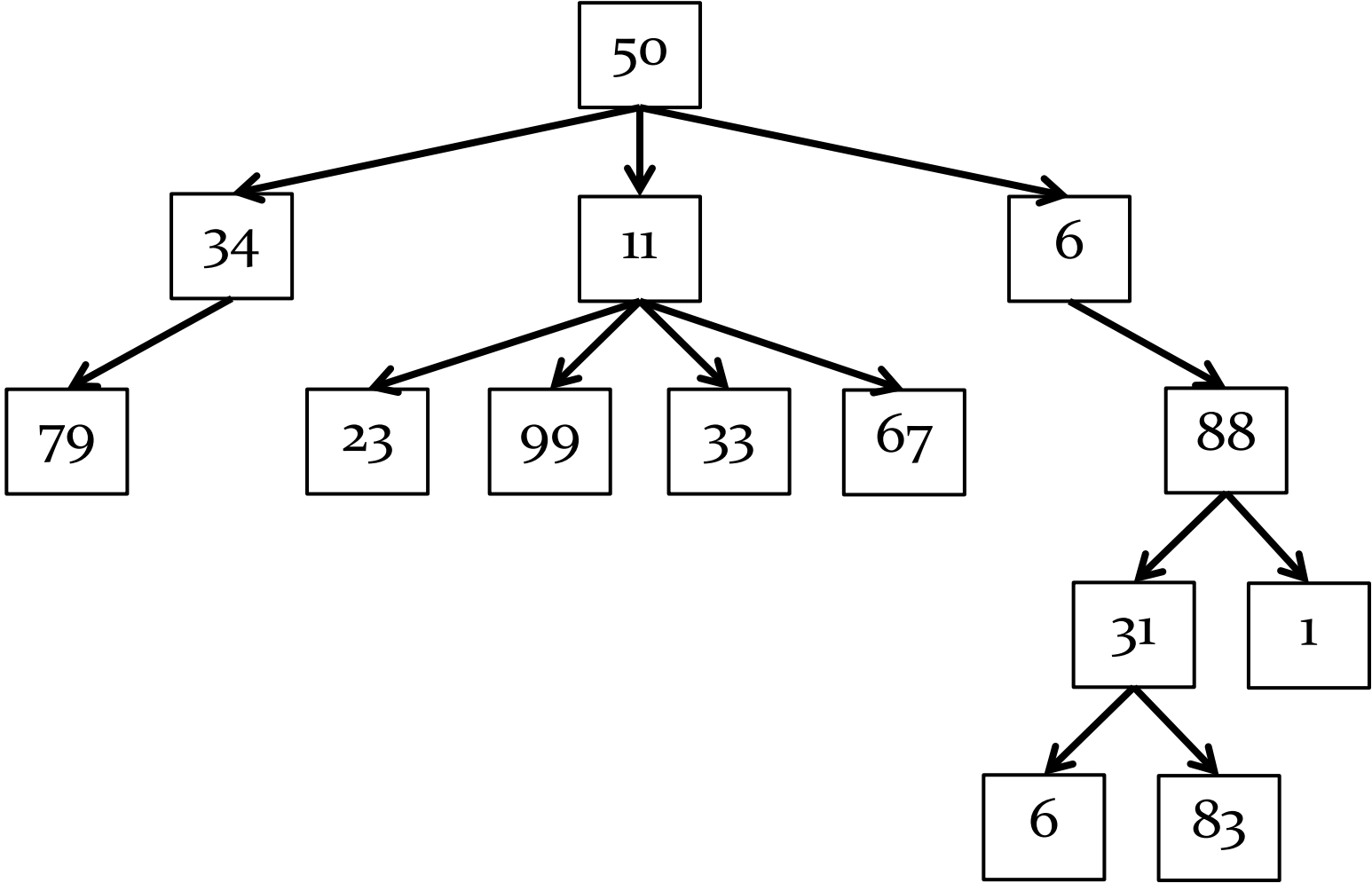


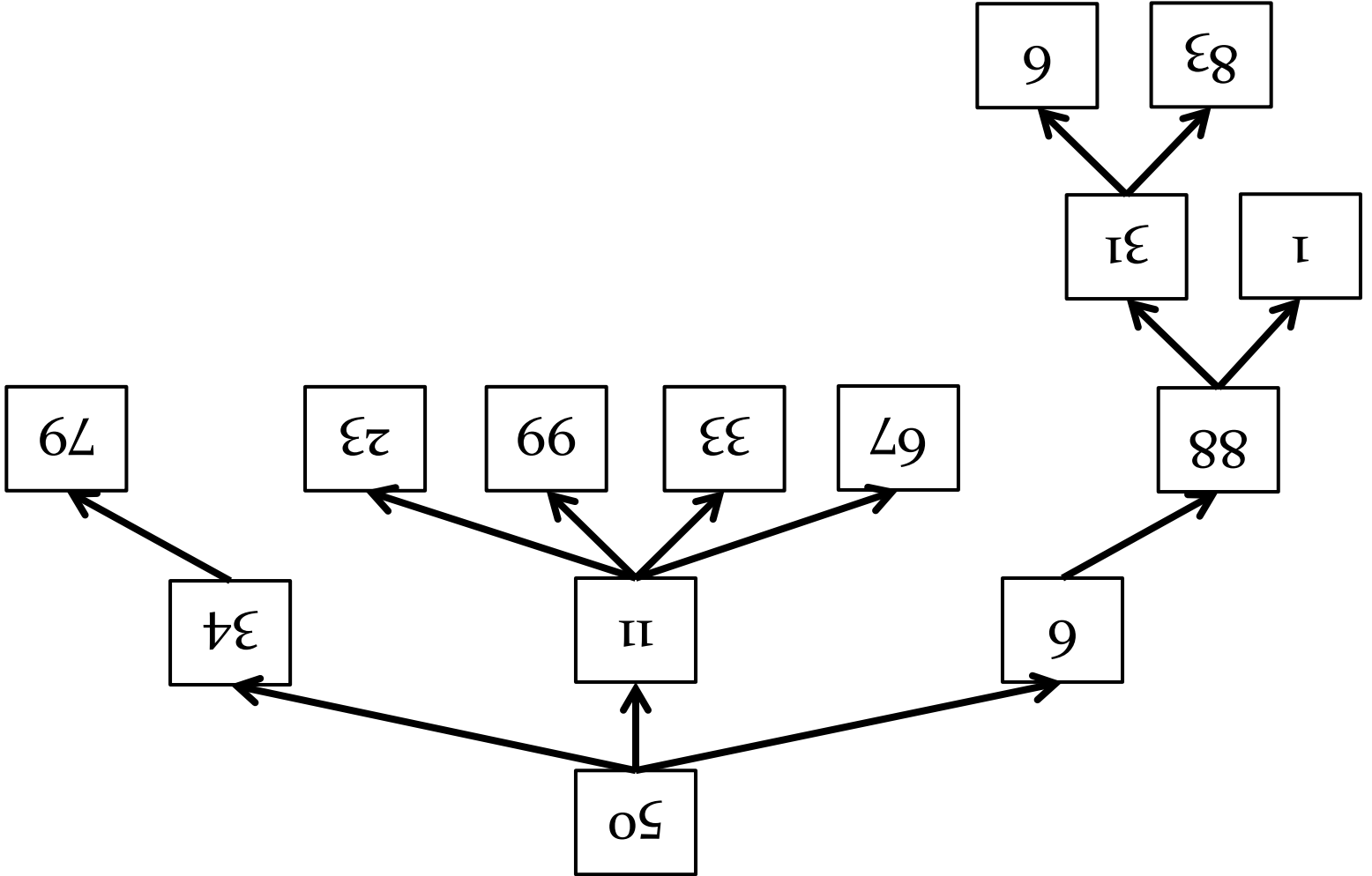
# Recursive Objects (Part 4)

# 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

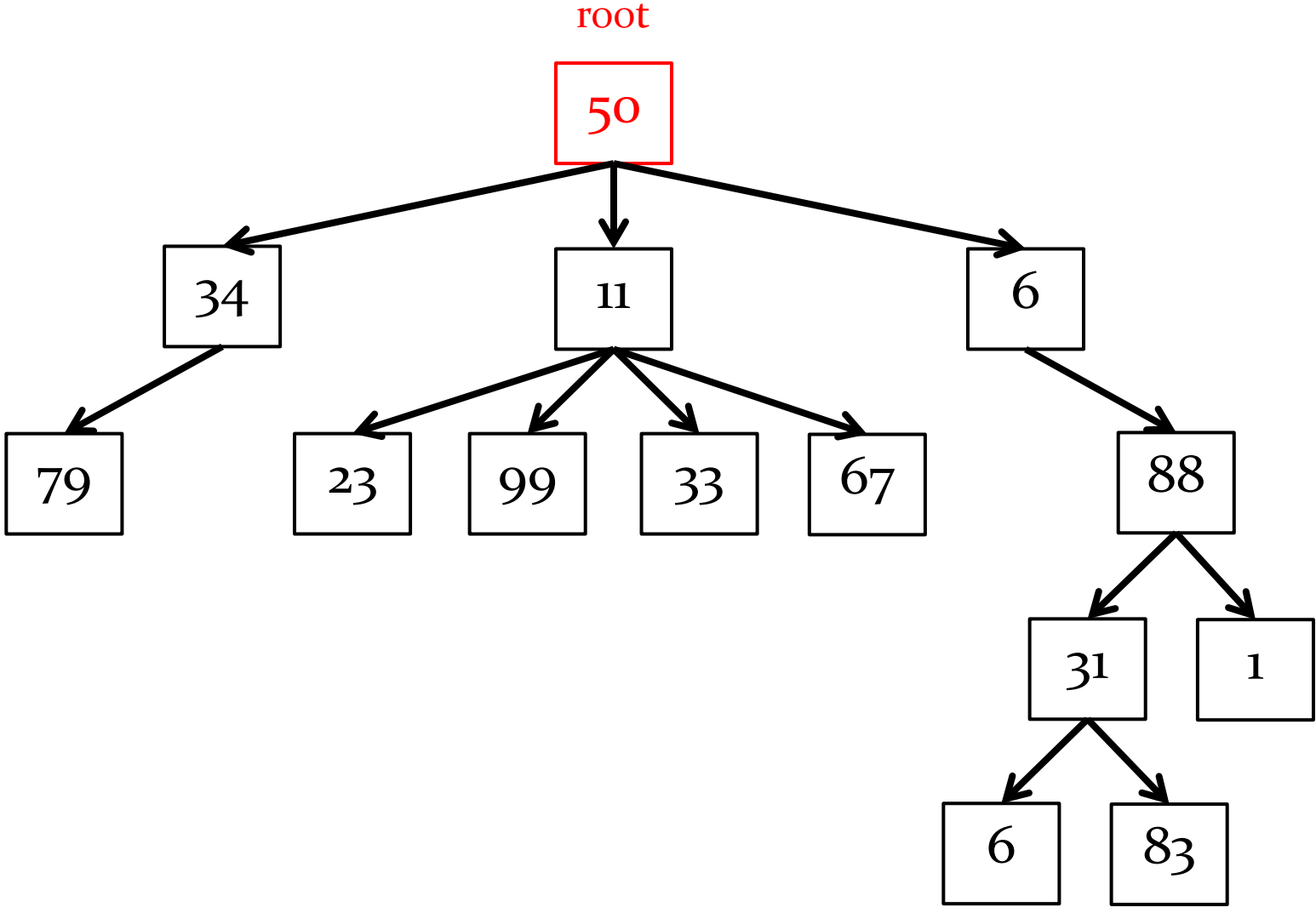




# Trees

---

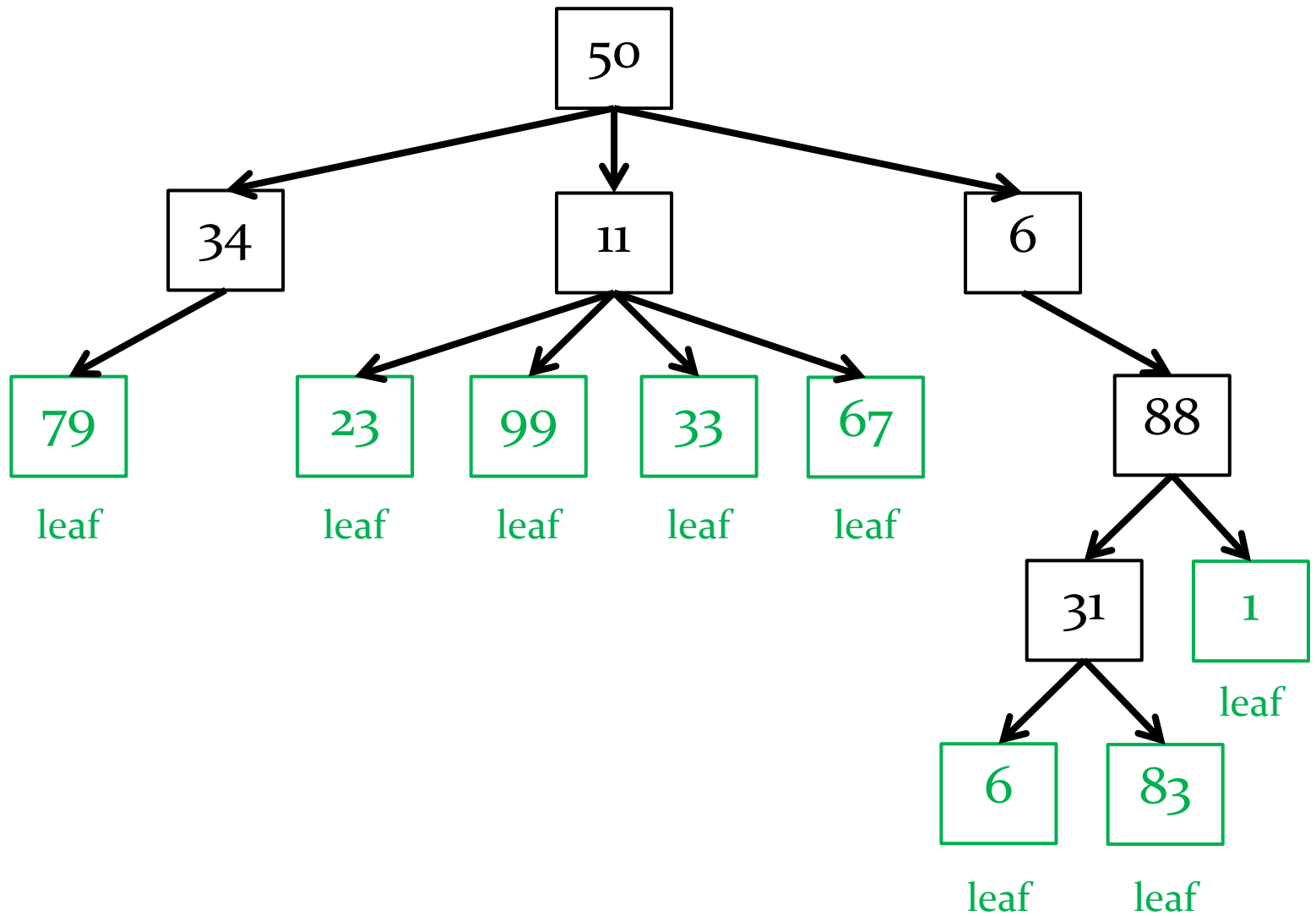
- ▶ the root of the tree is the node that has no parent node
- ▶ all algorithms start at the root



# Trees

---

- ▶ a node without any children is called a leaf

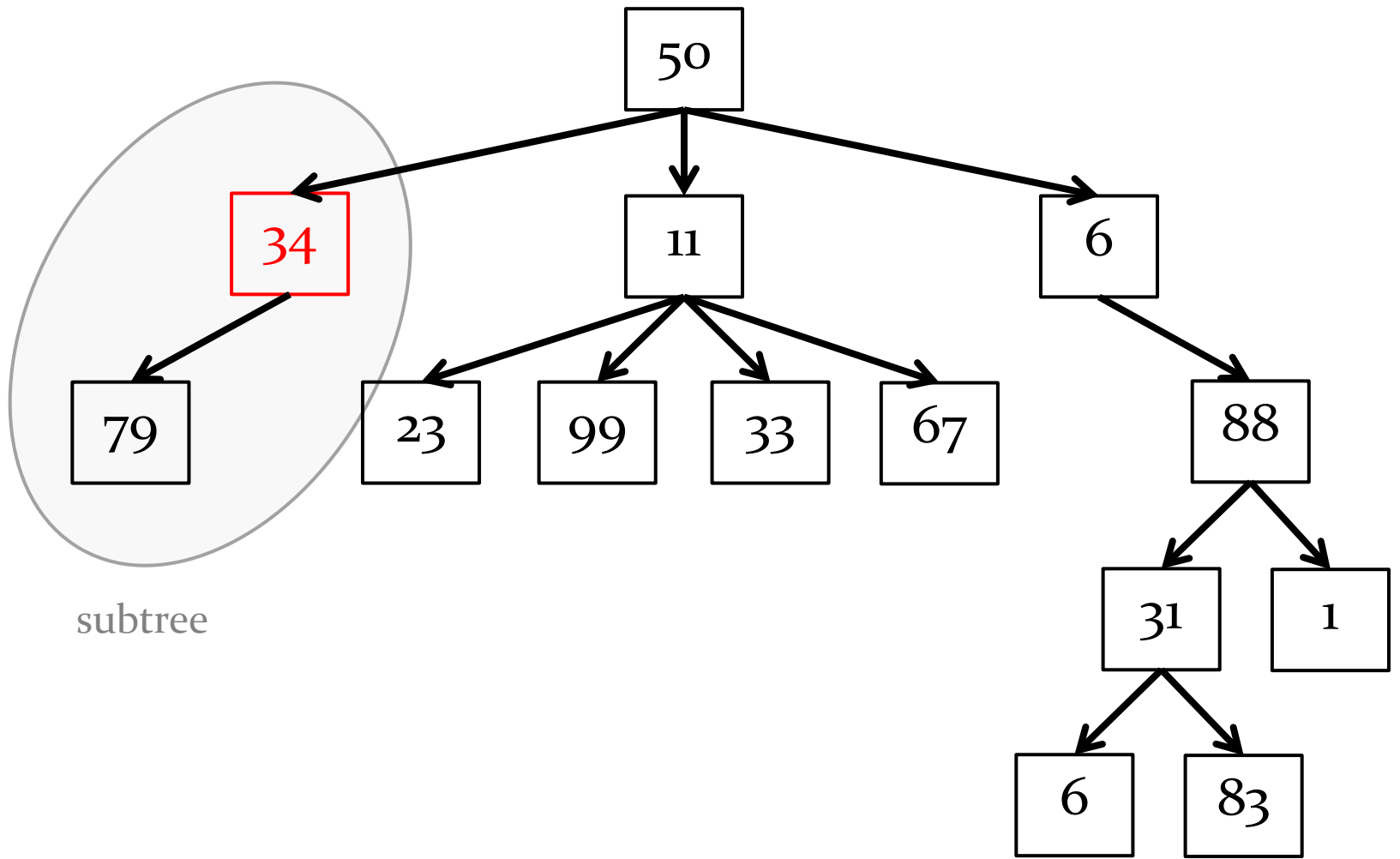


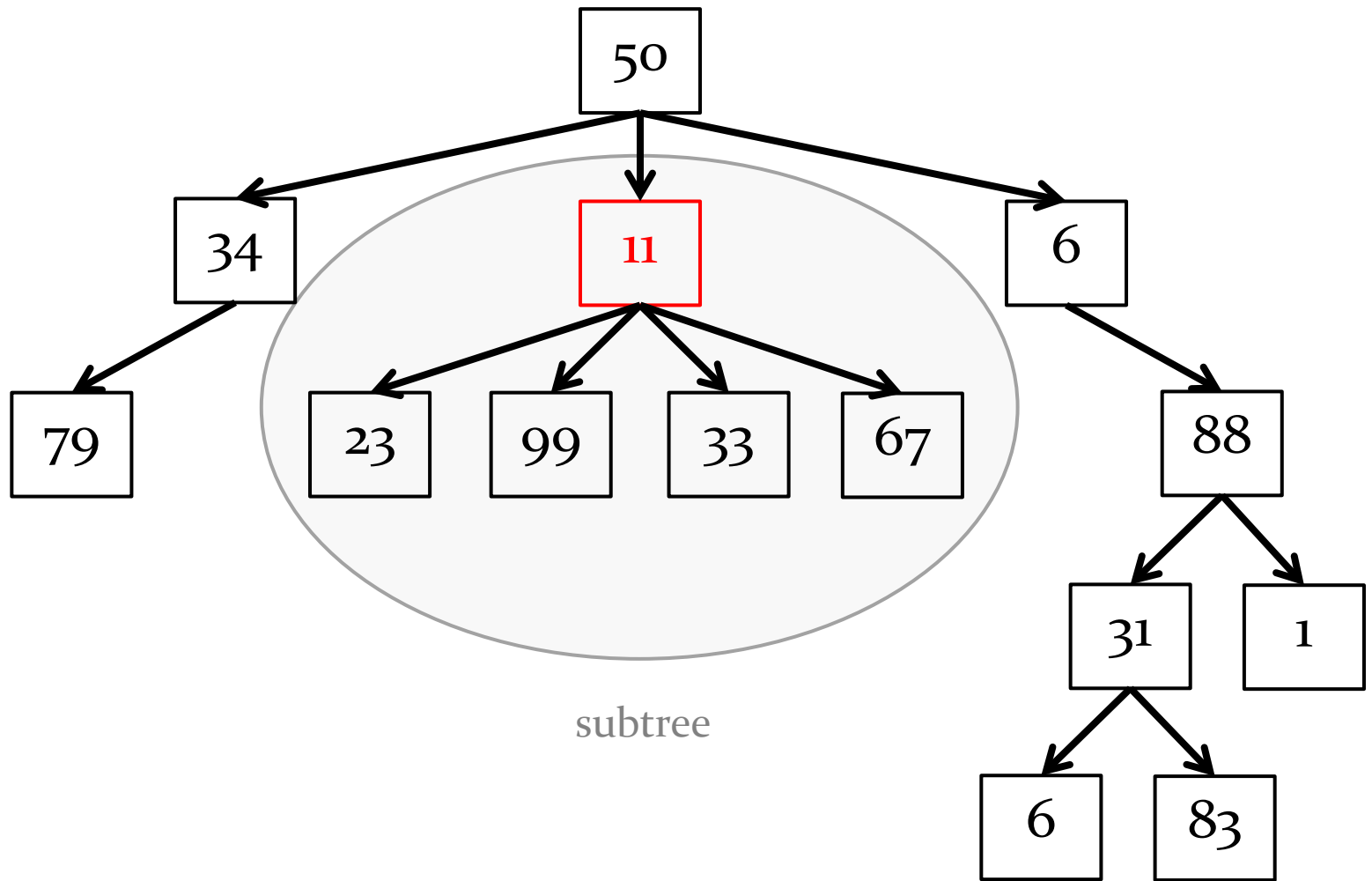


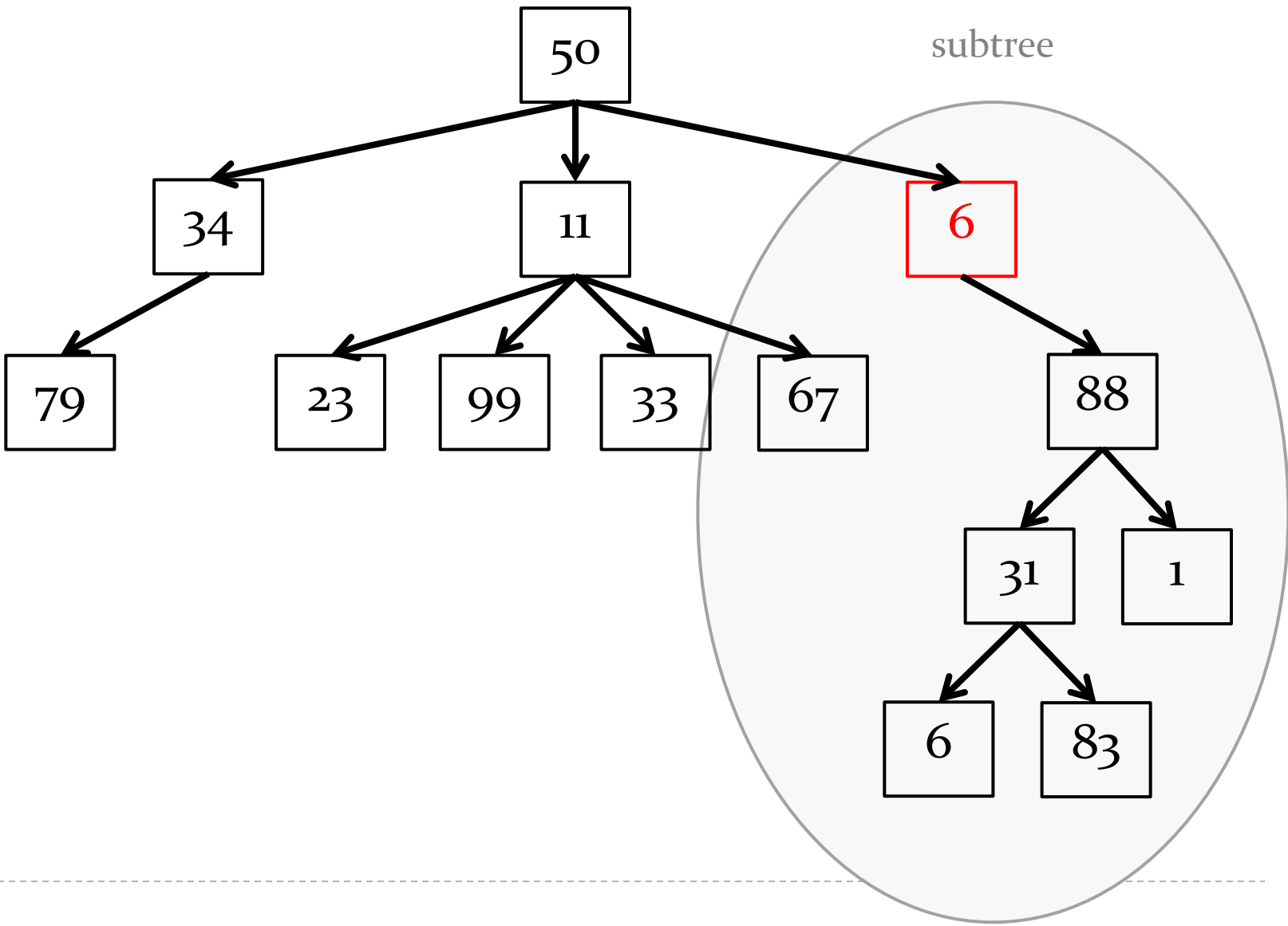
# Trees

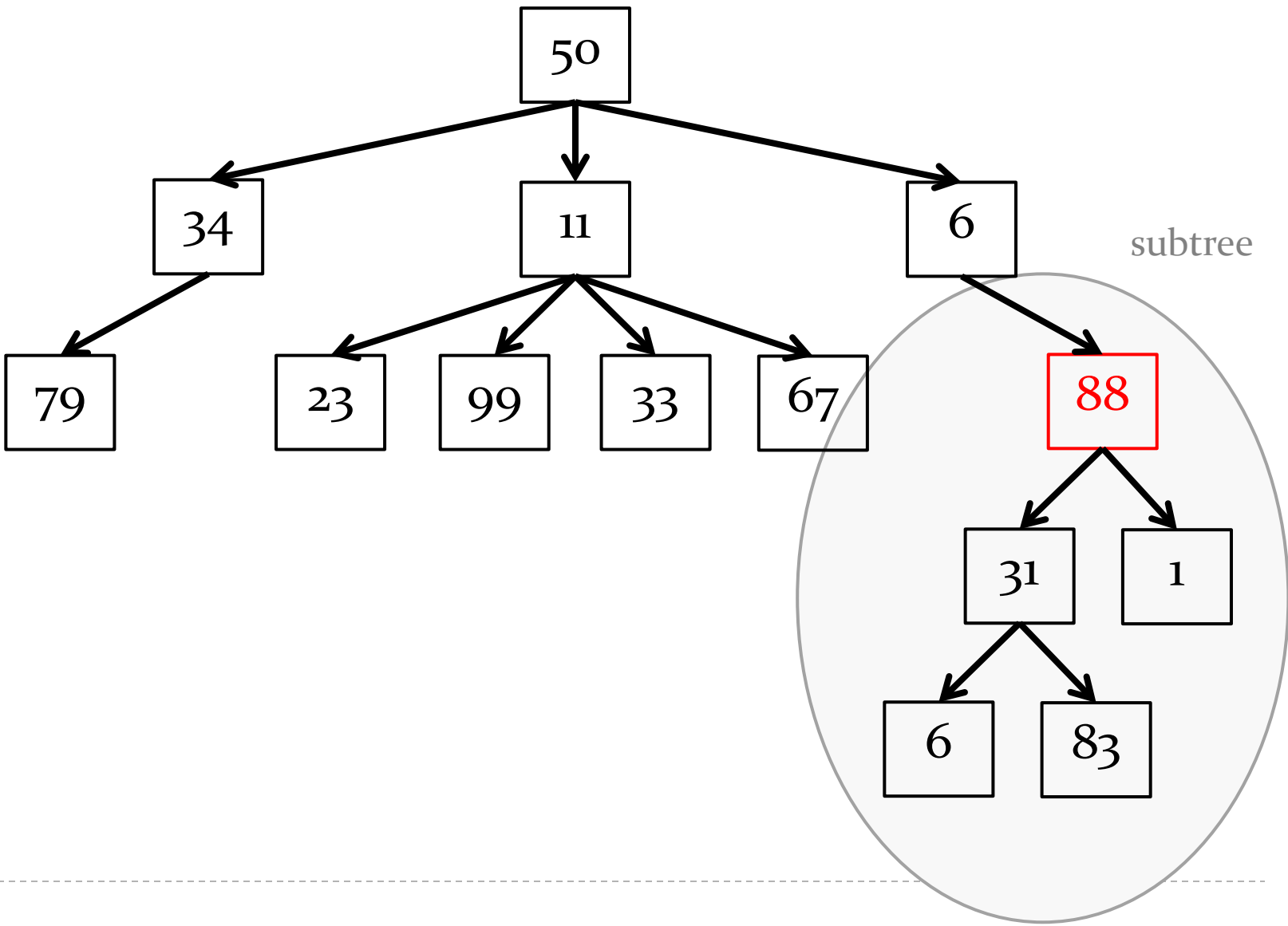
---

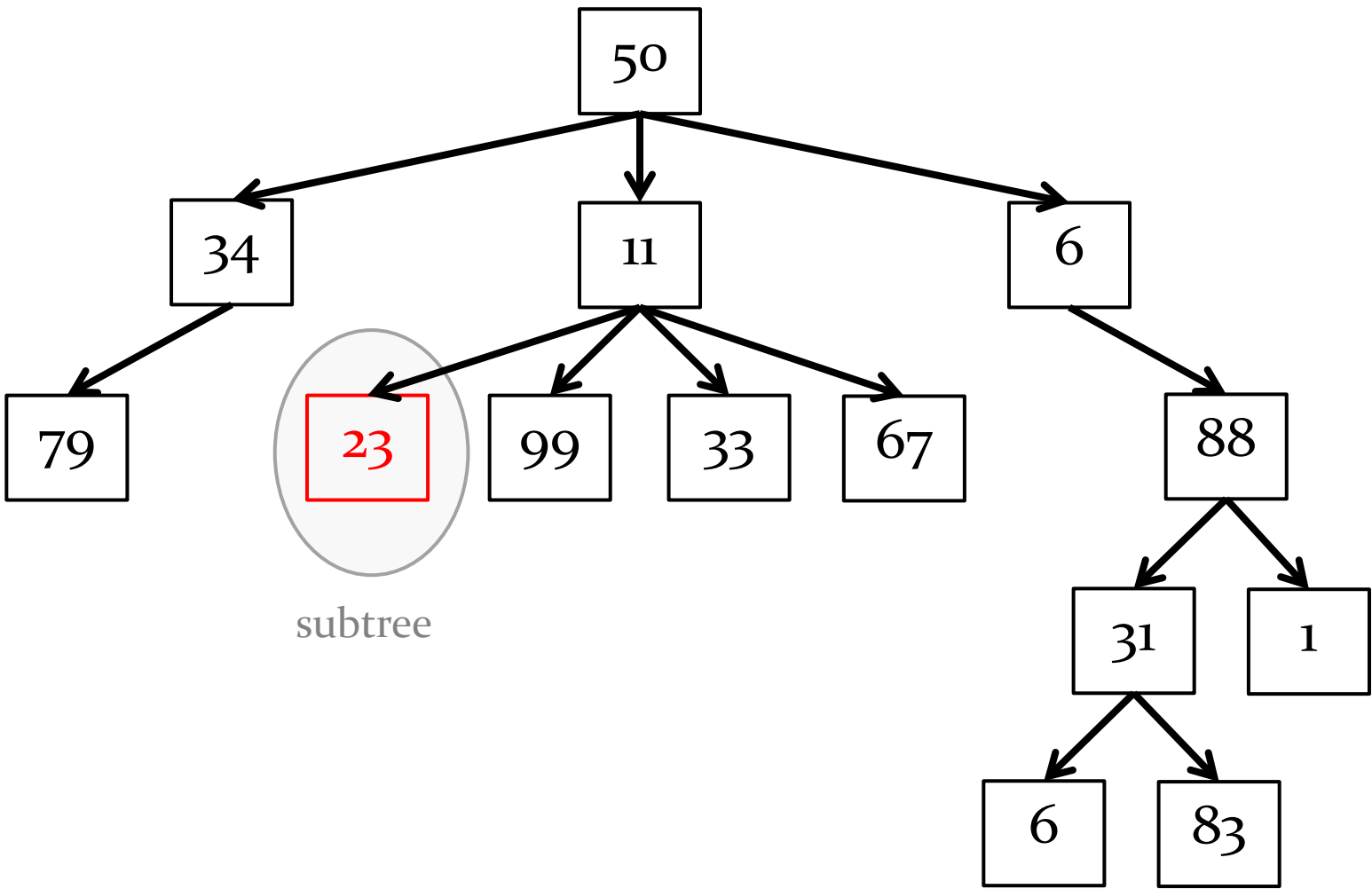
- ▶ the recursive structure of a tree means that every node is the root of a tree







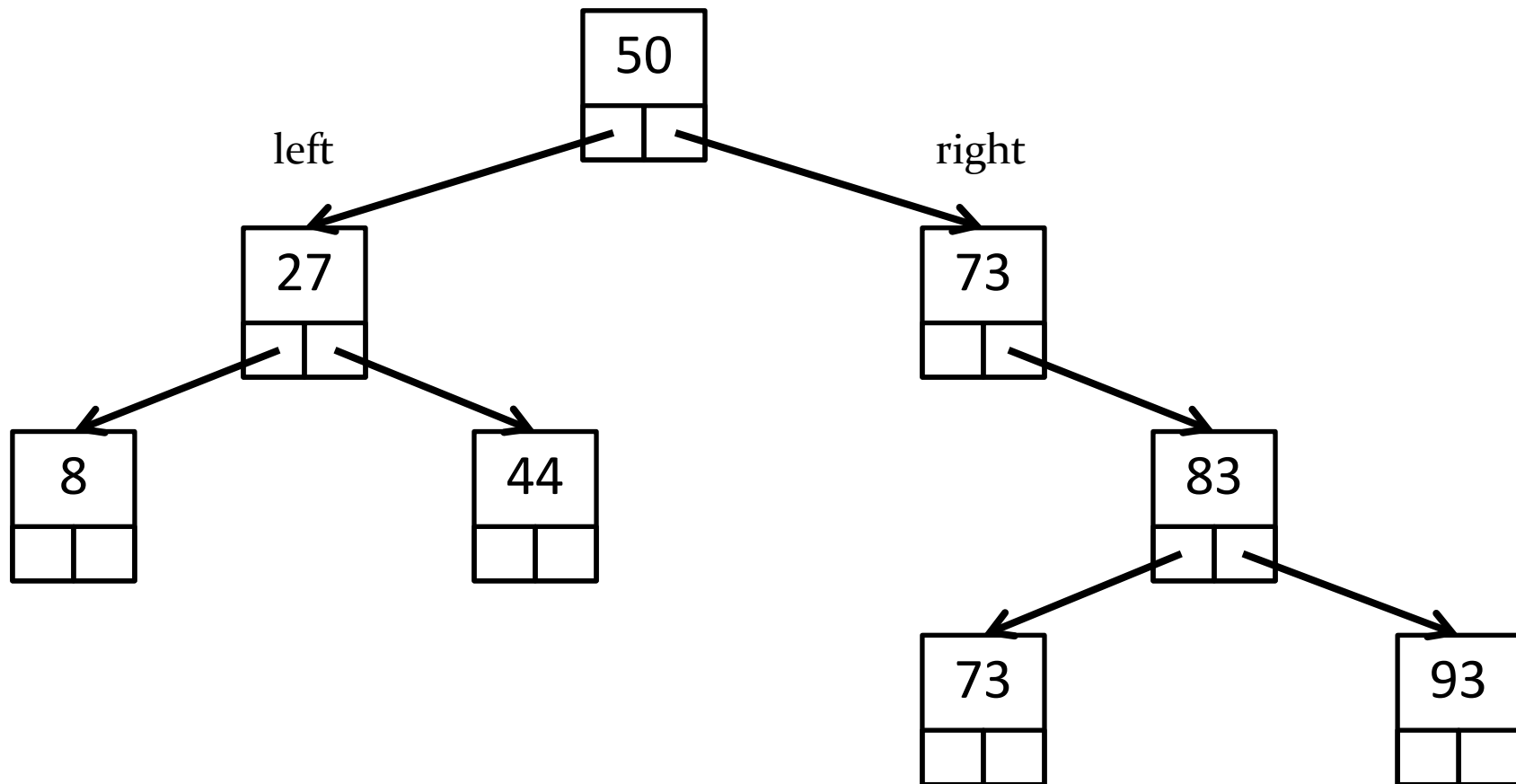




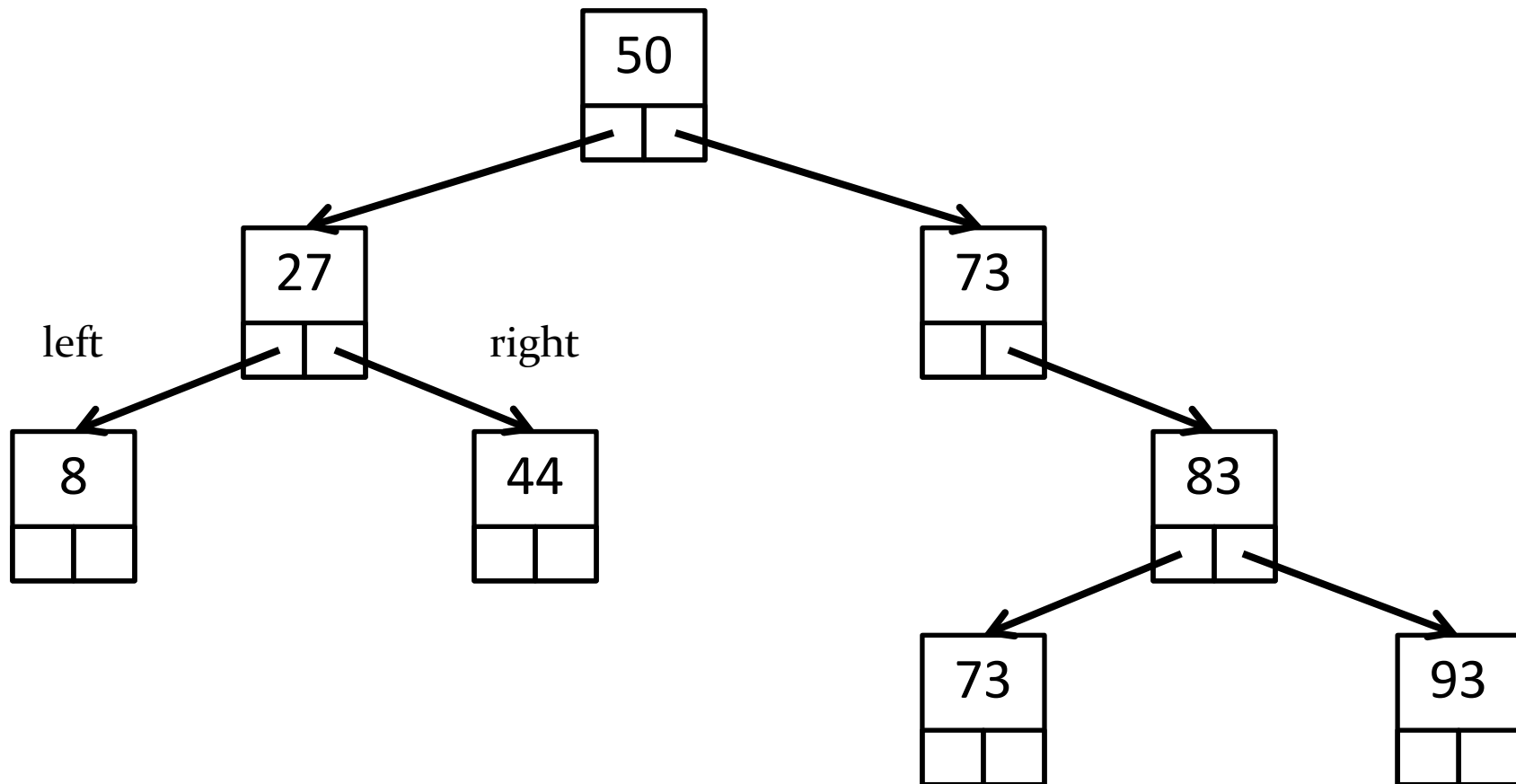
# Binary Tree

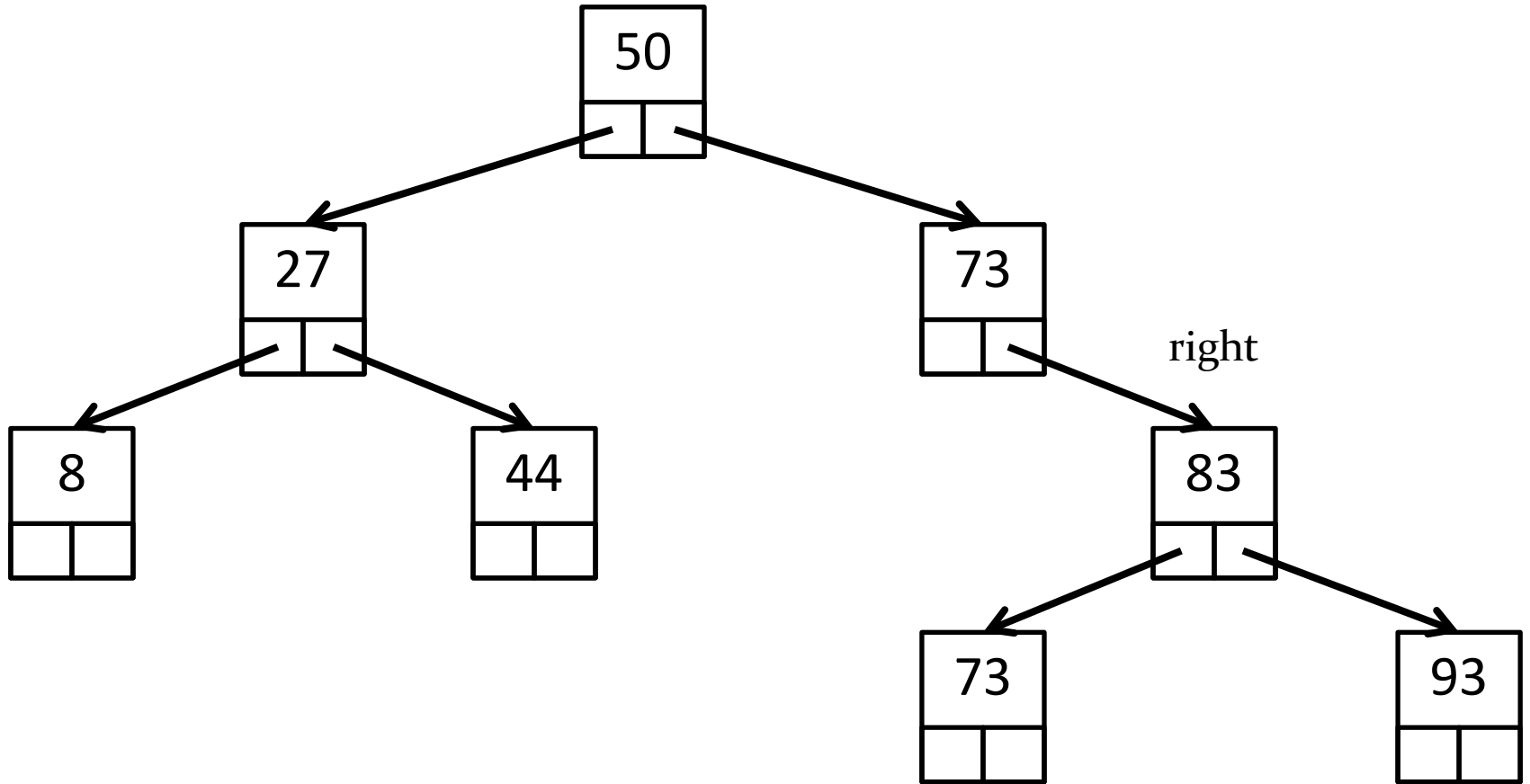
---

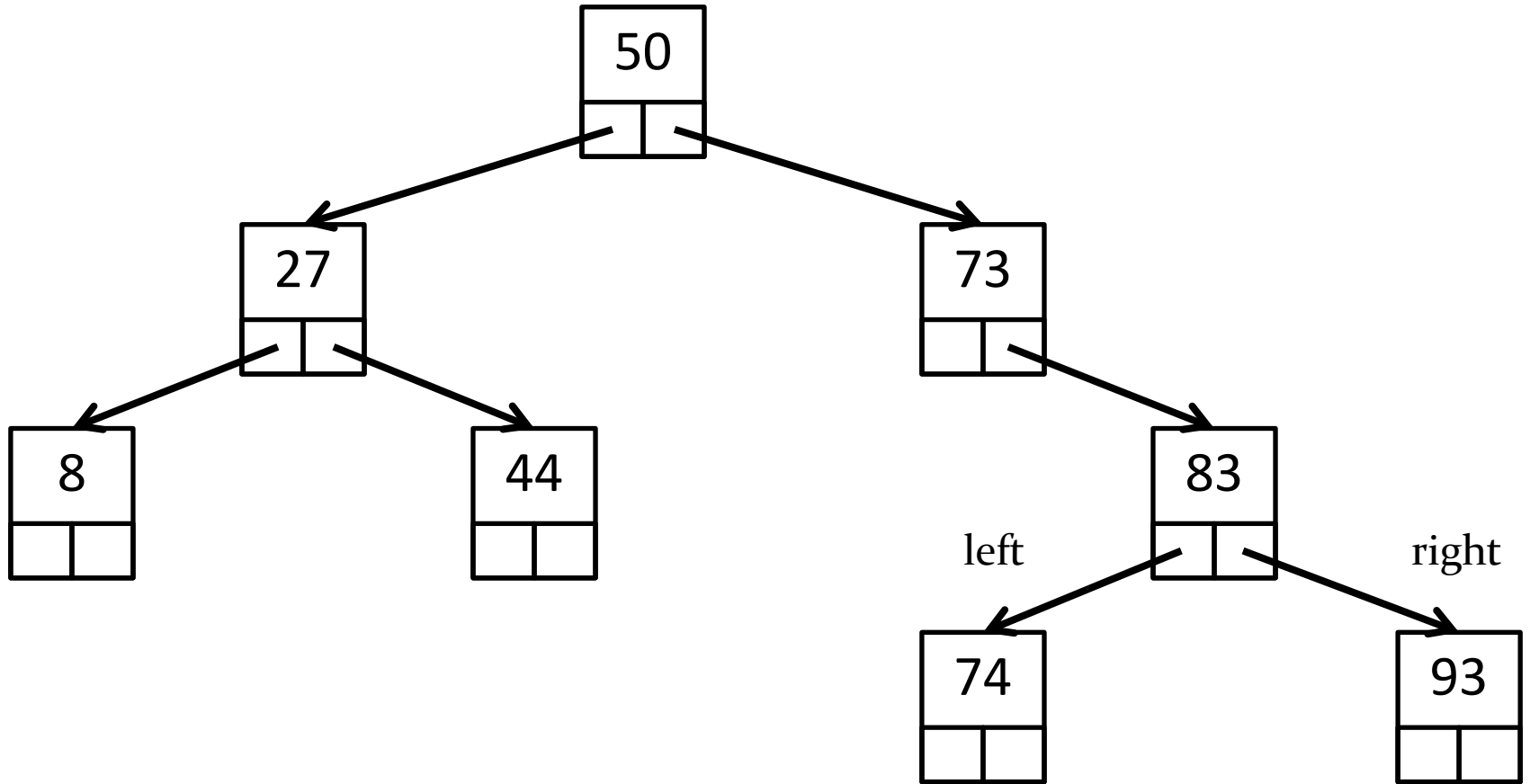
- ▶ a binary tree is a tree where each node has at most two children
  - ▶ very common in computer science
  - ▶ many variations
- ▶ traditionally, the children nodes are called the left node and the right node











# Binary Tree Algorithms

---

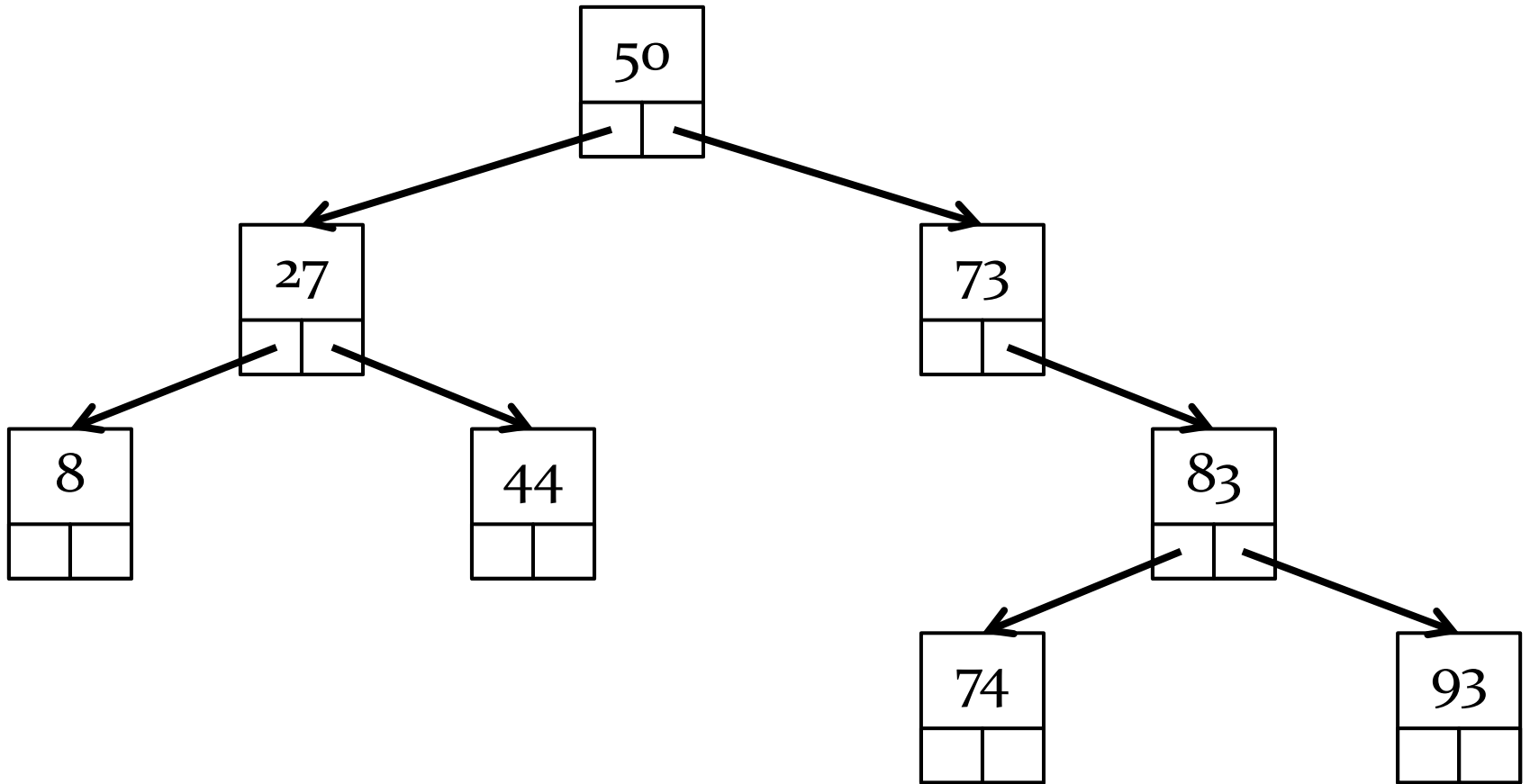
- ▶ the recursive structure of trees leads naturally to recursive algorithms that operate on trees
- ▶ for example, suppose that you want to search a binary tree for a particular element

```
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

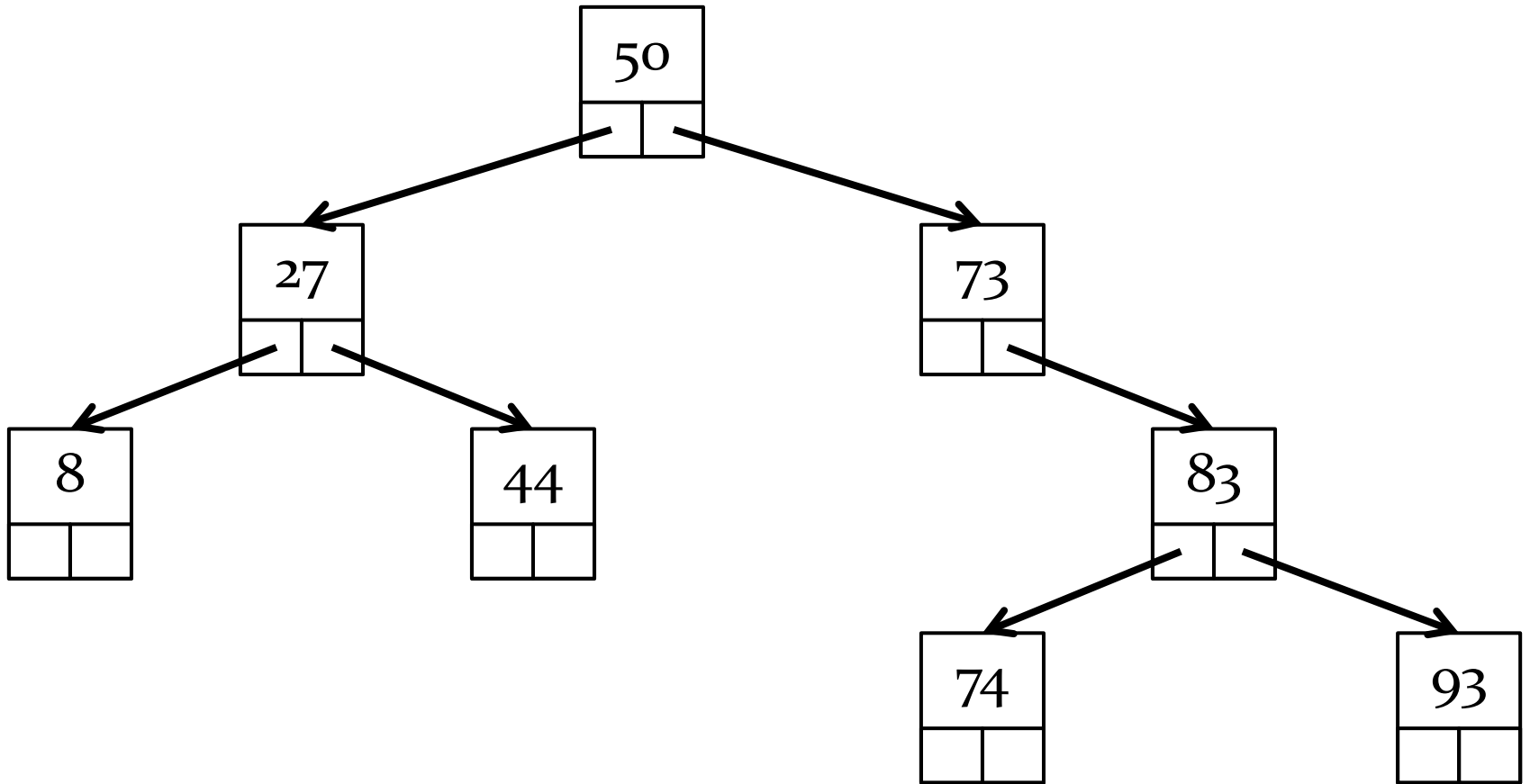
---

- ▶ 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, 74, 83, 93

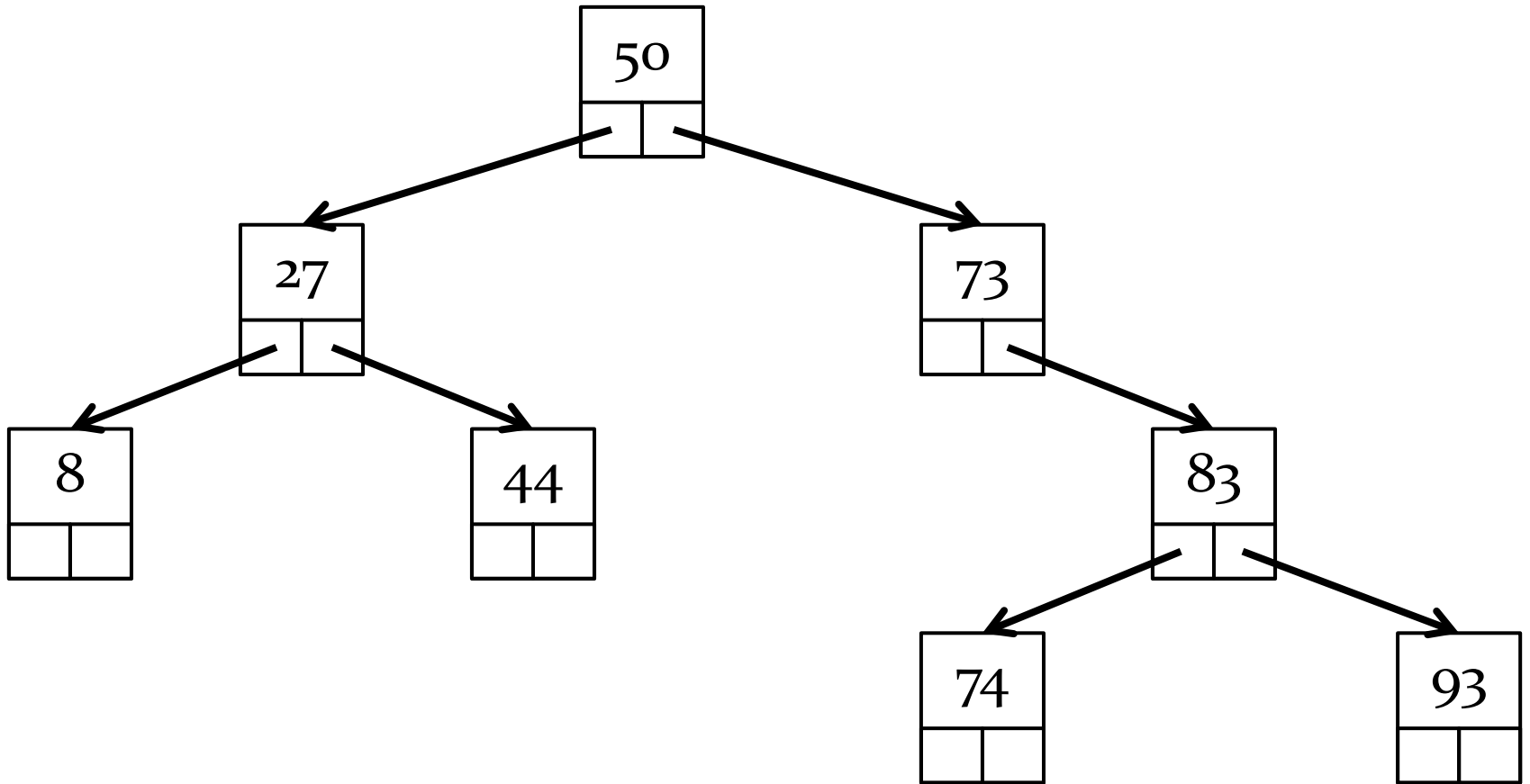




preorder: 50, 27, 8, 44, 73, 83, 74, 93







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

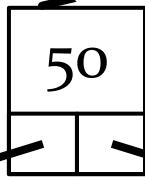


# Binary Search Trees (BST)

---

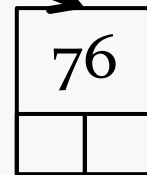
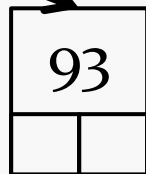
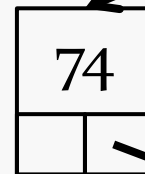
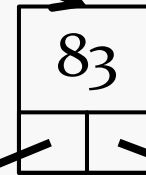
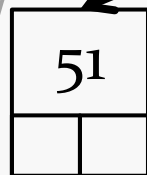
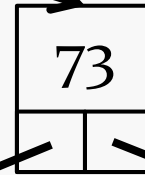
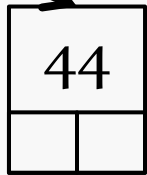
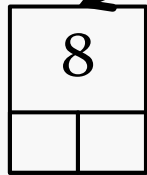
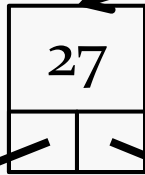
- ▶ the tree from the previous slide is a special kind of binary tree called a *binary search tree*
- ▶ in a binary search tree:
  1. all nodes in the left subtree have data elements that are less than the data element of the root node
  2. all nodes in the right subtree have data elements that are greater than the data element of the root node
  3. rules 1 and 2 apply recursively to every subtree

rest of tree  
not shown



right subtree

left subtree

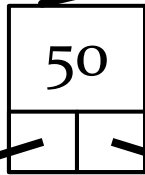


# Predecessors and Successors in a BST

---

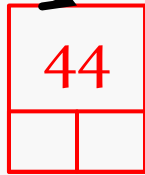
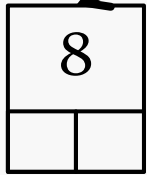
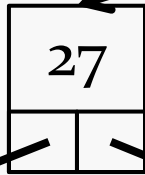
- ▶ in a BST there is something special about a node's:
  - ▶ left subtree right-most child
  - ▶ right subtree left-most child

rest of tree  
not shown

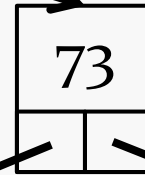


right subtree

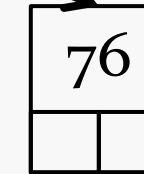
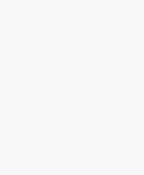
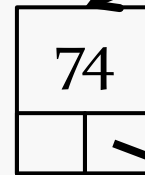
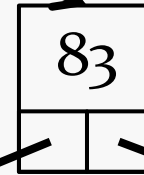
left subtree



rightmost  
child



leftmost  
child



rightmost child = inorder predecessor

leftmost child = inorder successor



# Deletion from a BST

---

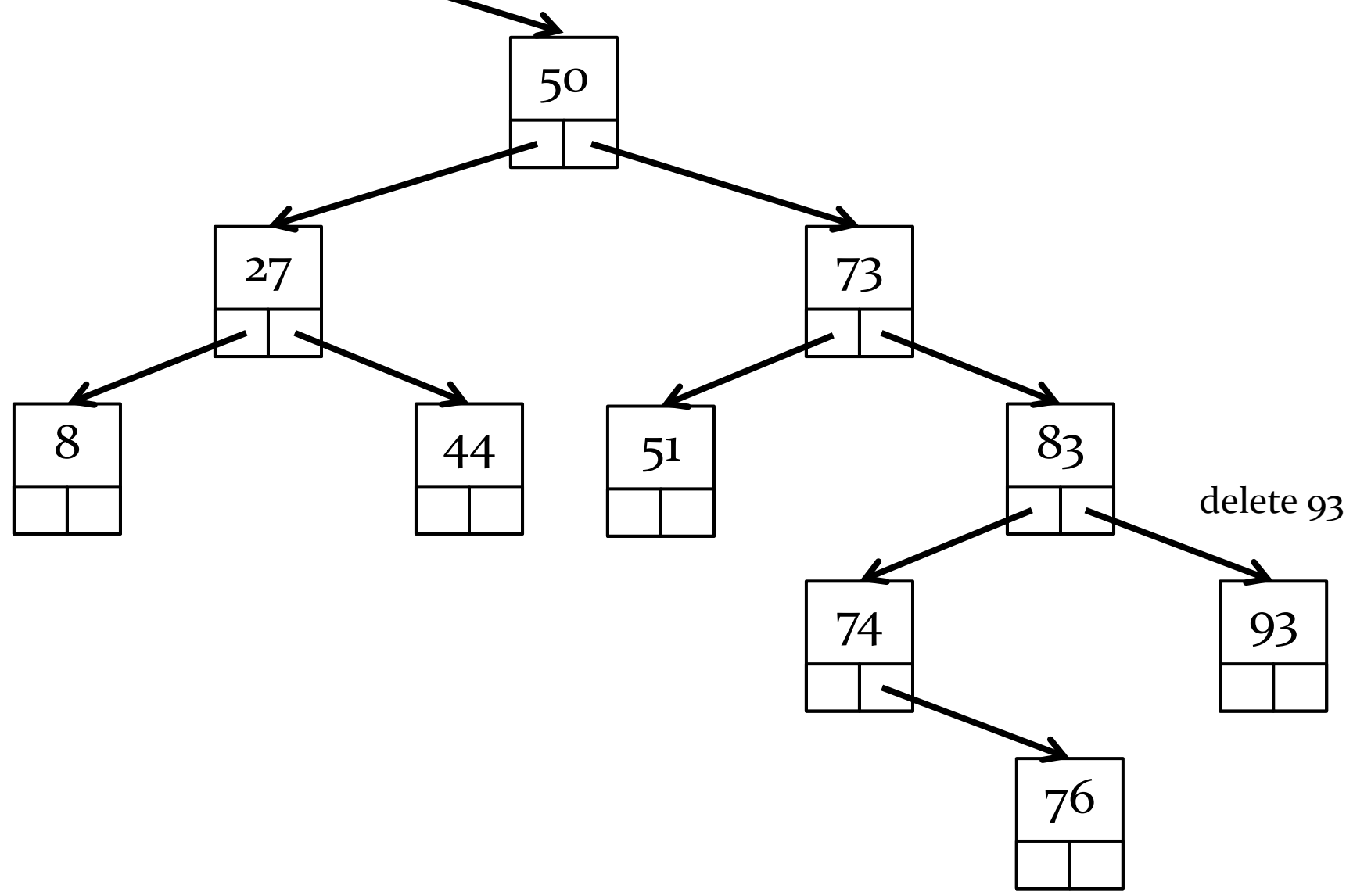
- ▶ to delete a node in a BST there are 3 cases to consider:
  1. deleting a leaf node
  2. deleting a node with one child
  3. deleting a node with two children

# Deleting a Leaf Node

---

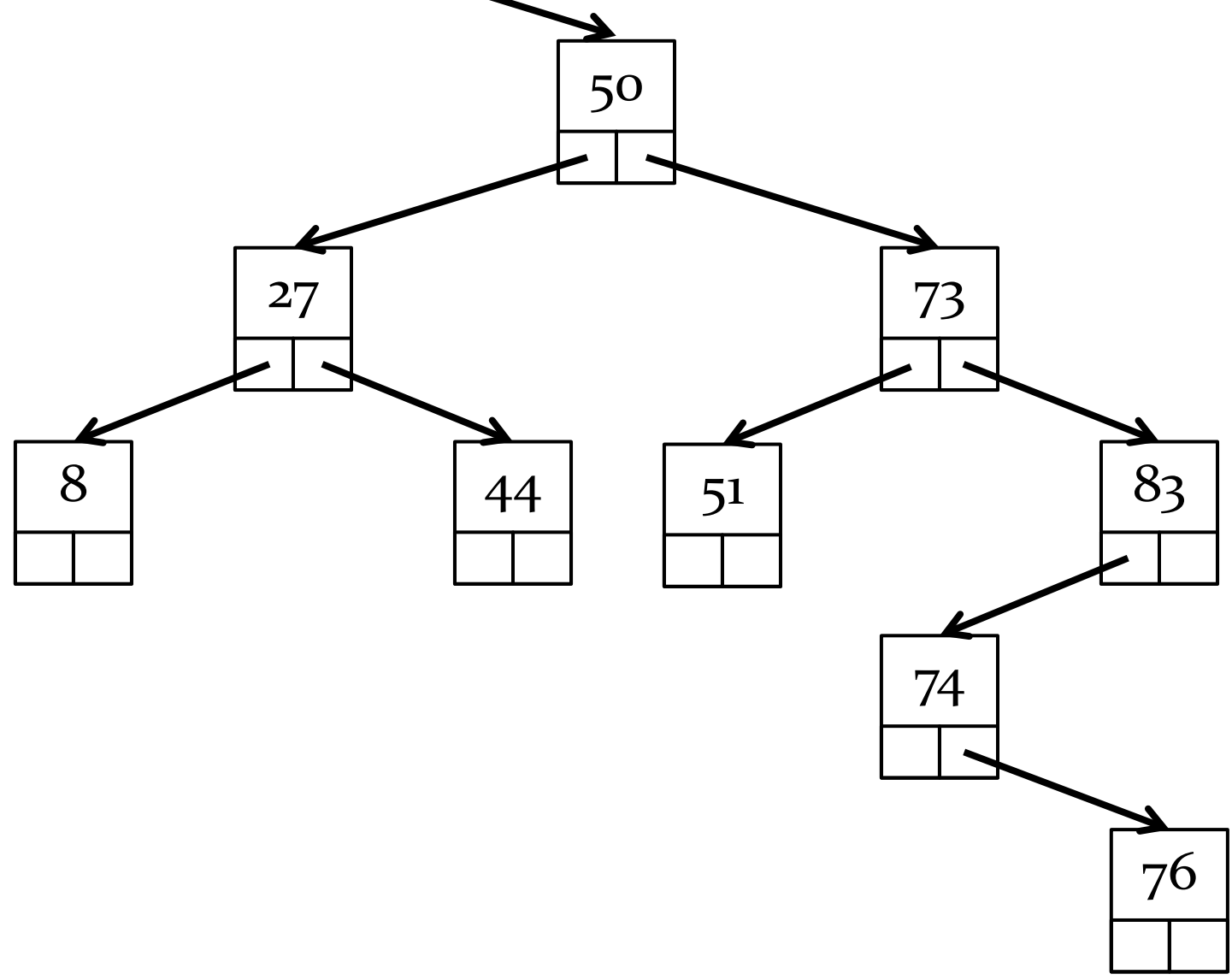
- ▶ deleting a leaf node is easy because the leaf has no children
  - ▶ simply remove the node from the tree
  
- ▶ e.g., delete 93

rest of tree  
not shown





rest of tree  
not shown

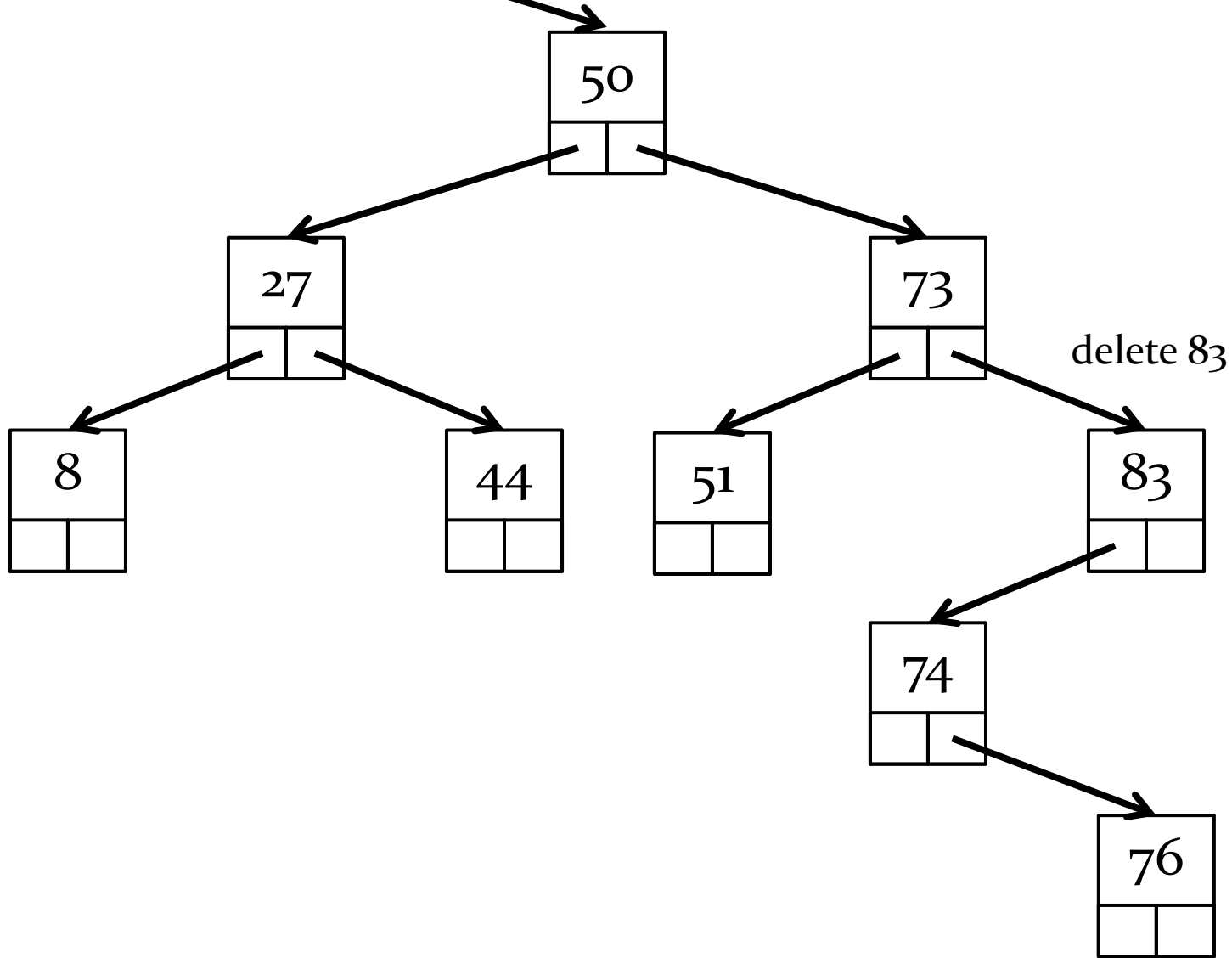


# Deleting a Node with One Child

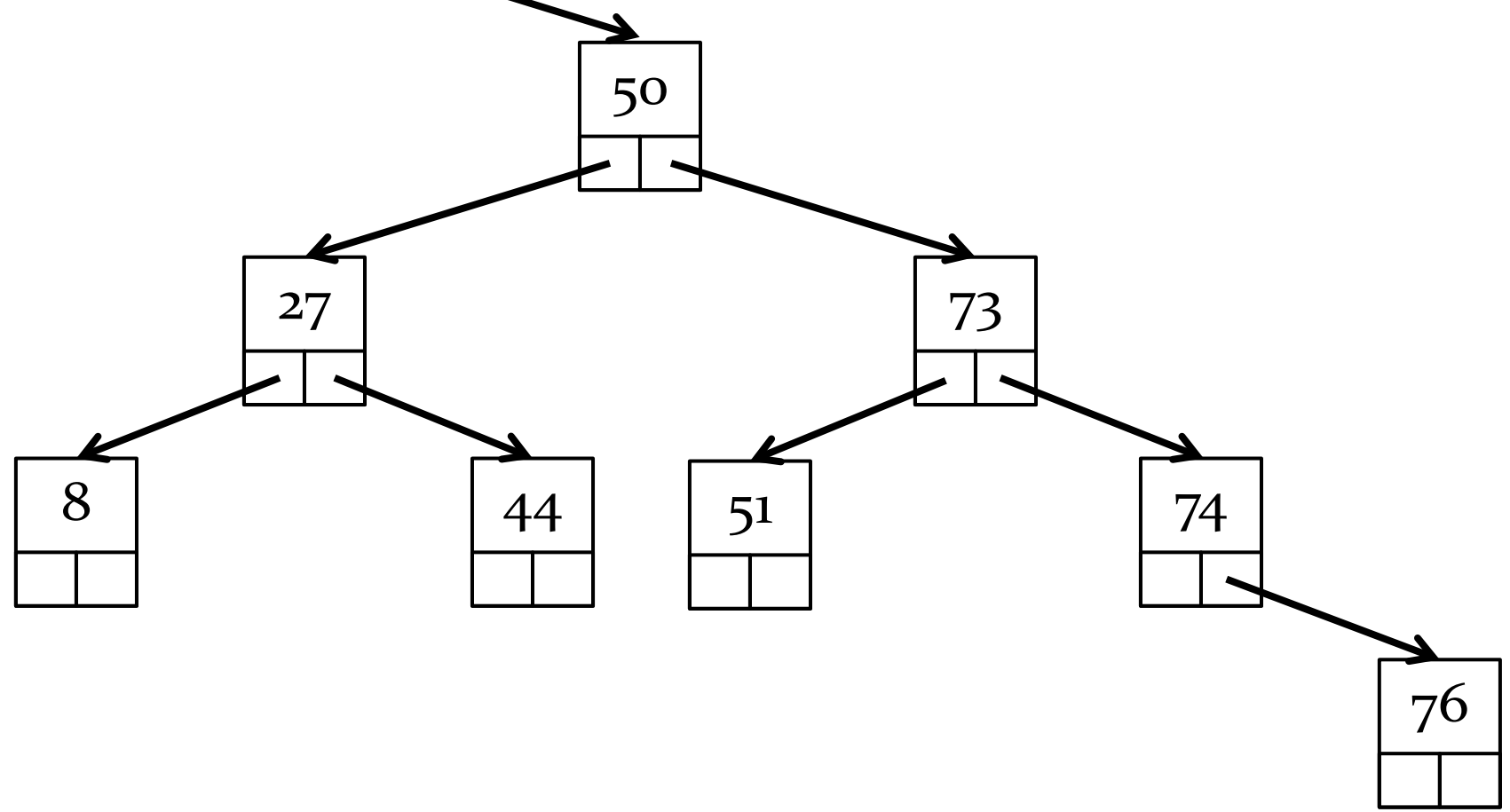
---

- ▶ 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



rest of tree  
not shown



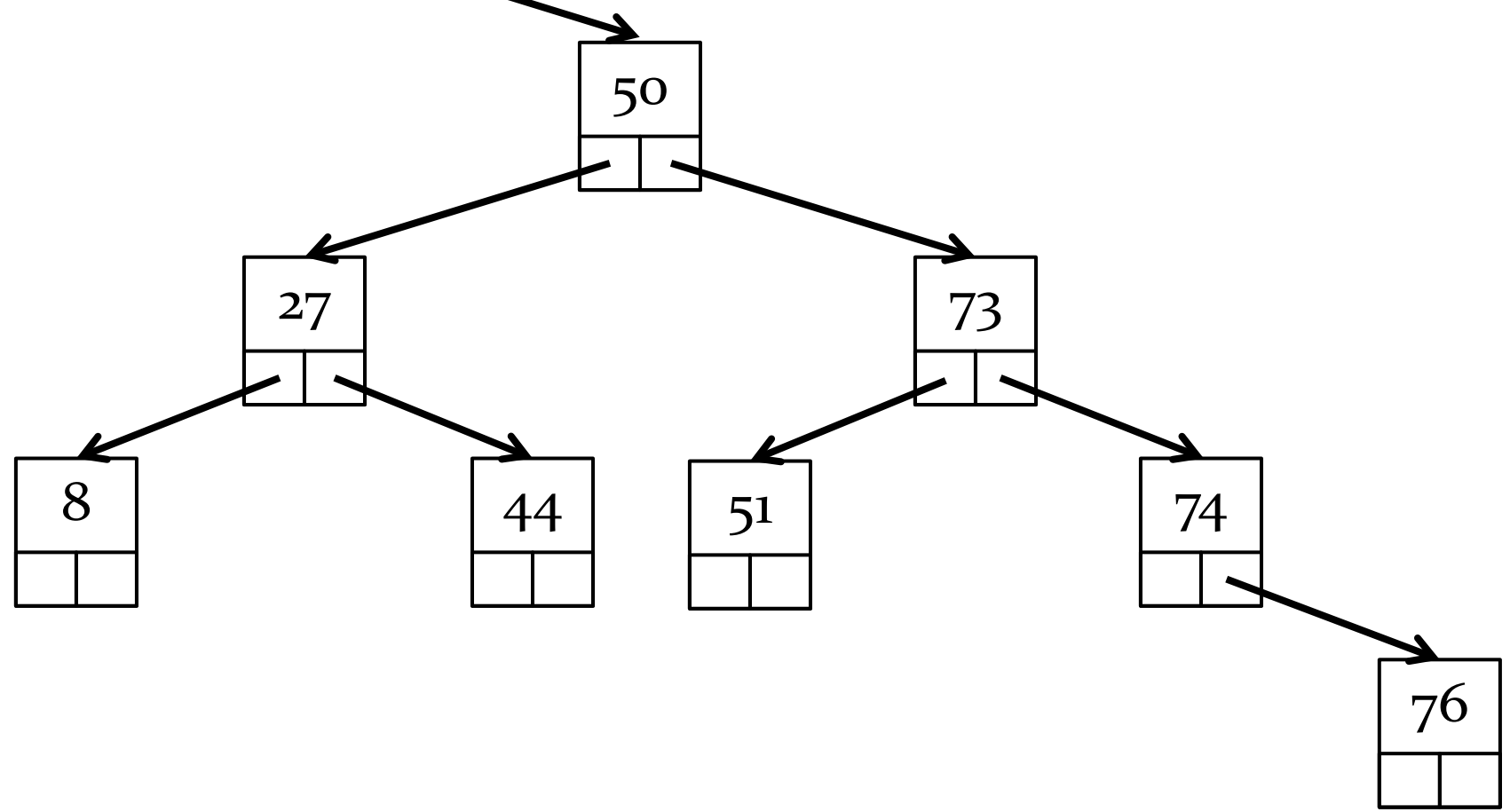
# Deleting a Node with Two Children

---

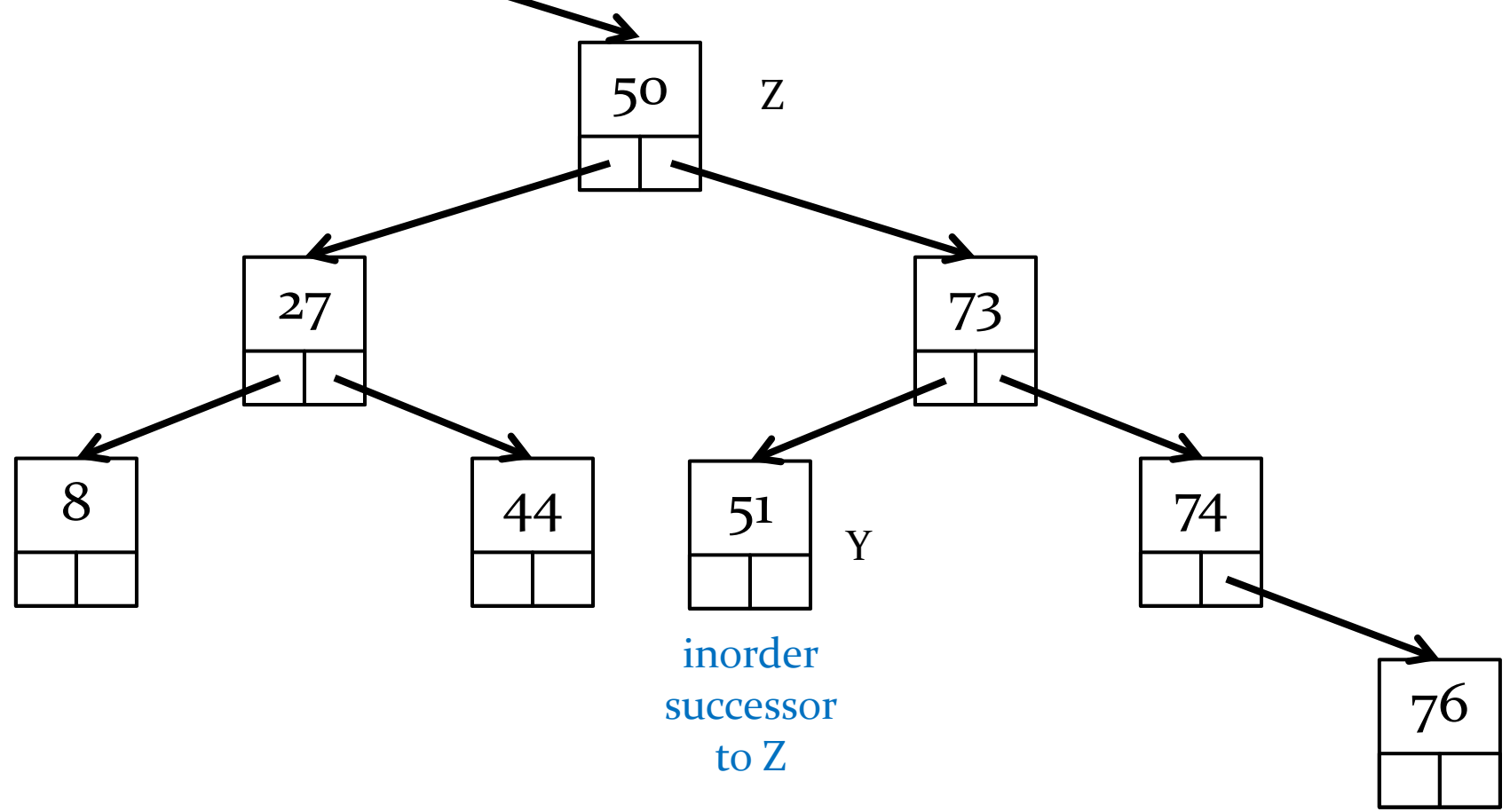
- ▶ deleting a node with two children is a little trickier
  - ▶ call the node to be deleted Z
  - ▶ find the inorder predecessor OR the inorder successor
    - ▶ call this node Y
      - if the inorder predecessor does not exist, then you must find the inorder successor (and vice versa)
  - ▶ copy the data element of Y into the data element of Z
  - ▶ delete Y
  
- ▶ e.g., delete 50

rest of tree  
not shown

delete 50

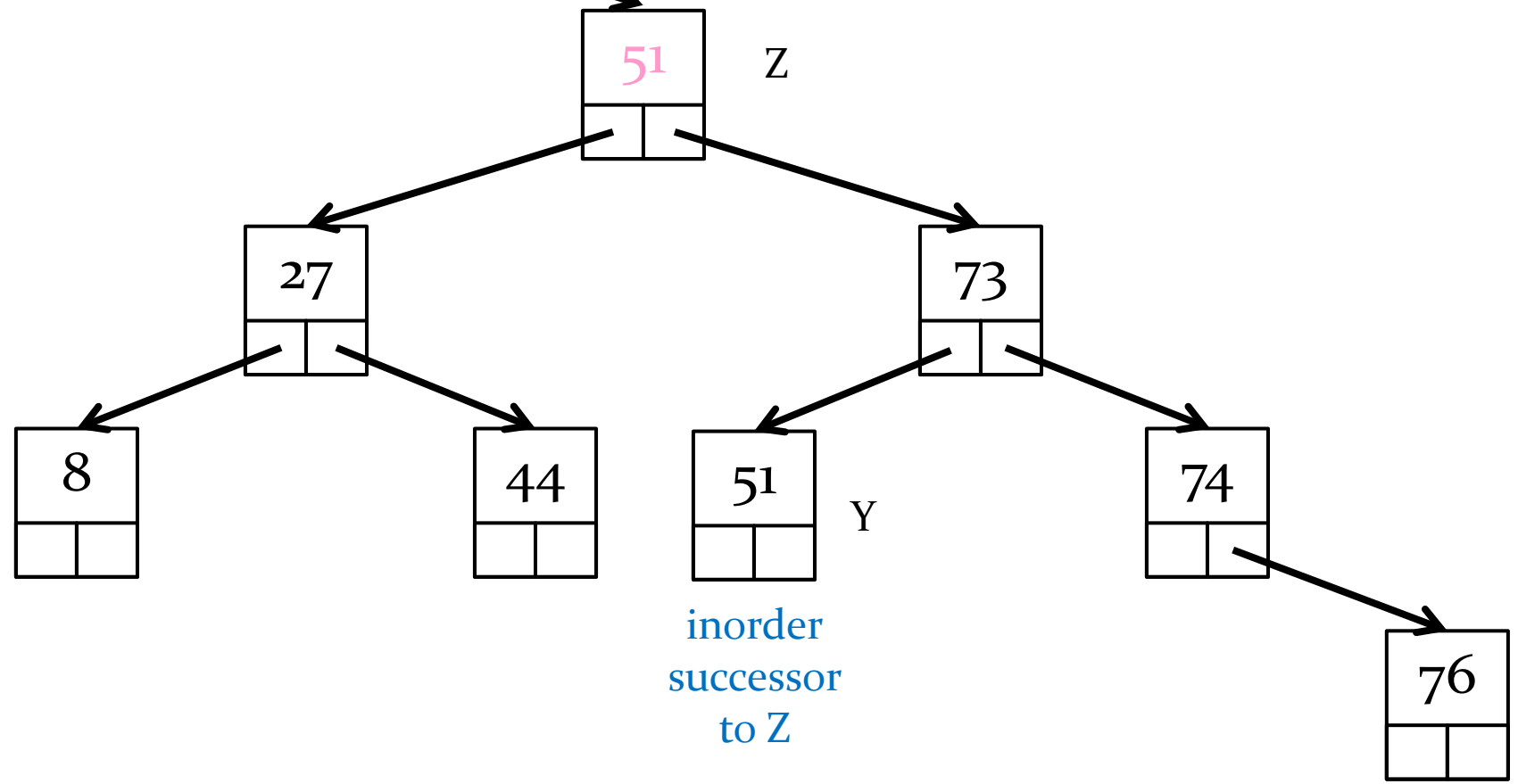


rest of tree  
not shown



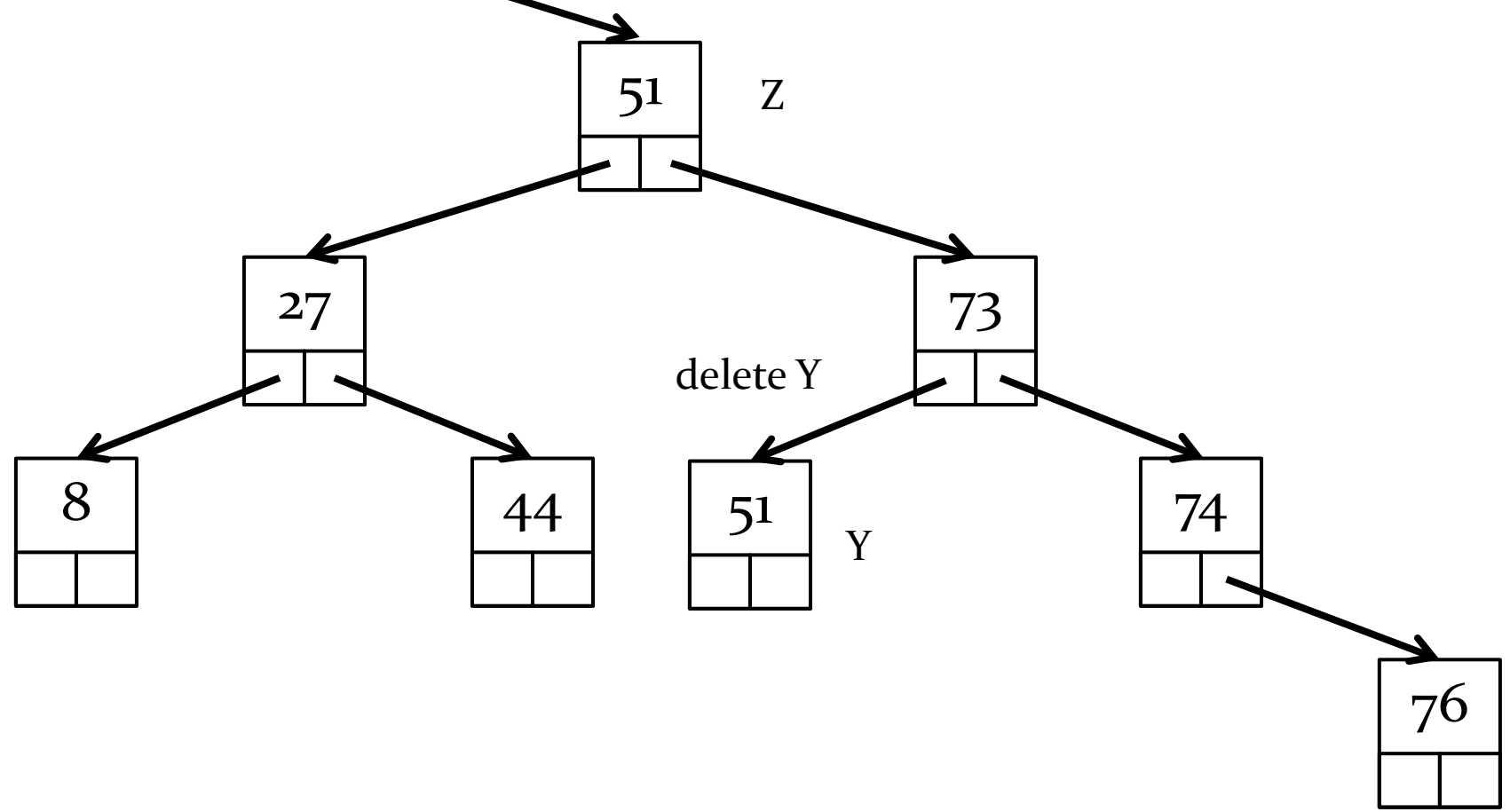
rest of tree  
not shown

copy Y data to Z data





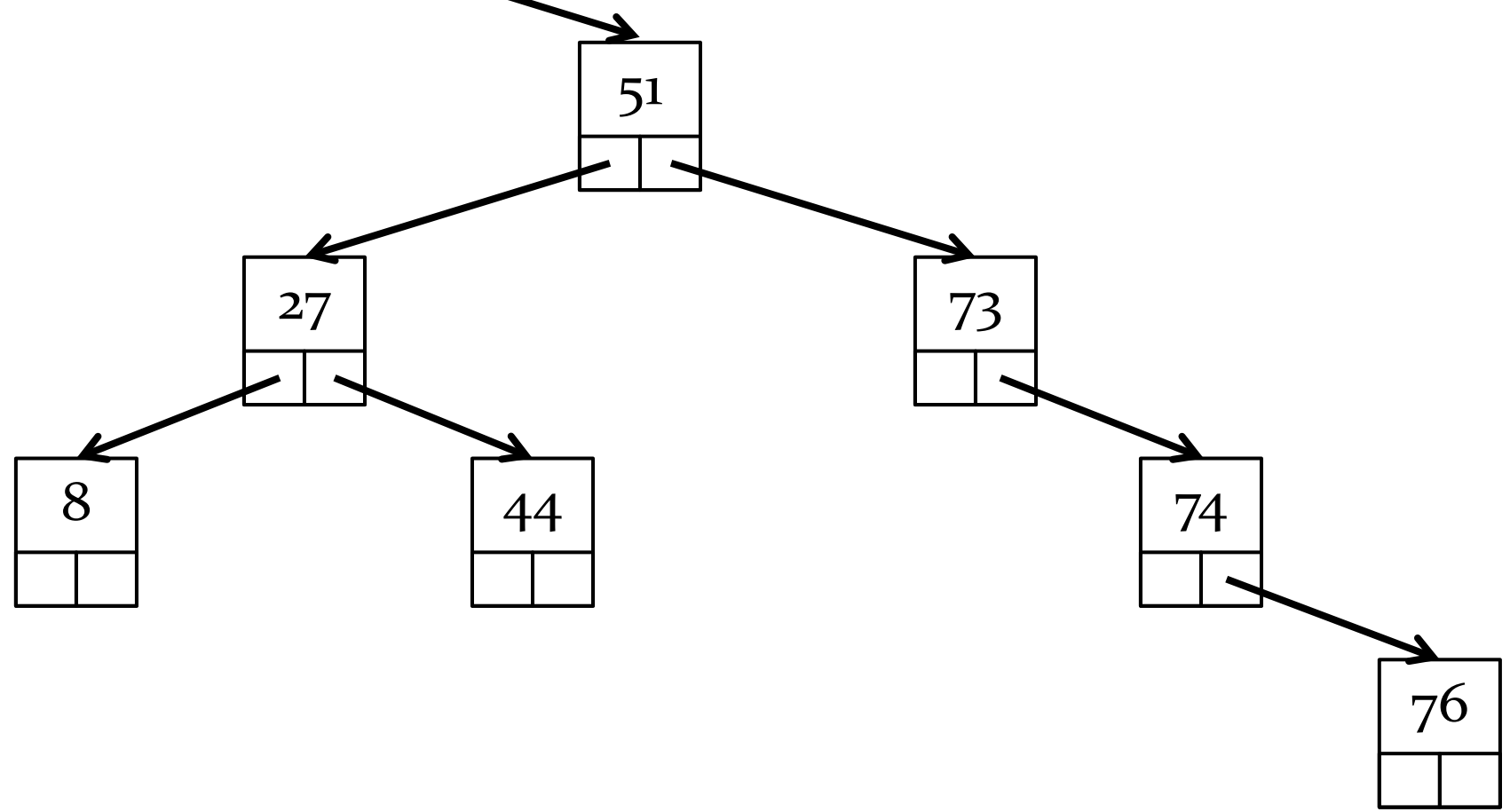
rest of tree  
not shown



delete Y



rest of tree  
not shown



# Recursion and Data Structures in Computer Graphics

Ray Tracing

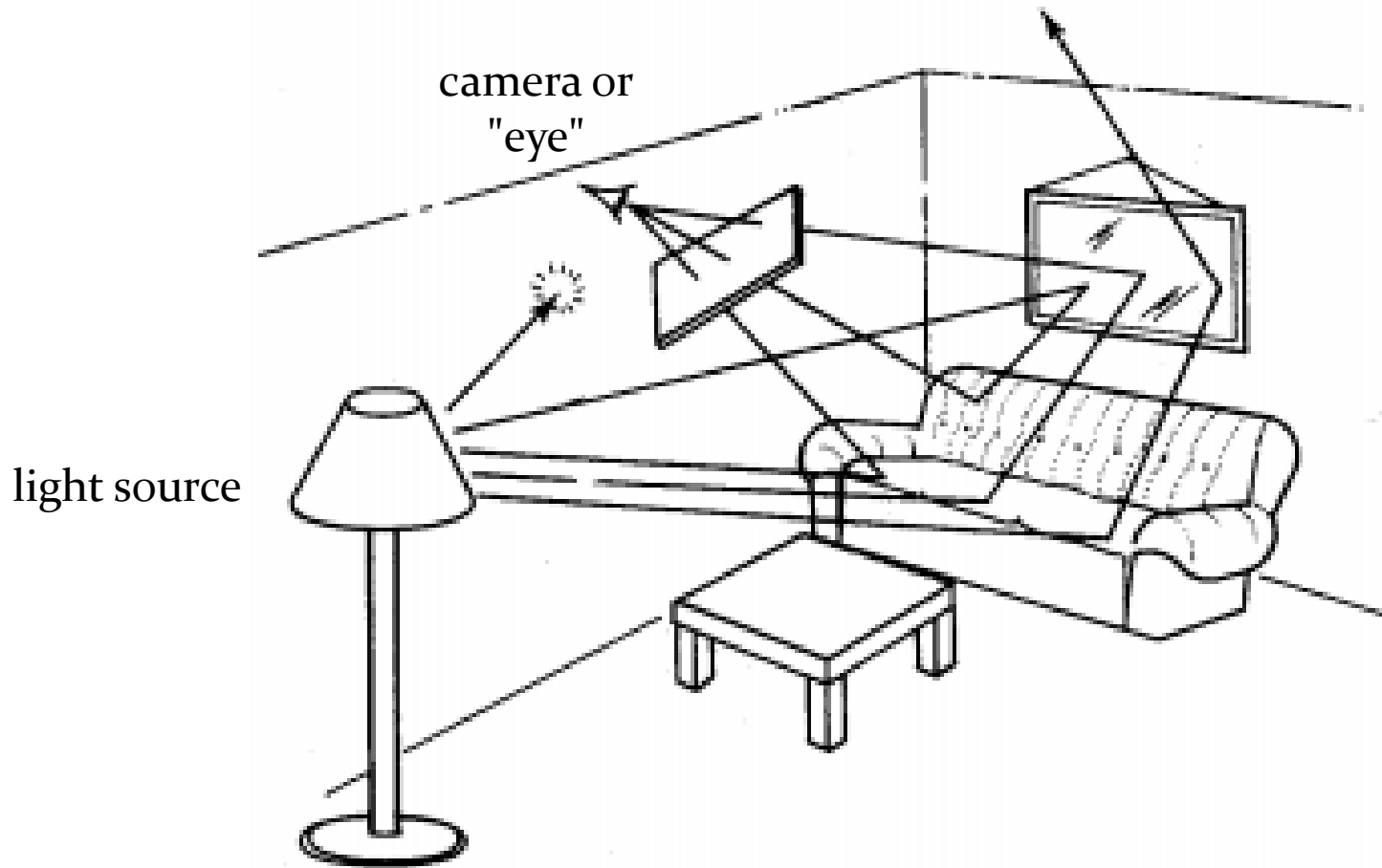
# Forward Ray Tracing

---

- ▶ imagine that you take a picture of a room using a camera
- ▶ exactly what is the camera sensing?
  - ▶ light reflected from the surfaces of objects into the camera lens

# Forward Ray Tracing

---



# Forward Ray Tracing

---

- ▶ forward ray tracing traces the paths of light from the light source to the camera to produce an image
- ▶ computationally infeasible because almost all of the possible paths of light miss the camera

# Backward Ray Tracing

---

- ▶ backward ray tracing traces the paths of light from the camera out into the environment to produce an image
- ▶ computationally feasible because the process starts with a single\* ray per screen pixel

# Why Ray Tracing: Shadows

---



shadows with ~1000 light sources



# Ray Tracing: Reflections

---



# Ray Tracing: Reflections

---



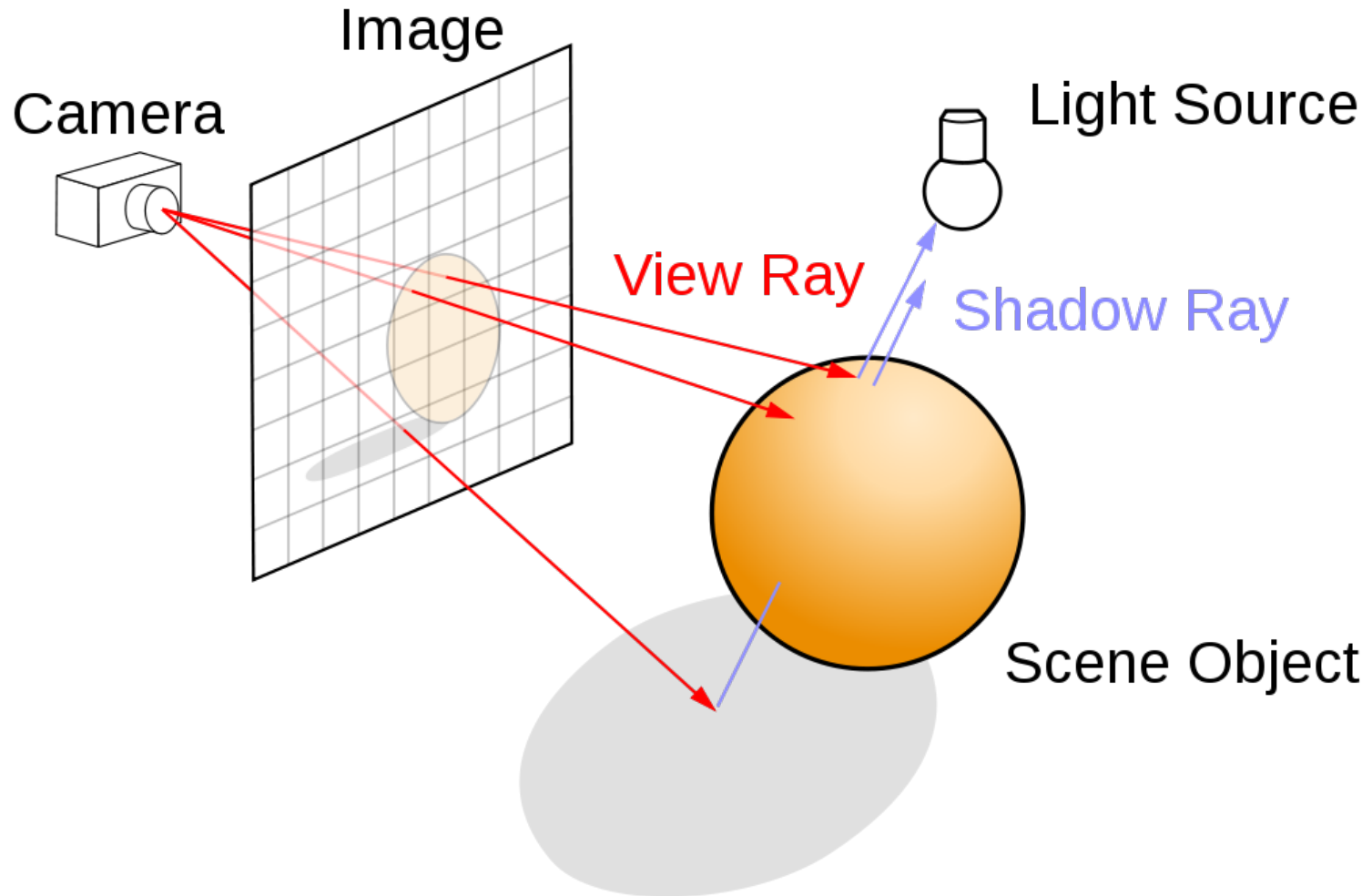
# Comment on Previous Images

---

- ▶ most of the rendering in the previous images was not done using ray tracing
- ▶ ray tracing was only used on those parts of the image that would produce a noticeable difference

# Backward Ray Tracing

---



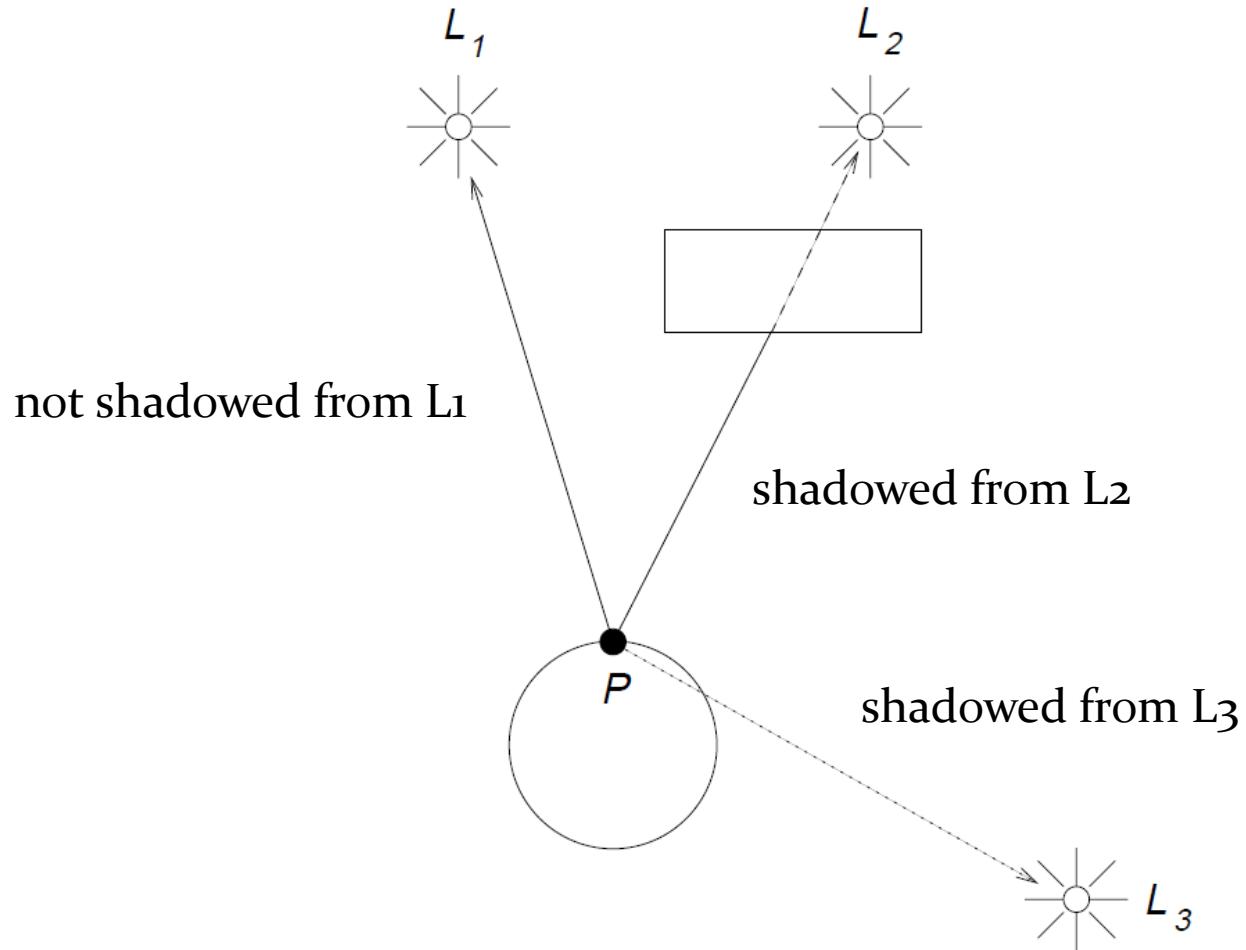
# Shadows

---

- ▶ we can determine if a point is in shadow by tracing rays from the point to each light source
  - ▶ called shadow rays
- ▶ if a shadow ray hits an object before it reaches the light source then the point is in shadow

# Shadows

---



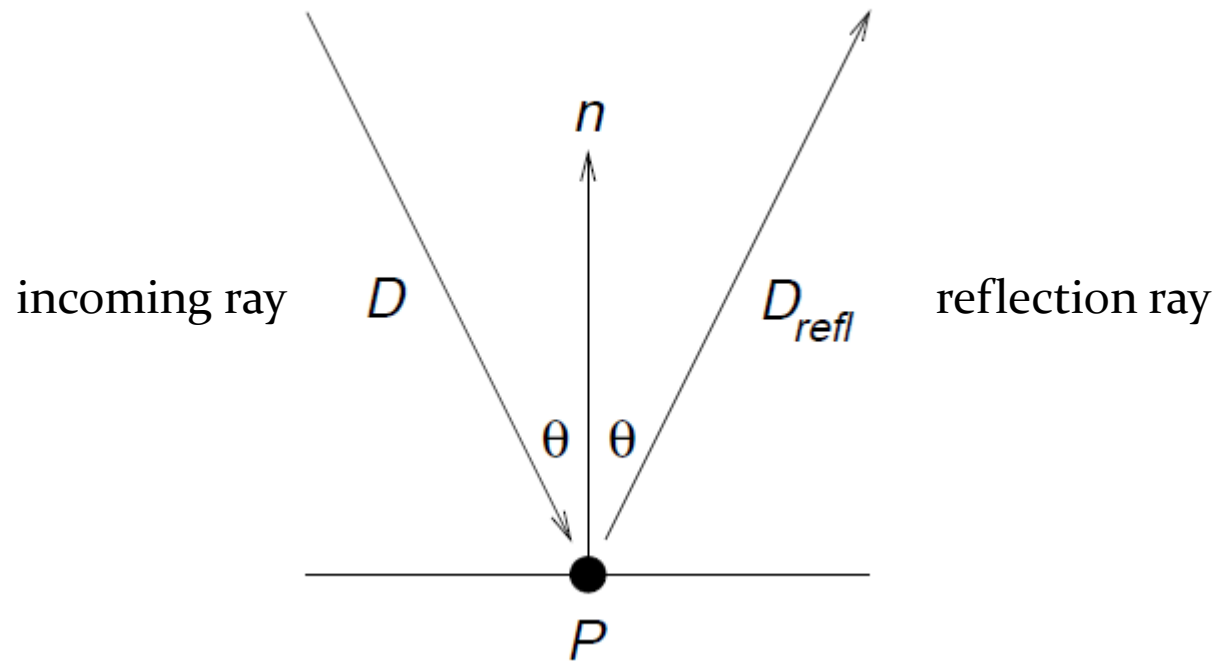
# Reflections

---

- ▶ if the ray hits a shiny object then we would like to know what reflection is seen at the hit point
- ▶ we can cast a new ray in the mirror reflection direction to determine the reflection

# Reflections

---





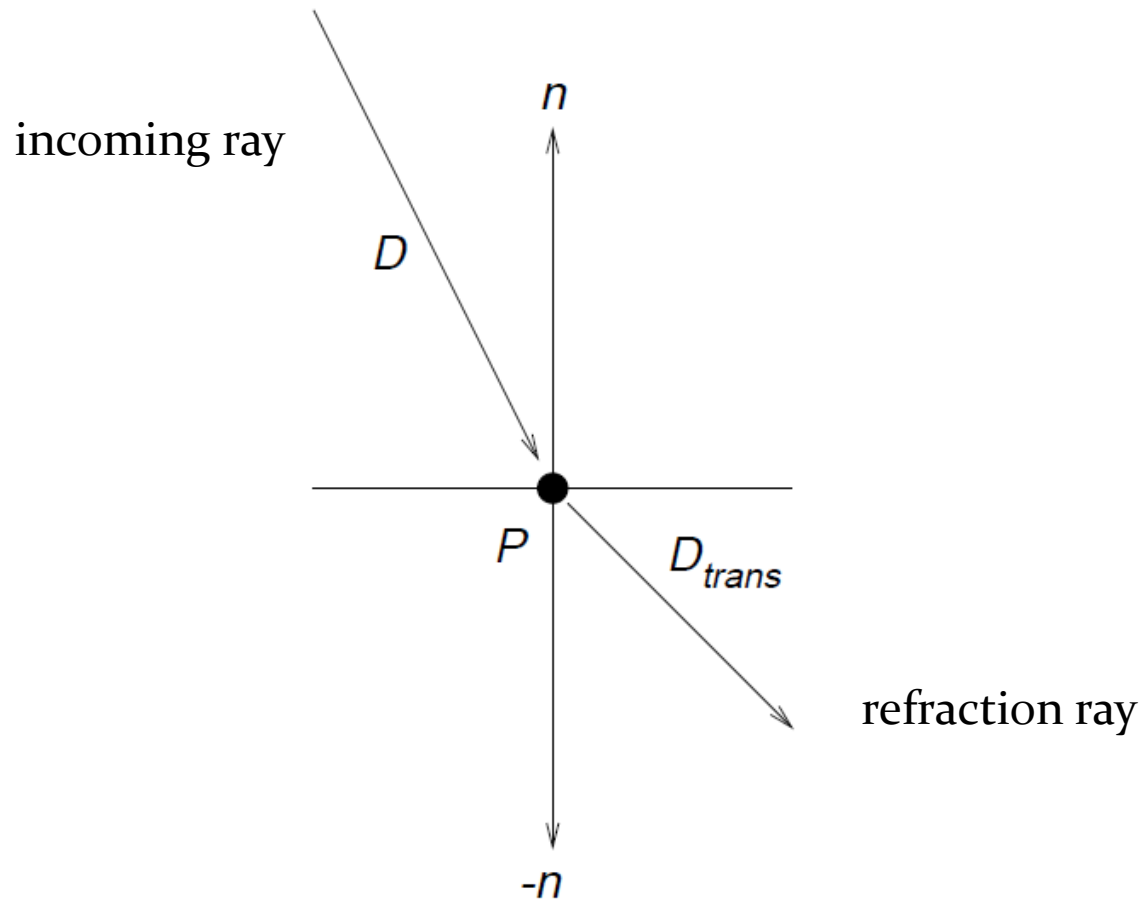
# Transparent Objects

---

- ▶ if the ray hits a transparent object then we would like to know what can be seen through the object
- ▶ we can cast a new ray in the refraction direction to determine what can be seen through the object

# Transparent Objects

---

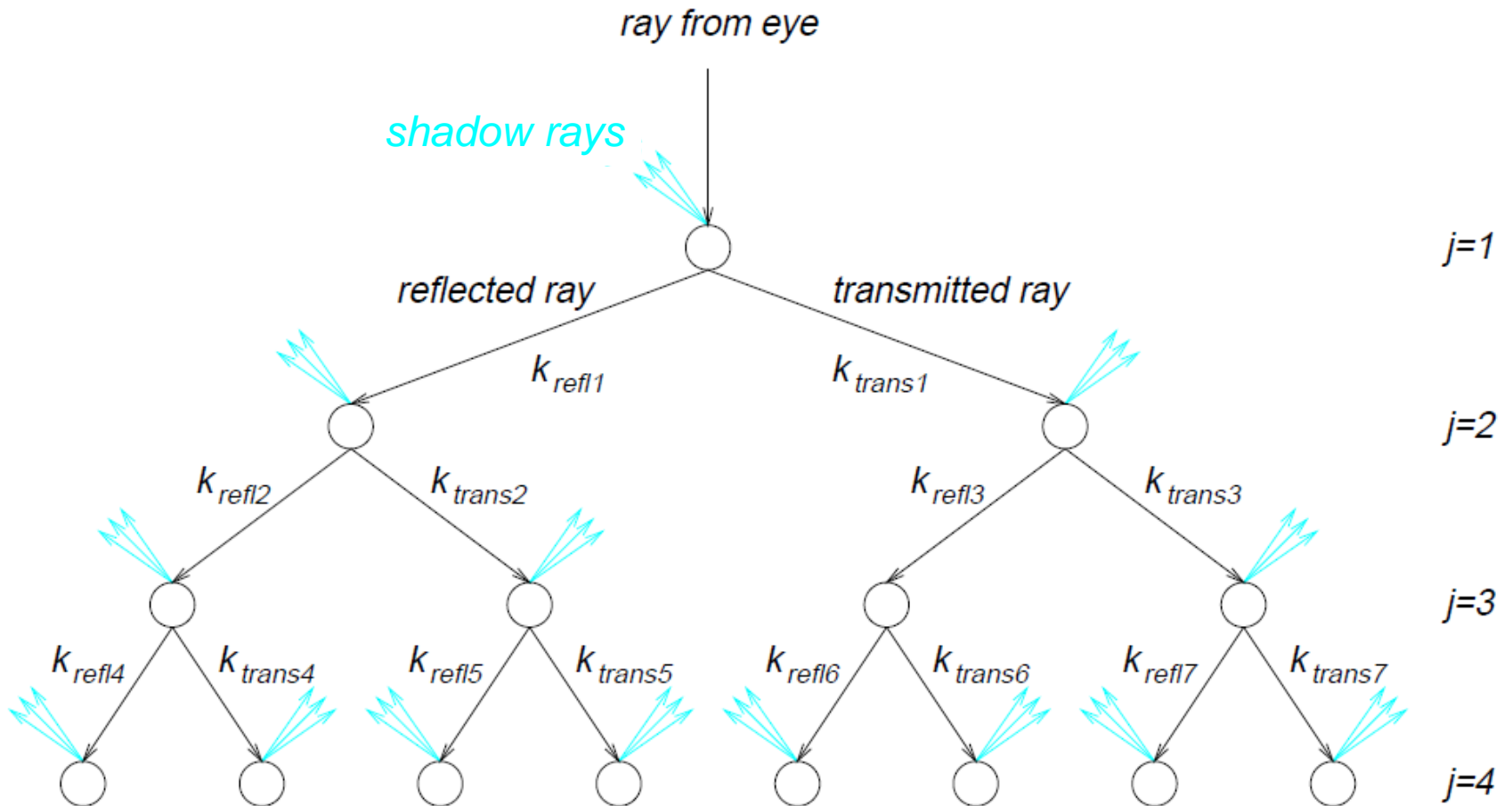


# Recursion

---

- ▶ each reflected and refracted ray can be treated as a new emanating from a hit point
  - ▶ i.e., we recursively trace the reflected and refracted rays

# Ray Tracing as a Binary Tree



# Stopping the Recursion

---

- ▶ what are the base cases?
  - ▶ ray misses all objects
  - ▶ level of recursion exceeds a fixed value
  - ▶ other cases outside the scope of CSE1030

# How Fast is Ray Tracing

---

- ▶ approaching real time for non-cinematic quality, e.g.,
  - ▶ Brigade 2 game engine
  - ▶ NVIDIA OptiX
    - ▶ [demos here if you have a high-end NVIDIA graphics card](#)
- ▶ cinematic quality is much slower

# How Fast is Ray Tracing

---

- ▶ 678 million triangles
- ▶ rays
  - ▶ 111 million diffuse
  - ▶ 37 million specular
  - ▶ 26 million shadow
- ▶ 1.2 billion ray-triangle intersections
- ▶ 106 minutes on 2006 hardware



# Bounding Volumes

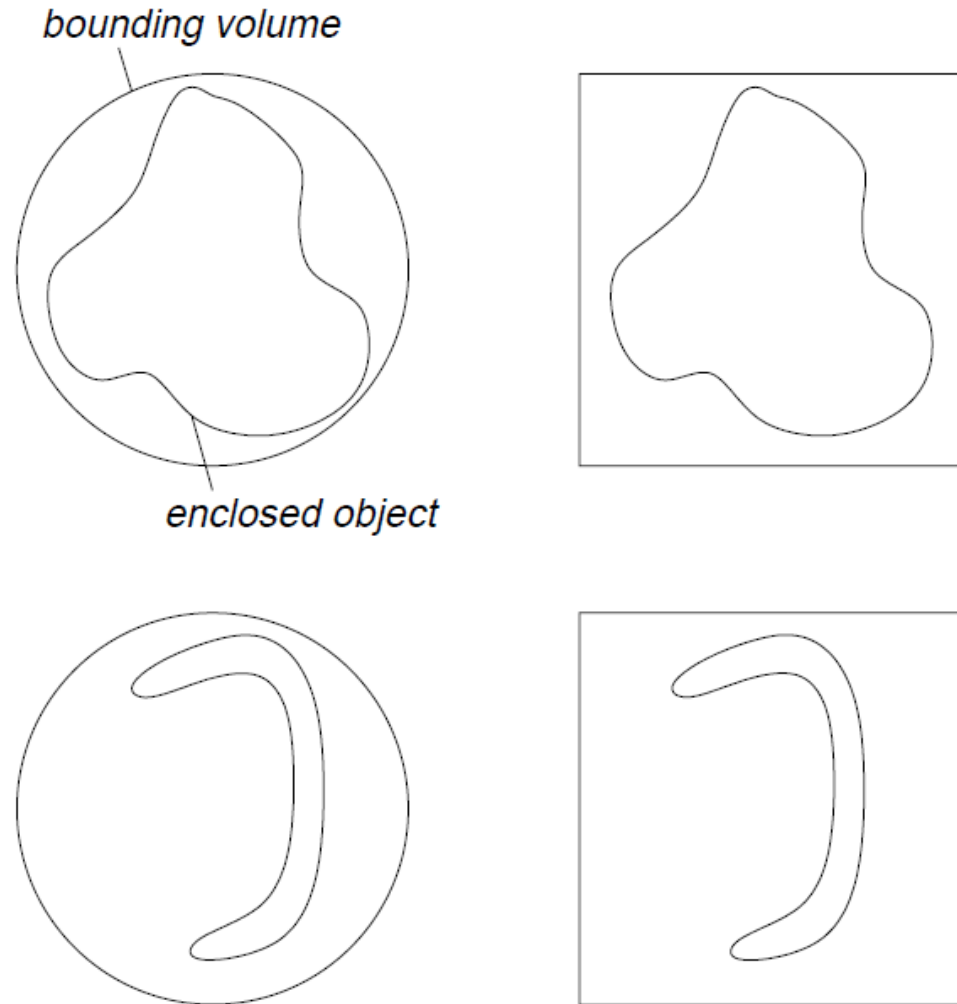
---

- ▶ it is easy to compute the intersection of a ray with certain shapes, e.g.,
  - ▶ spheres and cubes
- ▶ it is hard or expensive to compute the intersection of a ray with arbitrary shapes
- ▶ idea
  - ▶ put complex shapes inside simple ones



# Bounding Volumes

---



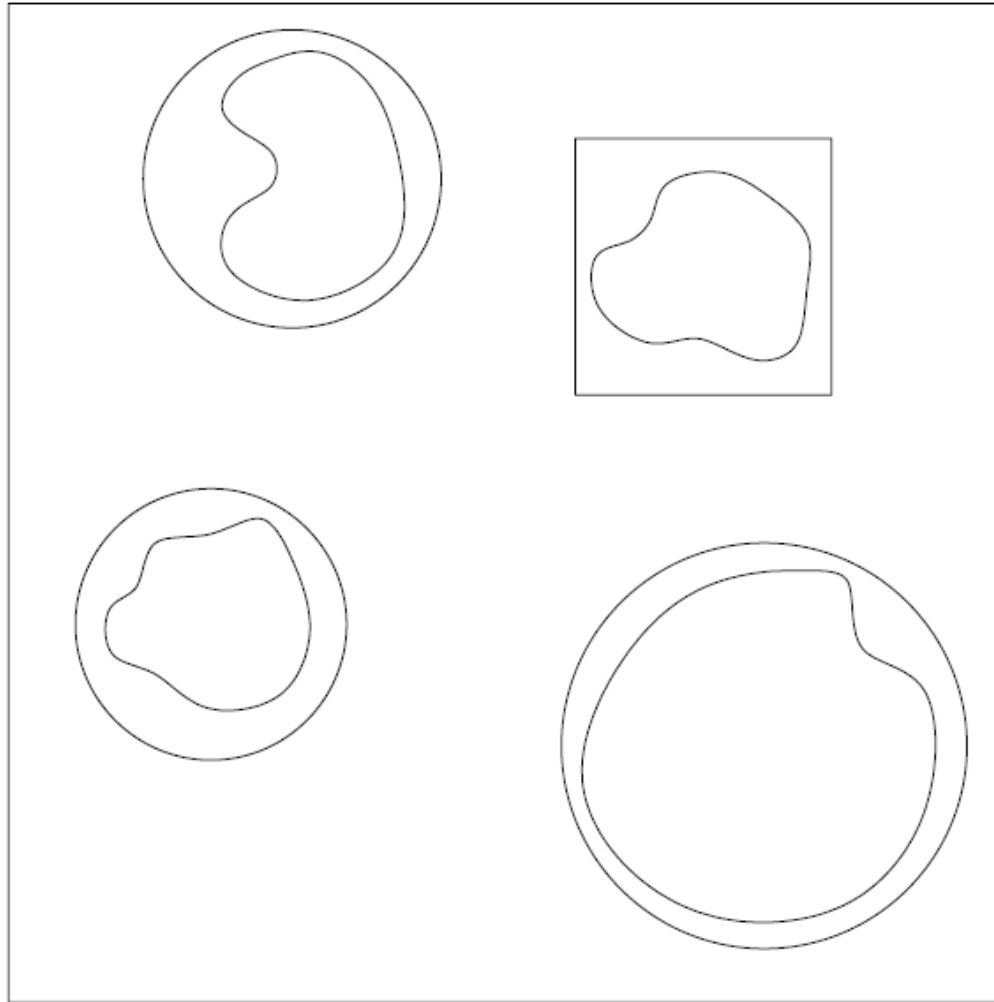
# Hierarchy of Bounding Volumes

---

- ▶ why stop at putting complex shapes into bounding volumes?
- ▶ why not put bounding volumes inside bounding volumes?

# Hierarchy of Bounding Volumes

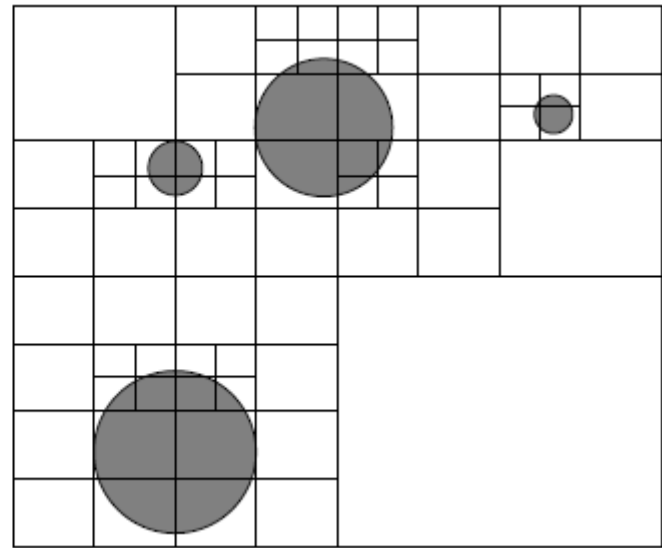
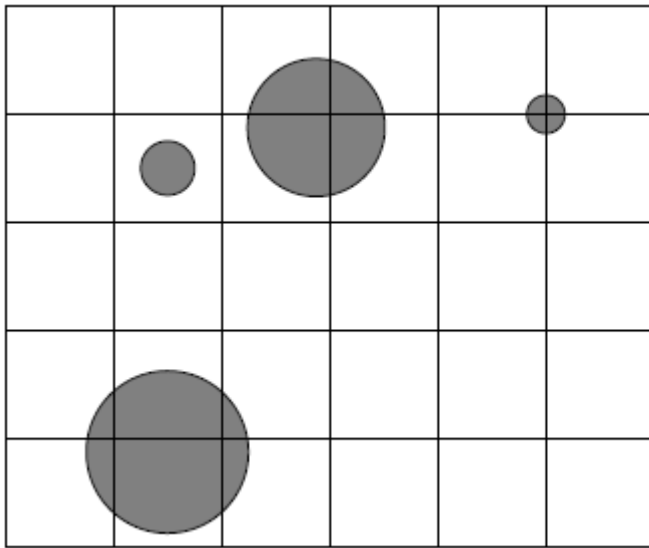
---



# Spatial Subdivision

---

- ▶ instead of putting objects inside volumes we can subdivide space



# Quadtree Decomposition

---

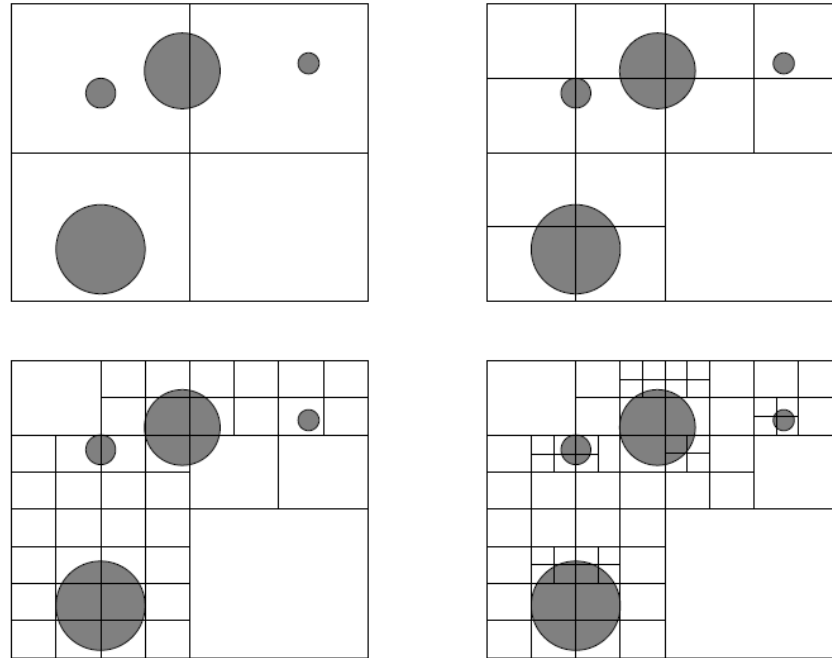


Figure 7.16: Quadtree decomposition of 2D space. Each quadrant of the world is recursively subdivided into subquadrants until the subquadrant meets some measure of simplicity. In this example, the recursive subdivision stops when then i) the subquadrant is occupied by zero objects or; ii) the object occupying the subquadrant takes up more than one-half the area of the subquadrant or; iii) the depth of recursion is four. A more common heuristic is to subdivide any quadrant that that is occupied by  $n$  or more objects.

# Using a Quadtree in Ray Tracing

---

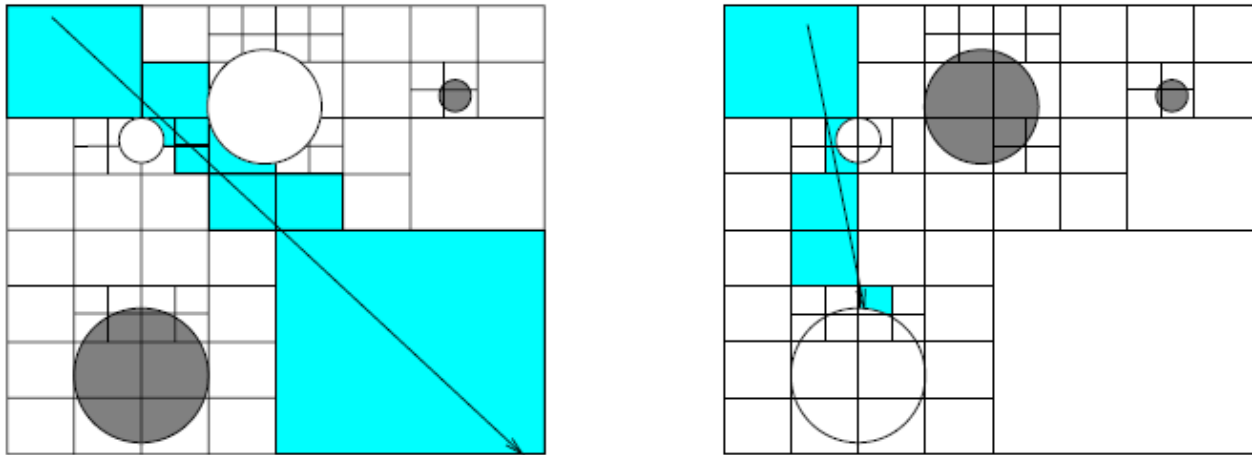


Figure 7.17: Using an octree to reduce the number of intersection calculations. On the left, nine voxels are examined and two objects are tested for an intersection with the ray. On the right, six voxels are examined and two objects are tested for an intersection with the ray. If an intersection is found, no more voxels need to be examined. This is because voxels are examined in the order that the ray passes through the world—the first intersection found must be the nearest hit point to the starting point of the ray.

# Open Source Ray Tracers

---

- ▶ Art of Illusion
- ▶ POV-Ray
- ▶ YafaRay
- ▶ Manta
- ▶ several others