

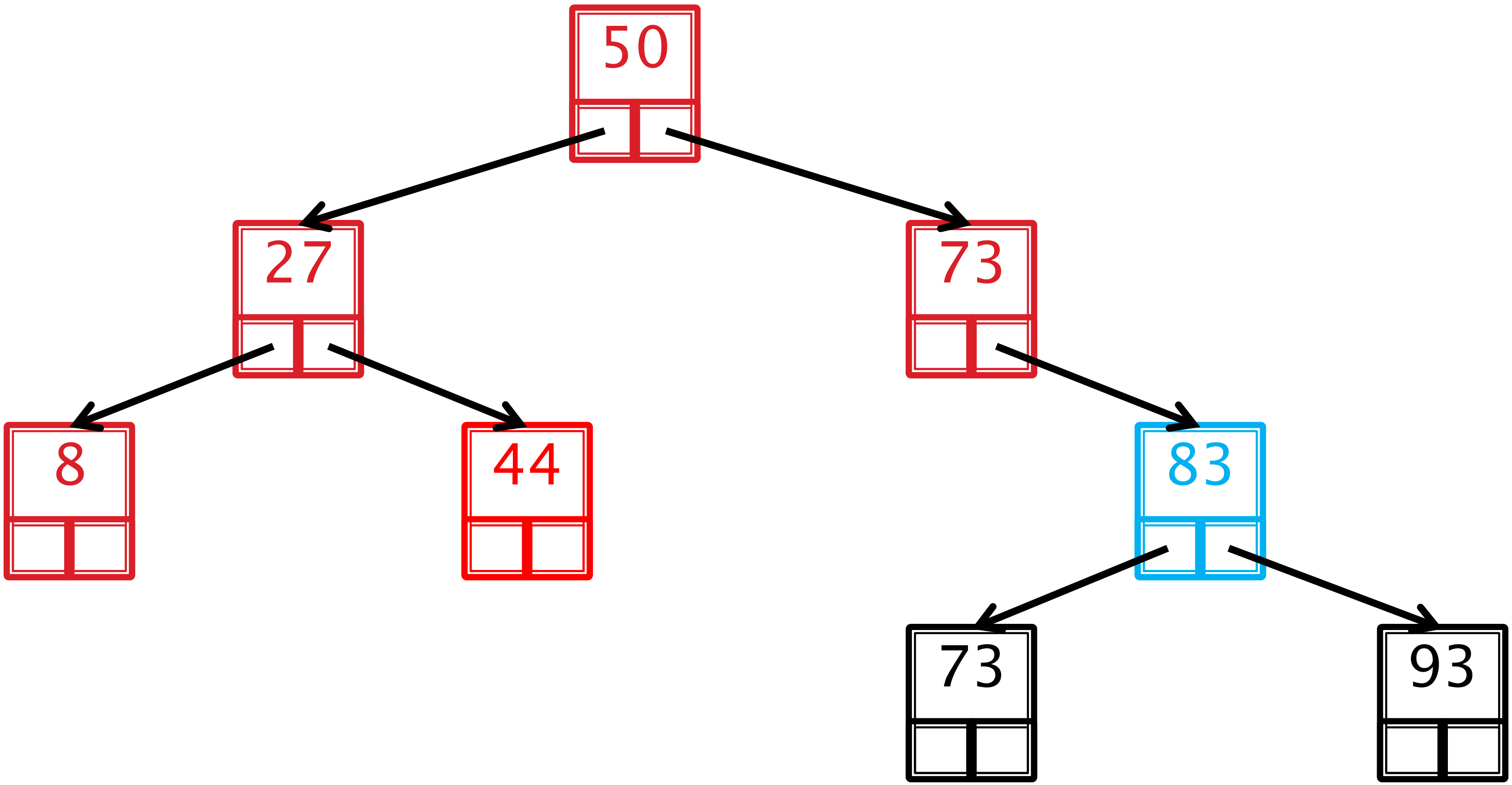





BFS: 50, 27, 73, 8

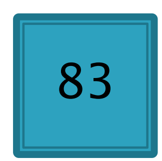
dequeue 8

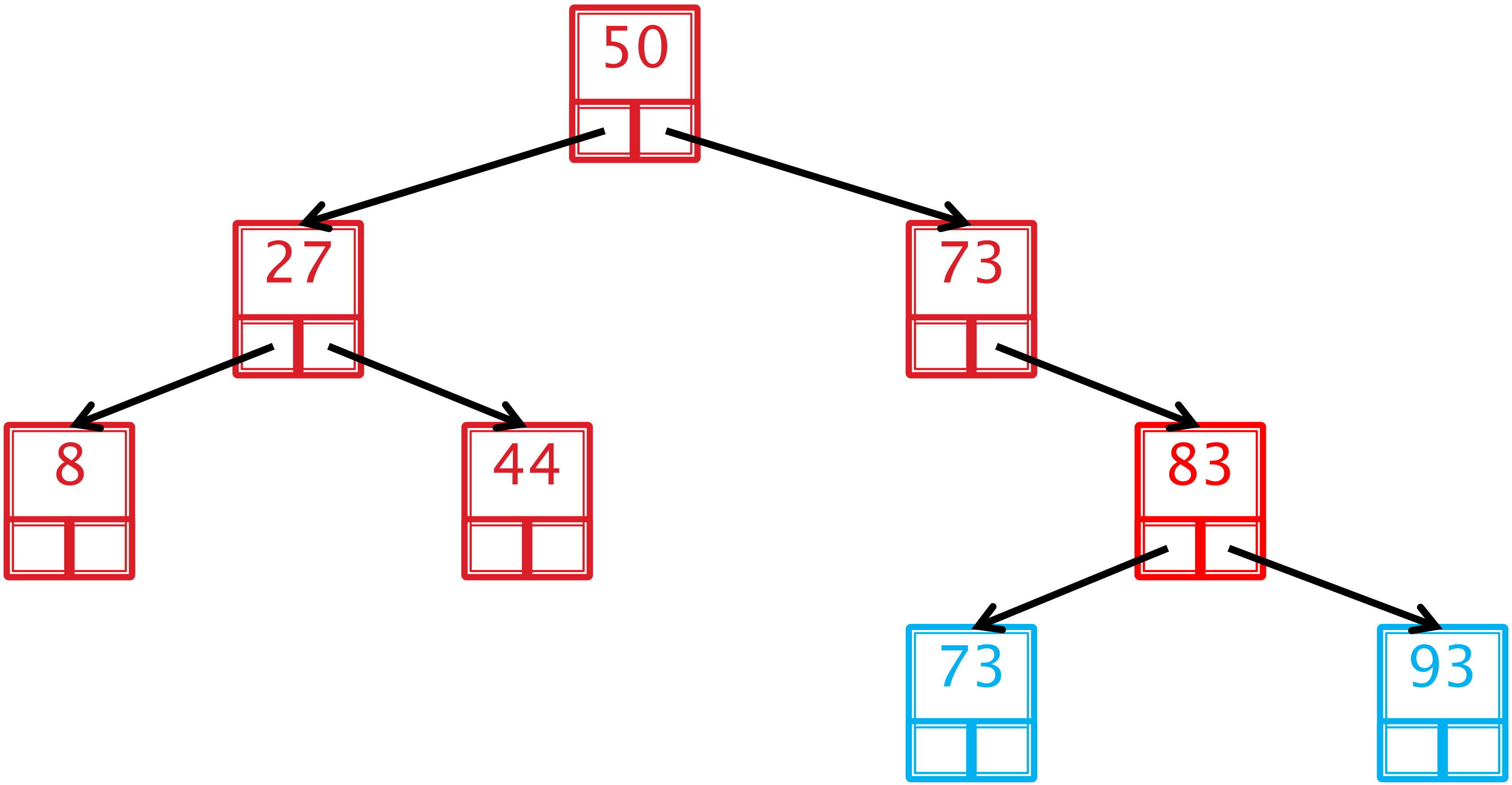





BFS: 50, 27, 73, 8, 44

dequeue 44

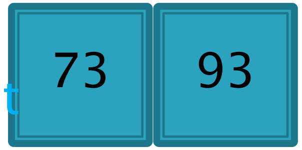


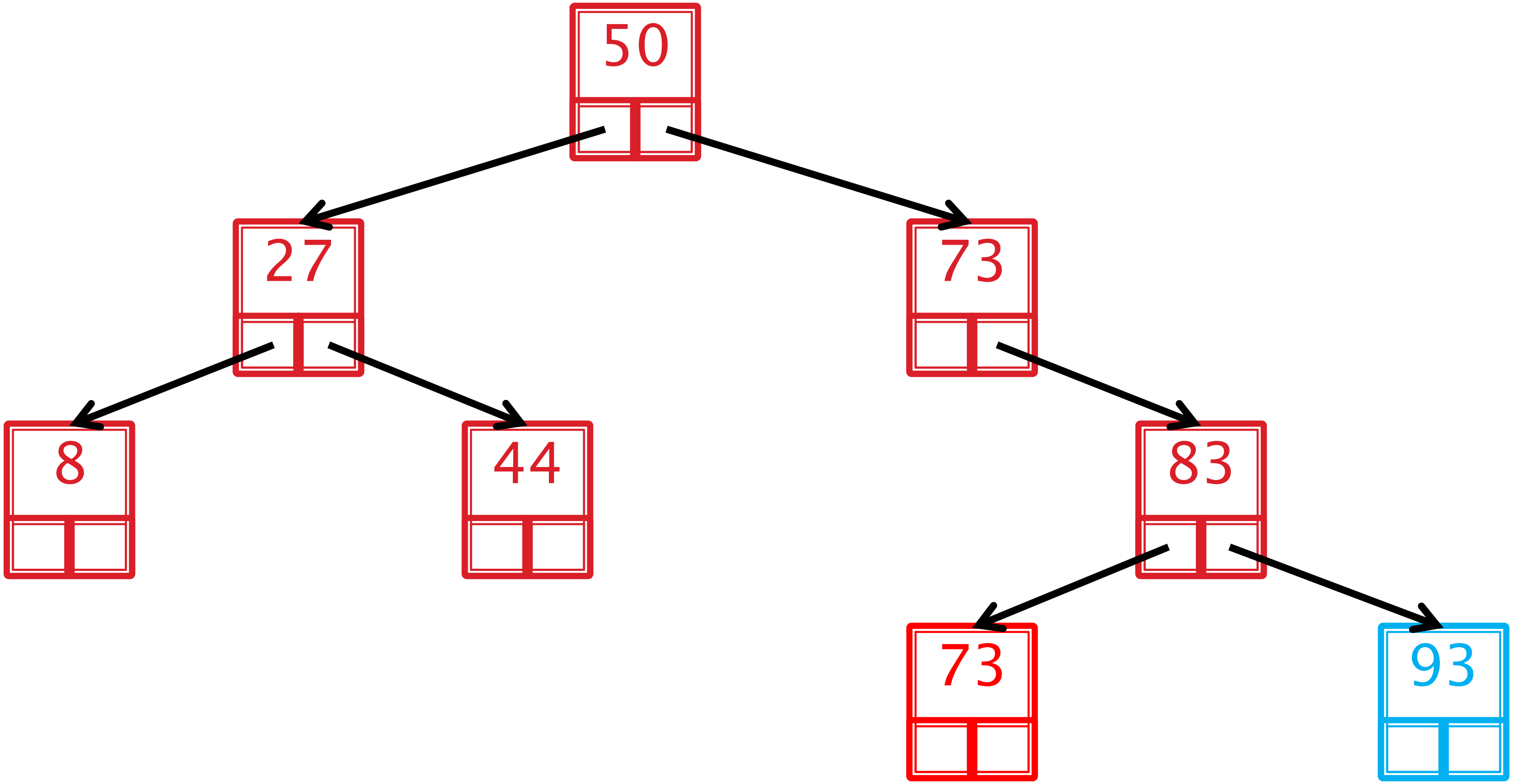



BFS: 50, 27, 73, 8, 44, 83

dequeue 83,

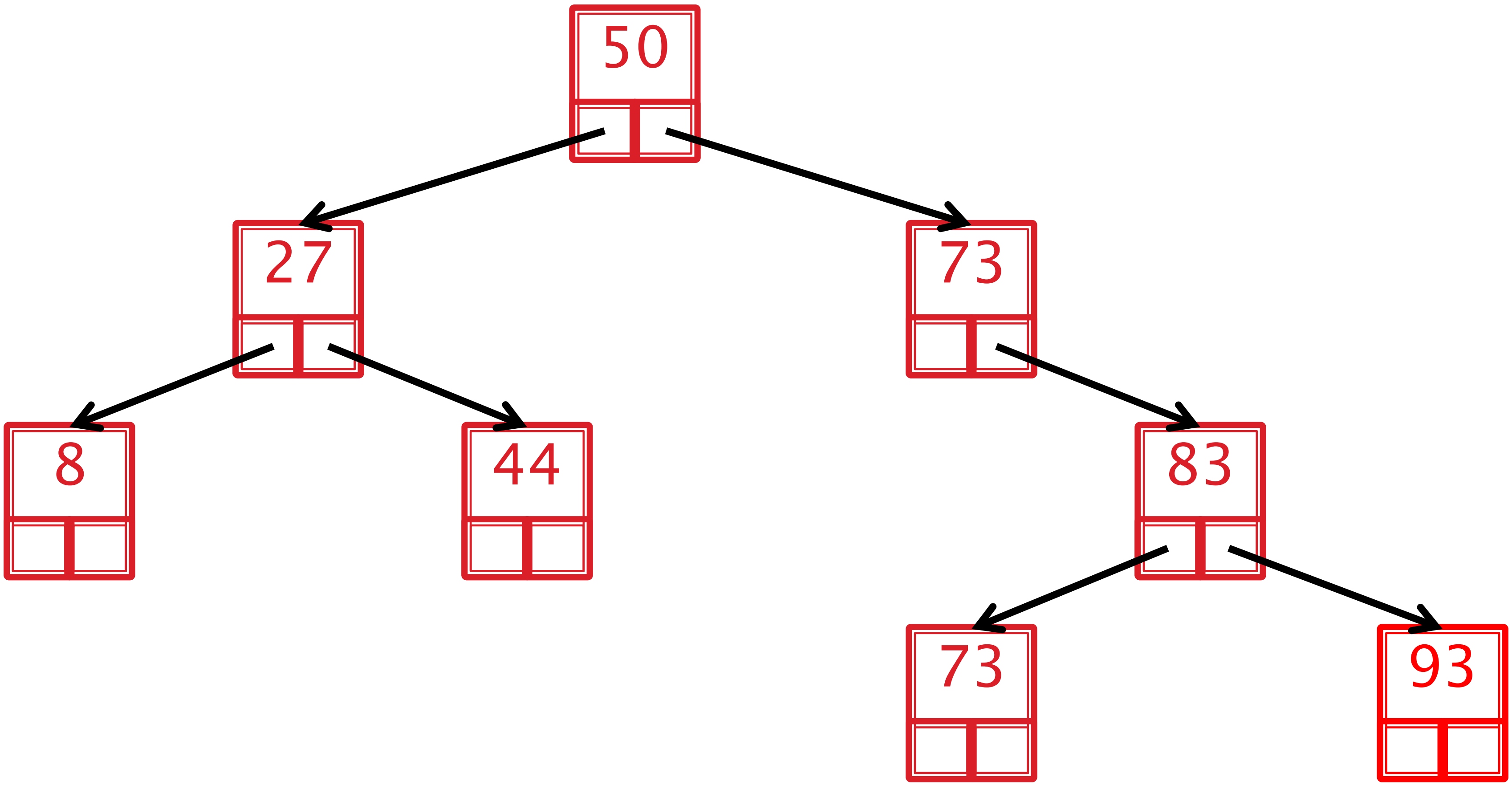
enqueue left and right





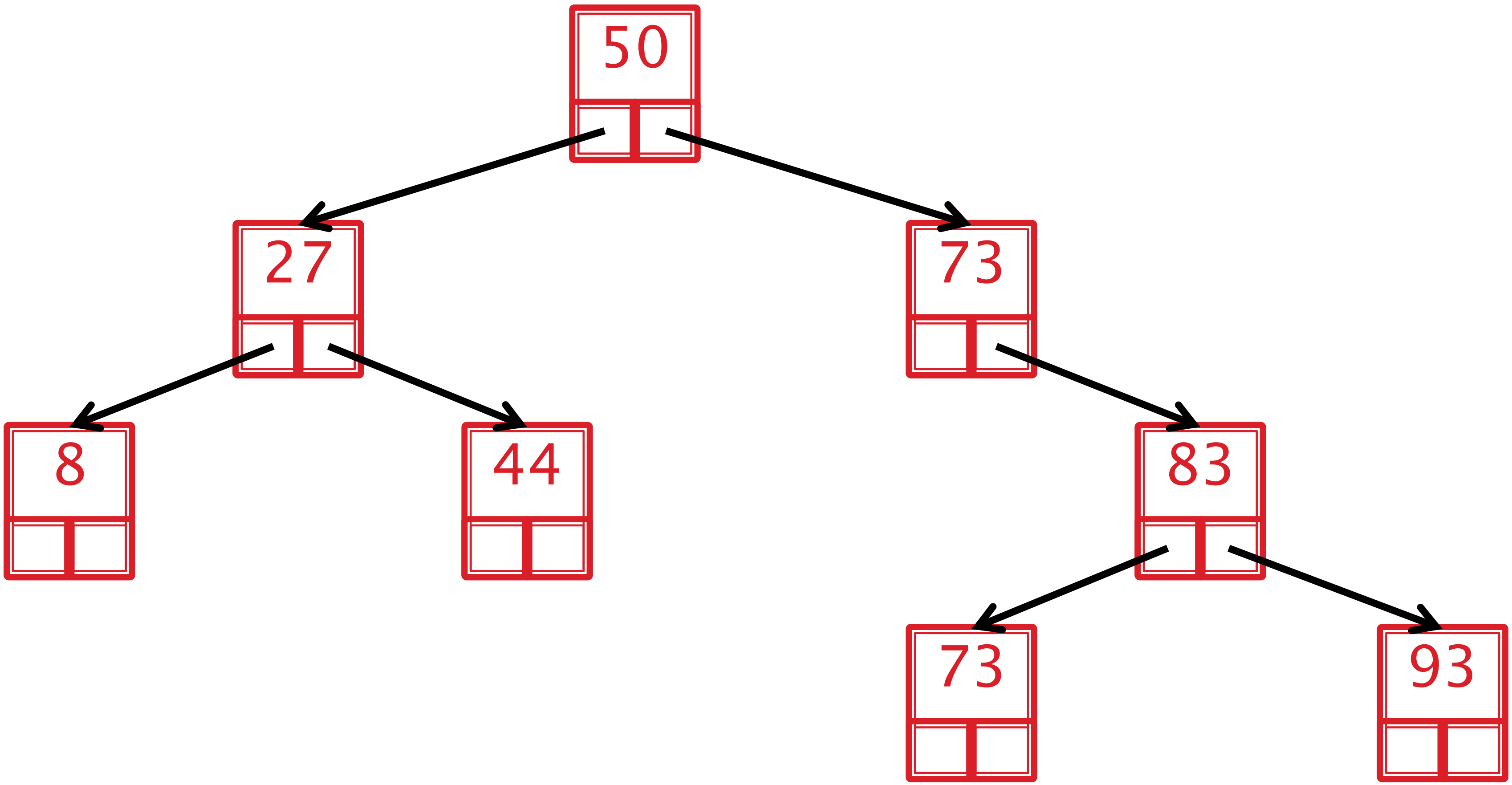

BFS: 50, 27, 73, 8, 44, 83, 73

dequeue 73




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

dequeue 93




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

queue empty