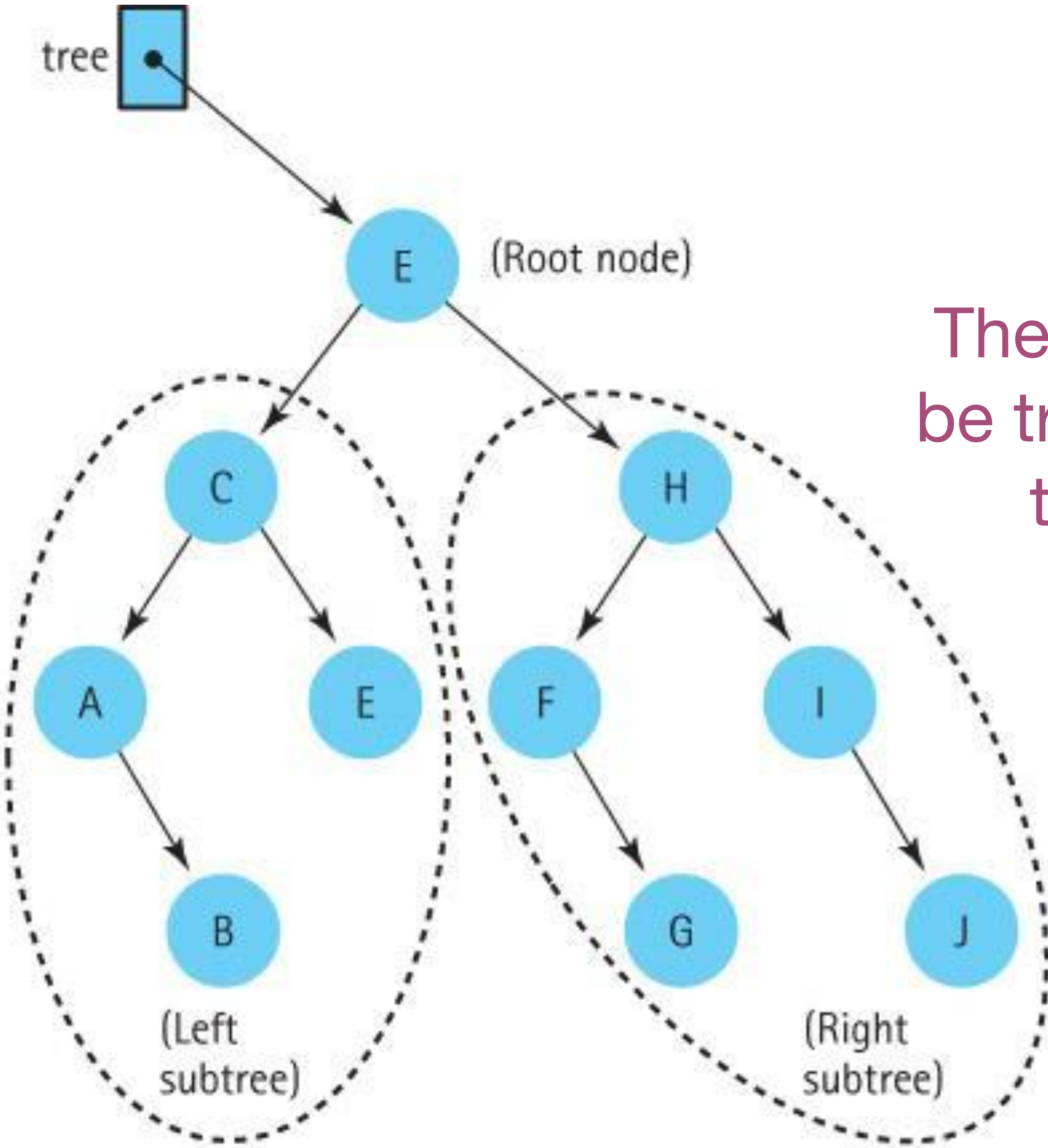
Binary Search Tree (BST)

- This lecture: **Binary Search Tree (BST)** -- A binary tree where at **ANY NODE**, its value is:
  - **greater than or equal to** the value of any node in its left subtree, and
  - **less than** the value of any node in its right subtree.
- You can think of any node as a 'pivot': its entire left subtree is smaller or equal to its value, and entire right subtree is larger. This property must be true for every node.



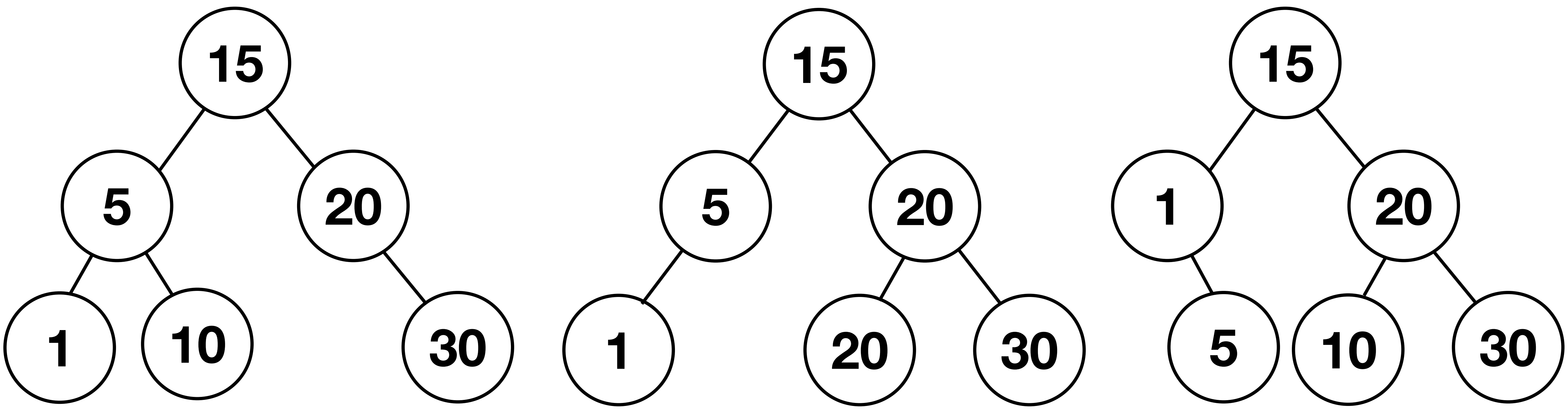
All values in the left subtree are less than or equal to the value in the root node.

All values in the right subtree are greater than the value in the root node.

The BST conditions have to be true for <u>**every node**</u> in the tree, not just the root!

Clicker Question #1

Are the following trees valid binary search trees?

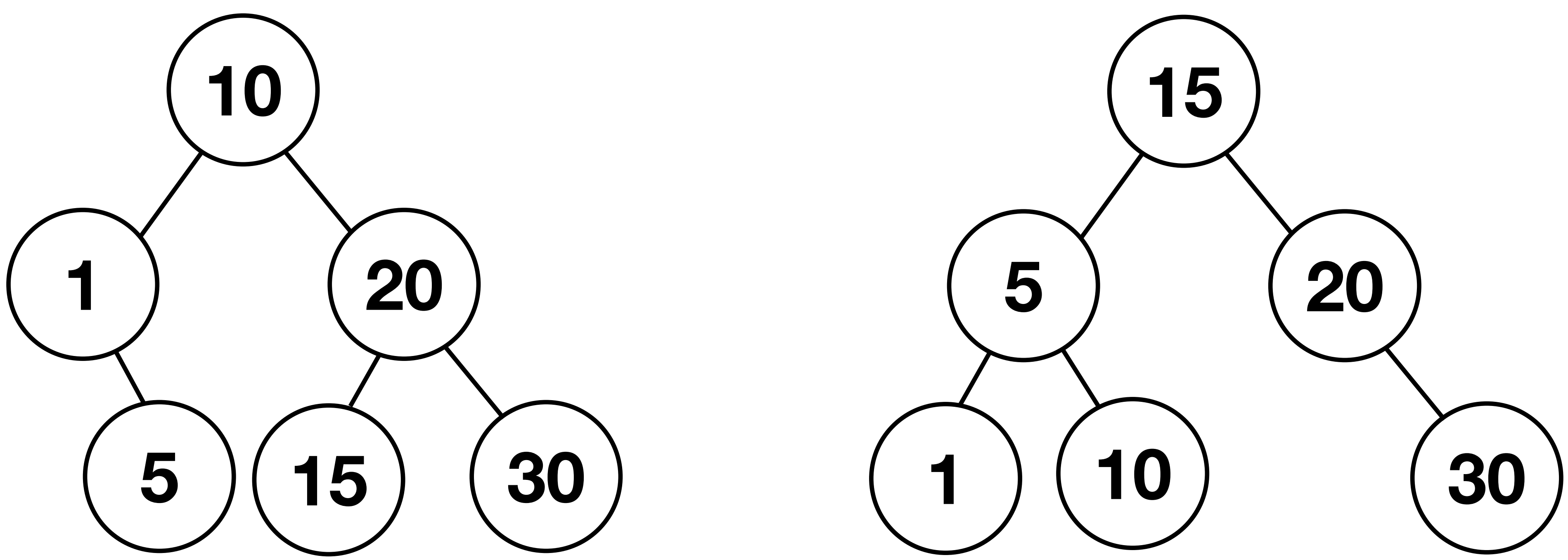


X Y Z

a) Only X is
b) Only Y is
c) Only Z is
d) Both X and Z are
e) Both X and Y are

In-Order Traversal of BST

- Let's work out the **in-order traversal** results of the following two valid BSTs.



For both, in order traversal gives the same result:

1, 5, 10, 15, 20, 30. This is clearly sorted!

In-Order Traversal of BST

- For a **BST**, **in-order** traversal visits every node in **ascending** (more precisely, non-decreasing) order.
  - This make sense because with in-order traversal, you visit (e.g. print out) the entire left-subtree first, then the current node, and then the entire right-subtree. Due to the properties of BST, this ends up visiting all nodes in ascending order.
- What about pre-order and post-order traversals of BST?
- What if I want to visit all nodes in **descending** order?

Hey! these are all different things

Please don’t confuse them

- **Binary Search**

  an algorithm on a sorted array.

- **Binary Tree**

  a tree where nodes have no more than 2 children.

- **Binary Search Tree**

  a binary tree with a special ordering property.

Search in a BST

However, Binary Search and BST are related, because the way you search in a BST is similar to performing a binary search in an ordered array.

Find 31

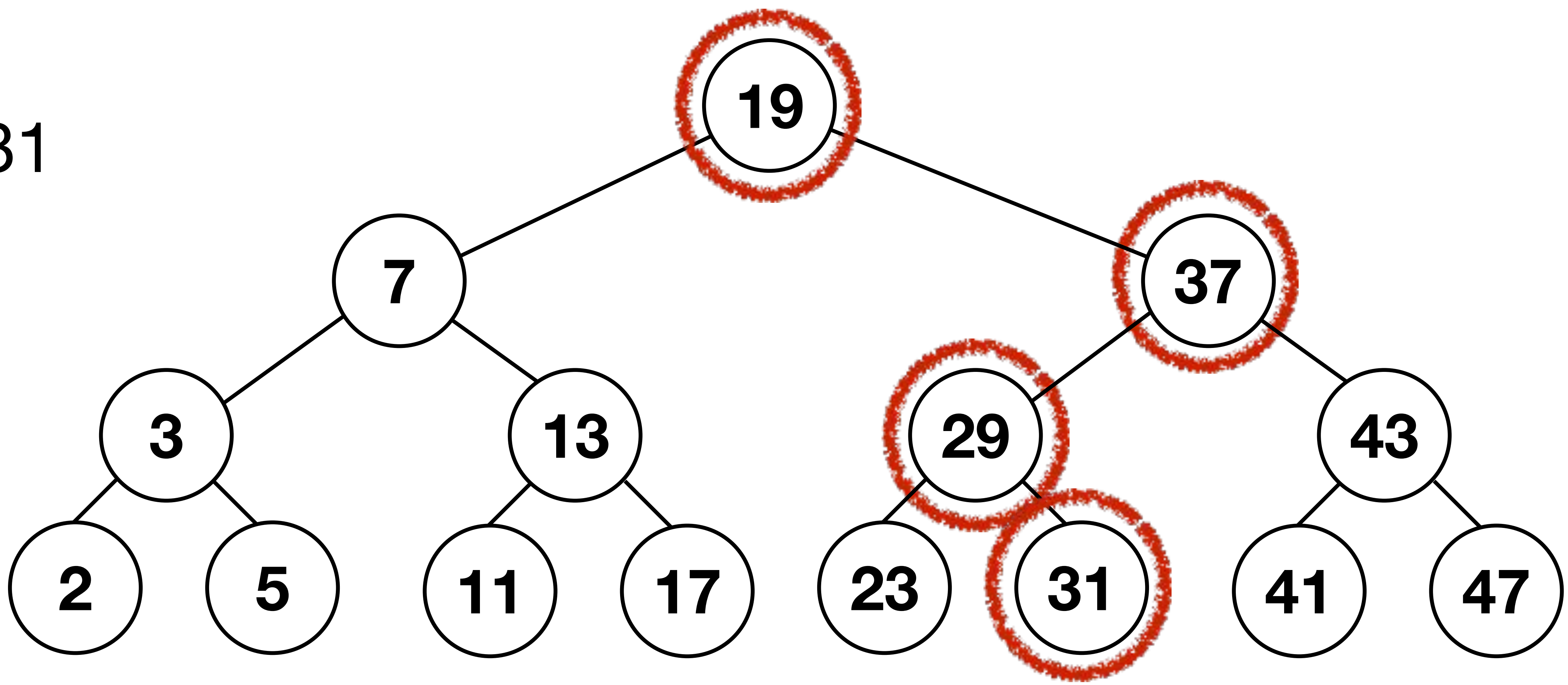
2	3	5	7	11	13	17	**19**	23	29	31	37	41	43	47
2	3	5	7	11	13	17	19	23	29	31	**37**	41	43	47
2	3	5	7	11	13	17	19	23	**29**	31	37	41	43	47
2	3	5	7	11	13	17	19	23	29	**31**	37	41	43	47

Search in a BST

Find 31

2	3	5	7	11	13	17	**19**	23	29	31	37	41	43	47
2	3	5	7	11	13	17	19	23	29	31	**37**	41	43	47
2	3	5	7	11	13	17	19	23	**29**	31	37	41	43	47
2	3	5	7	11	13	17	19	23	29	**31**	37	41	43	47

Find 31



Search in a BST

- To summarize, you start from the root node, then choose to go left or right depending on the comparison result. The search ends when either you've found the target or you've reached a leaf.
- The maximum number of steps is the **tree height**.
- As in binary search, search in BST can achieve `O(log N)` time. However, this requires the BST to be balanced (i.e. the height should be small).
- If you have a poorly constructed BST (e.g. degenerated to a linked list), you won't get the `O(log N)` performance!

BST Properties

- The **largest** element in a BST is in the **rightmost** node (i.e. starting from the root, follow the right child link all the way until you can't go further).
- Similarly, the **smallest** element in is in the **leftmost** node (i.e. starting from the root, follow the left child link all the way until you can't go further).
- **For a node that has two children**, its in-order **predecessor** is the largest element (i.e. rightmost node) in its left subtree. Its **successor** is the the smallest (i.e. leftmost node) in its right subtree.
- More accurately: predecessor is the element that comes just before the given node in in-order traversal; successor is the element that comes just after the given node in in-order traversal.

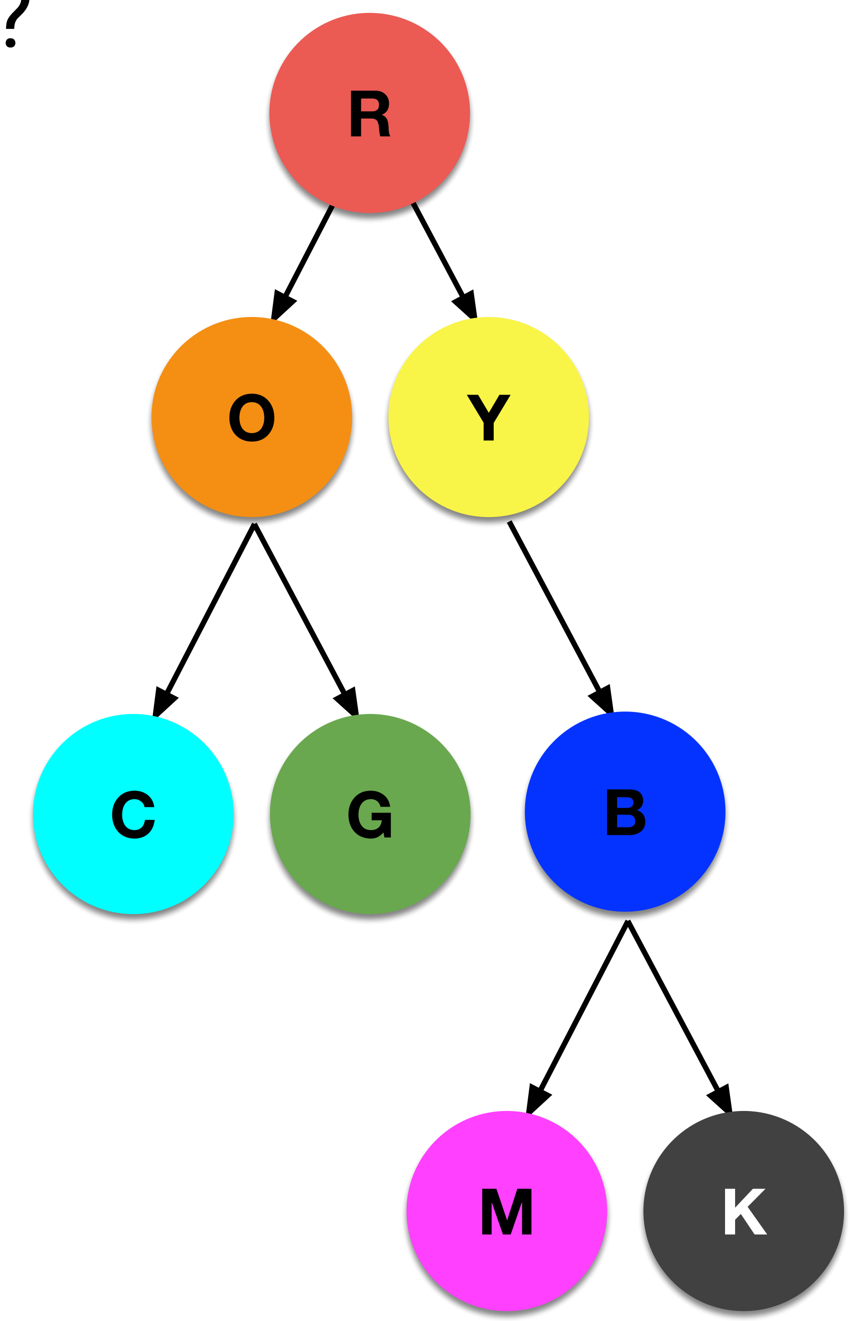
BST Properties

Q: Which node contain the **largest** element?
A: node K

Q: Which node contain **smallest** element?
A: node C

Q: Which node is root's **predecessor**?
A: node G

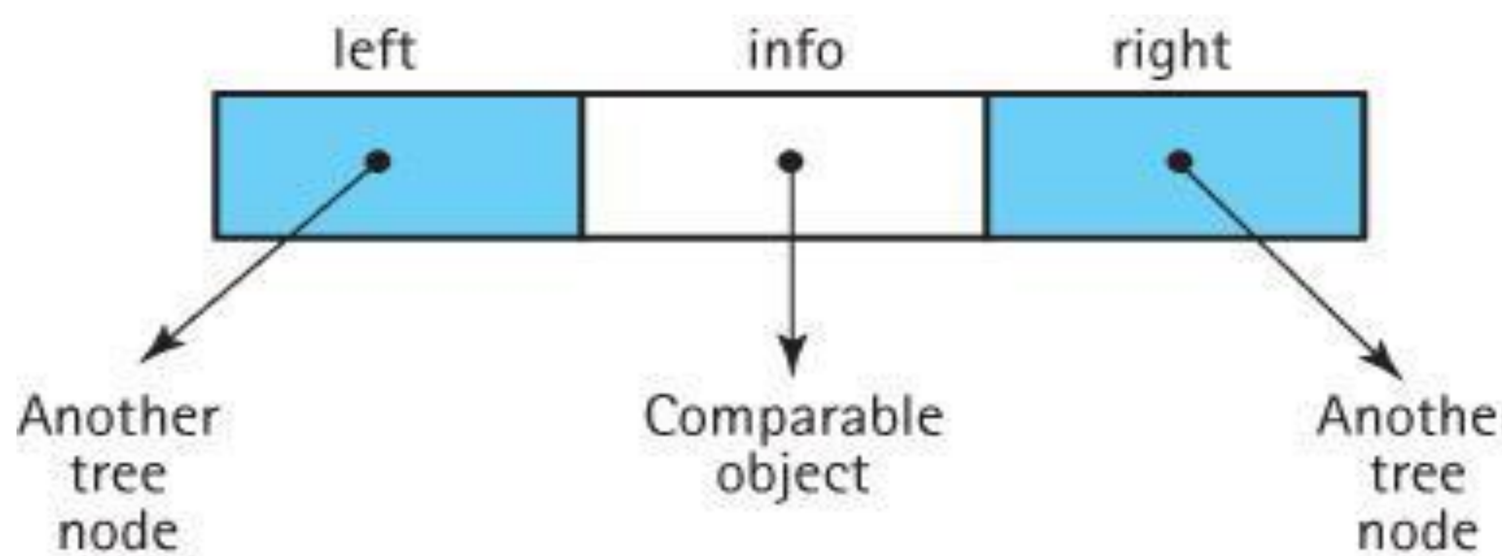
Q: Which node is root's **successor?**
A: node Y



BSTNode

```
public class BSTNode<T extends Comparable<T>>
{
  protected T info;  // info stored in a node
  protected BSTNode<T> left;  // link to the left child
  protected BSTNode<T> right; // link to the right child

  public BSTNode(T info) {
    this.info = info;
    left = null;
    right = null;
  }
 . . .
```



Plus the standard getters and setters for info, left, and right.

Basic Operations of BSTs

`add(elem)` : insert a new node to BST

`remove(elem)` : remove node containing elem

Must maintain BST ordering property

`contains(elem)` : return true if tree contains a node whose info equals elem.

`get(elem)`: find a tree node with info matching elem, return a reference to it; otherwise return null.

Exploit BST ordering property

`size()` : return count of nodes in BST.

Elegant recursive solution

(We will add some additional methods later.)

BinarySearchTree

```
public class BinarySearchTree<T extends Comparable<T>>
                                implements BSTInterface<T>
{
  protected BSTNode<T> root;  // pointer to the root node

  public void add(T element) {...}
  public boolean remove(T element) {...}
  public boolean contains(T element) {...}
  public T get(T element) {...}
  public int size() {...}
}
```

Creation and Maintenance of BST

- All modifying operations **must maintain the ordering constraint** of the BST.

  - **`add(elem)`** : insert new node to the BST

  - **`remove(elem)`** : remove node containing elem

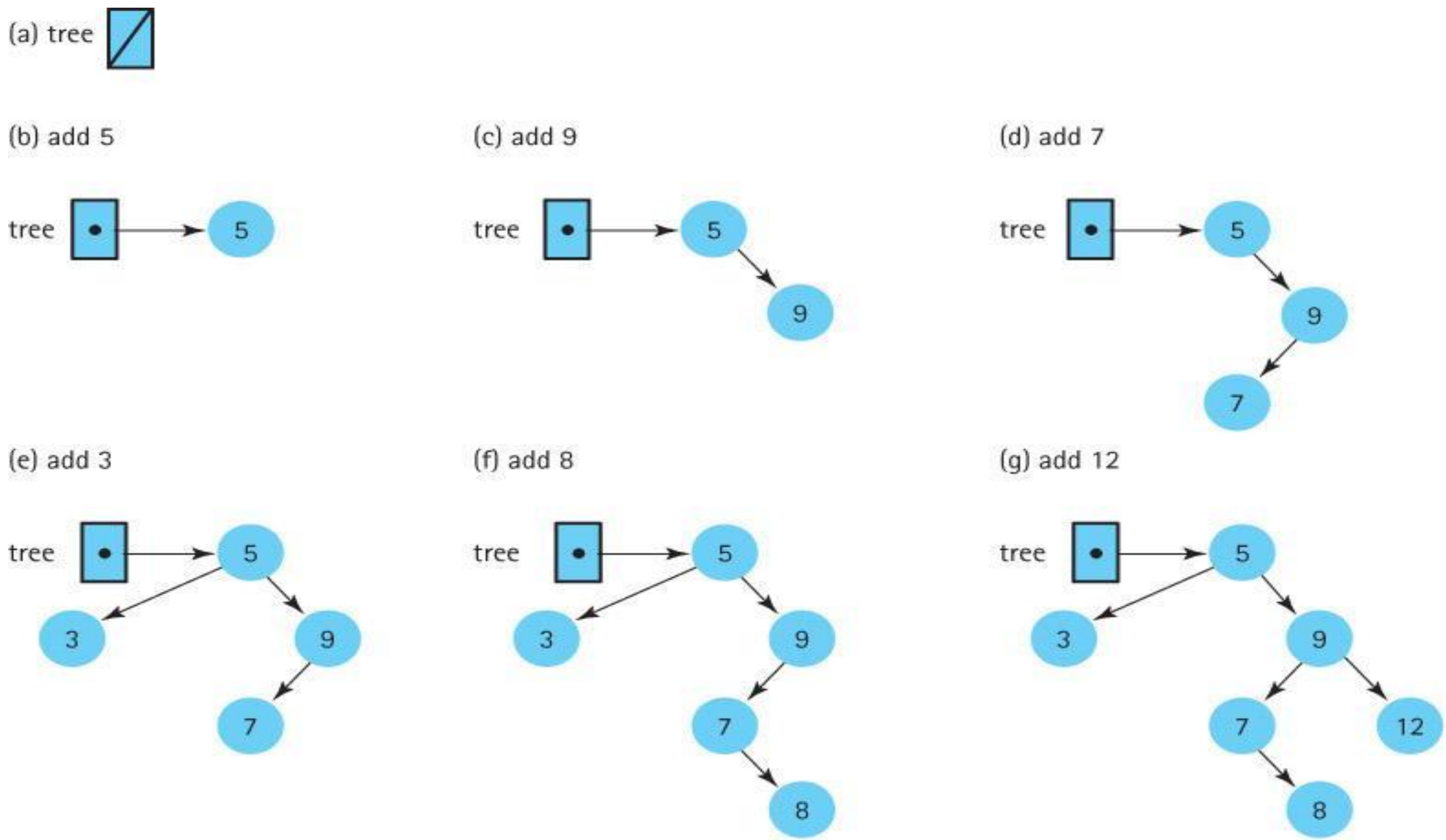
Inserting to an Empty BST

5, 9, 7, 3, 8, 12

(a) tree 

Inserting to an Empty BST

5, 9, 7, 3, 8, 12

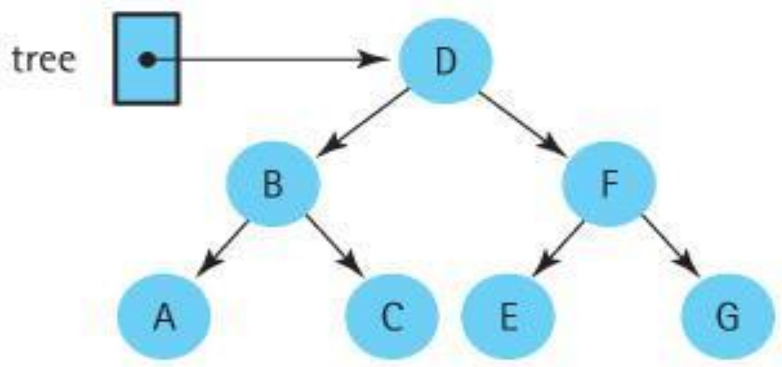


Summary of Insertion

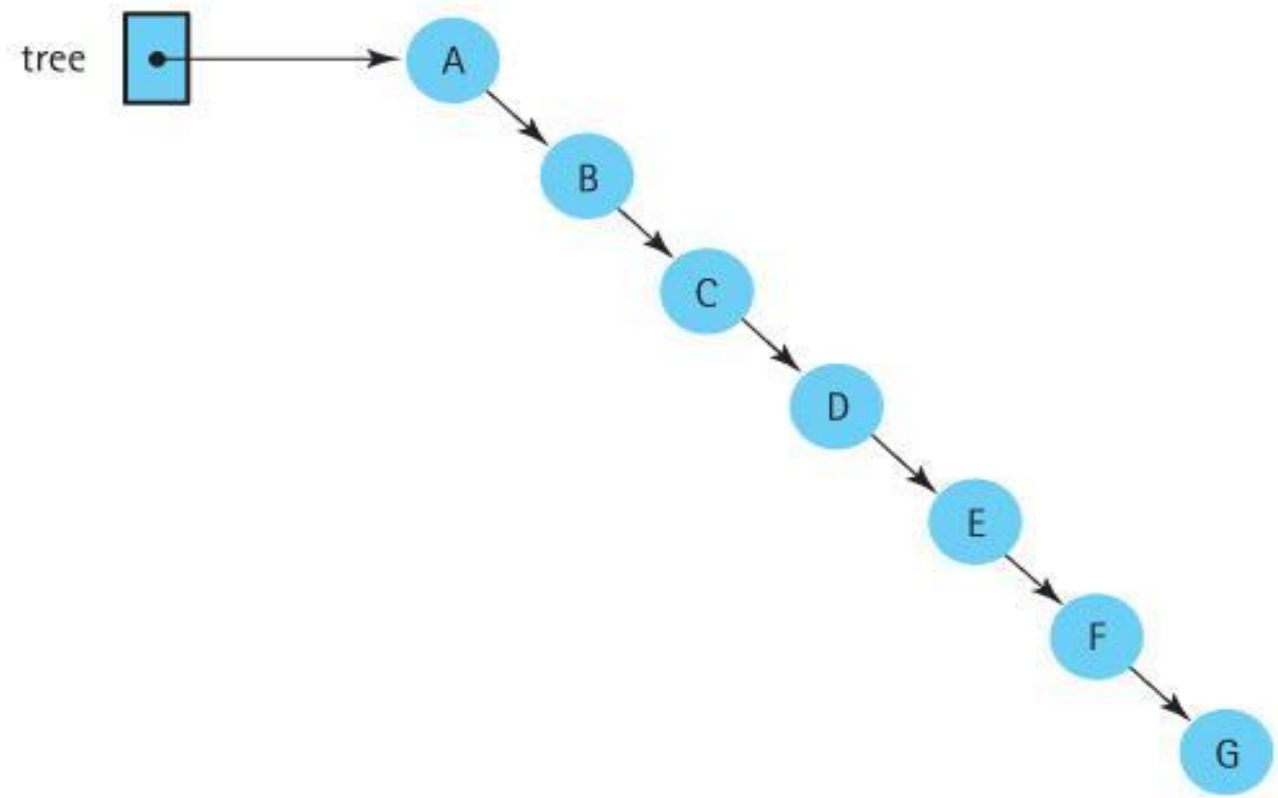
- First, find the node to insert the new element to. This is much the same process as trying to find an element that turns out not to exist.
- Once you've found the node, insert the new element as either its left child or right child, depending on the comparison result.
- Note that **the new element is always inserted into BST as a leaf node**!

A Key Point

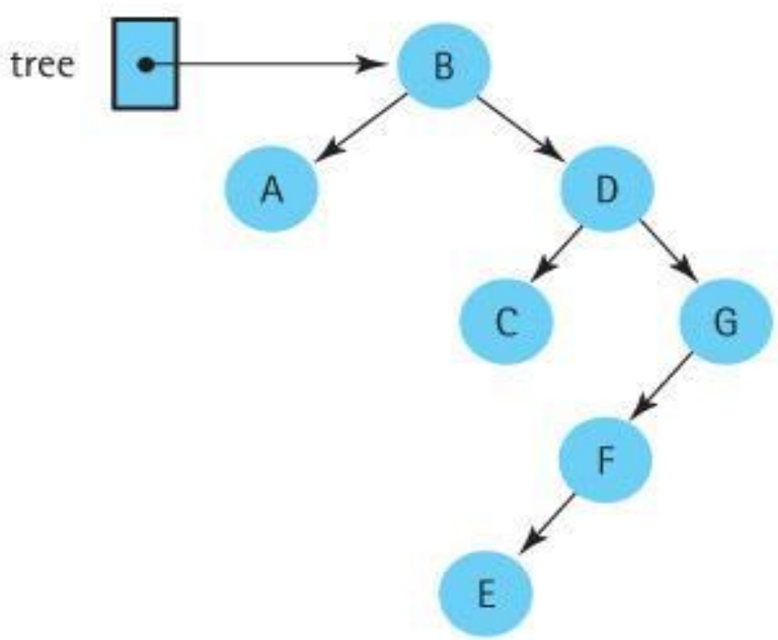
Insertion order will determine the shape of BST (some lead to more balanced tree, some do not)



D B F A C E G



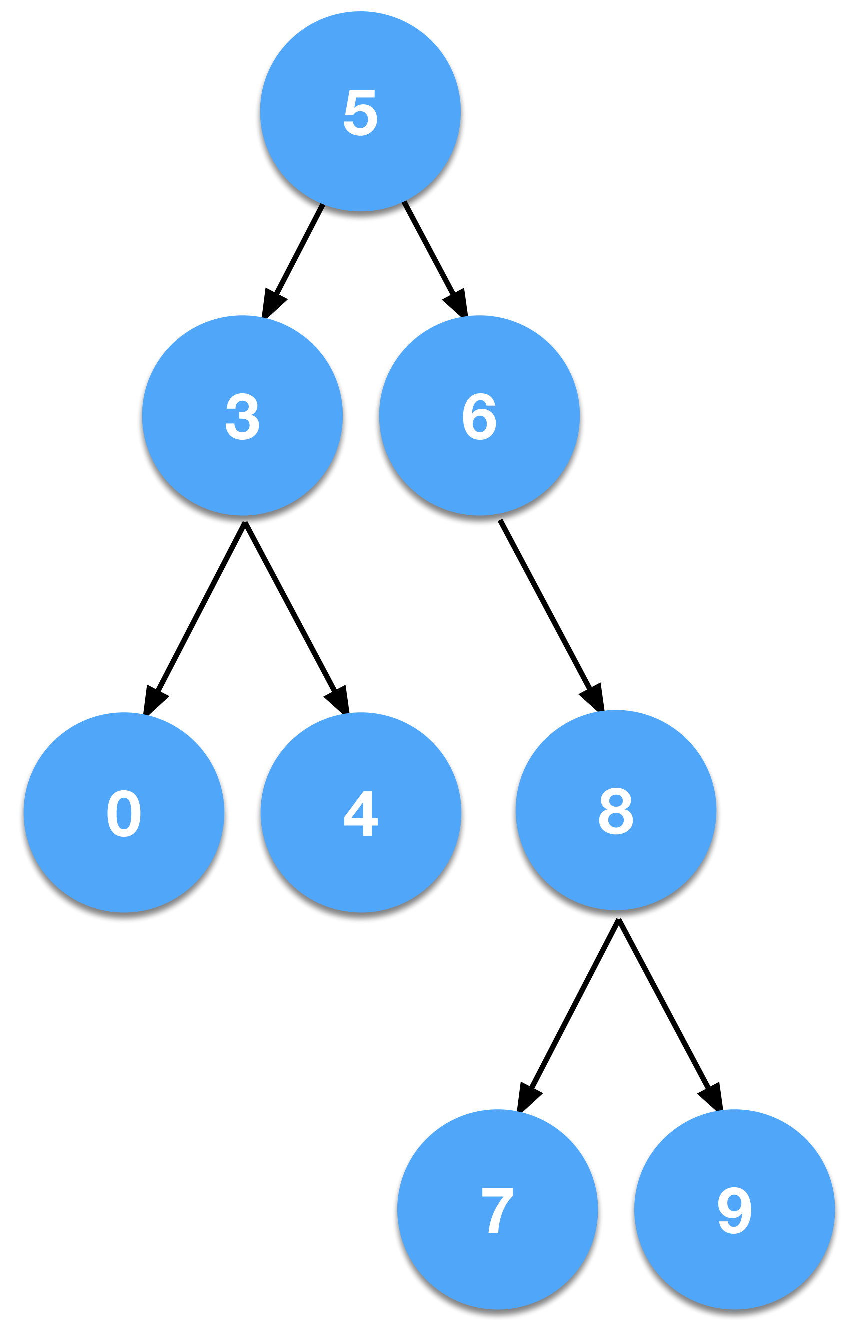
A B C D E F G



B A D C G F E

Clicker Question #2

Which of the following insertion orders could have led to this tree?



A) 5 6 8 4 3 0 7 9

B) 5 3 4 6 0 8 9 7

C) 5 3 6 0 4 7 8 9

D) 5 6 3 0 7 4 8 9

E) None of the above

The Remove Operation

`remove(elem)` : remove node containing elem

- The most complicated in BST operation.
- We must ensure the BST property is preserved when we remove an element.
- No need to memorize the code, but you must understand it conceptually, such that given a BST and a node to remove, you can draw the resulting BST after removal.

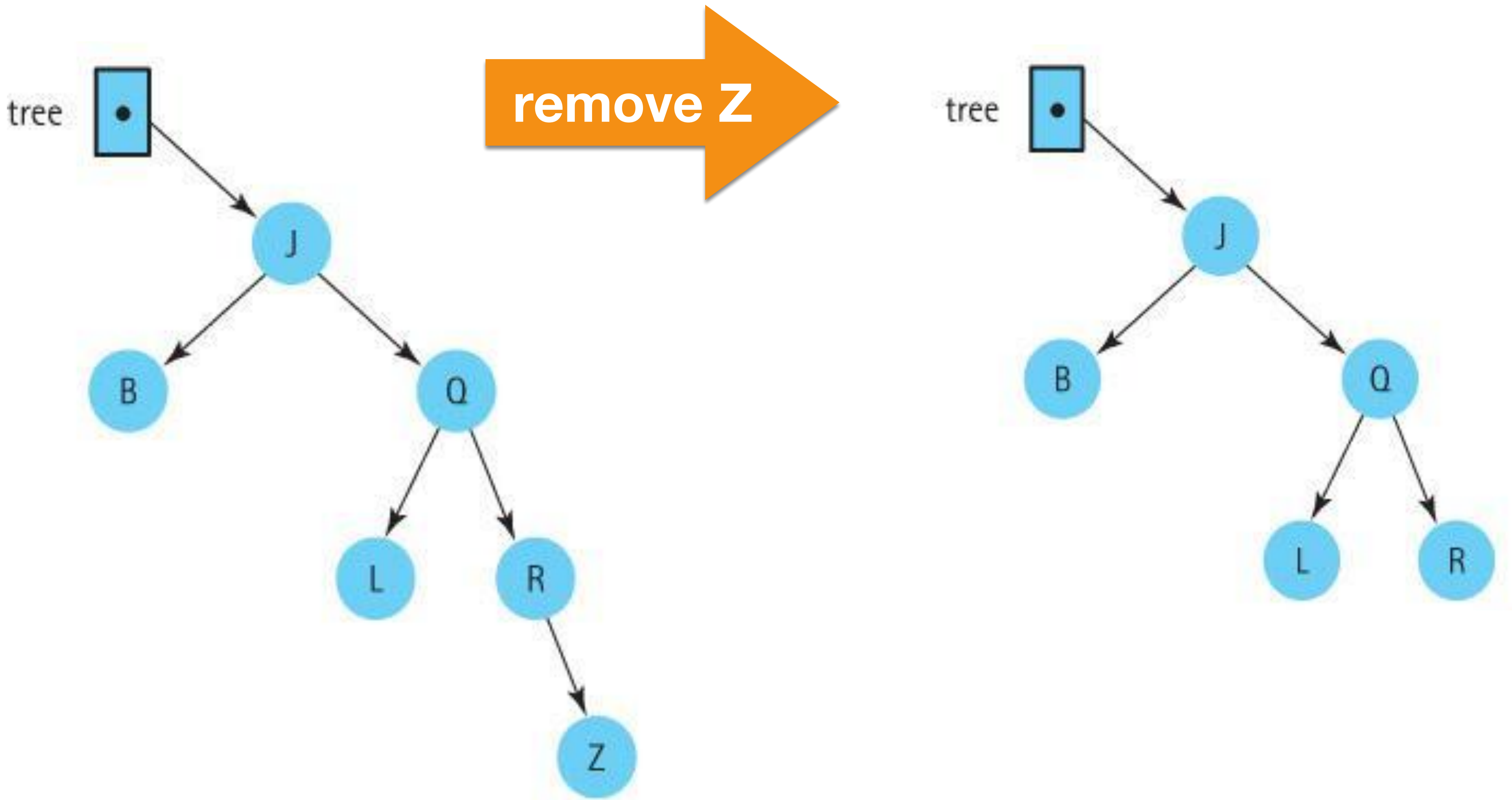
Three Cases for `remove`

Easy **Removing a leaf (no children)**: removing a leaf is simply a matter of setting the appropriate link of its parent to null.

OK **Removing a node with only one child**: make the reference from the parent skip over the removed node and point instead to the child of the node we intend to remove.

Tricky **Removing a node with two children**: replaces the node’s info with the info from another node in the tree so that the search property is retained - then remove this other node.

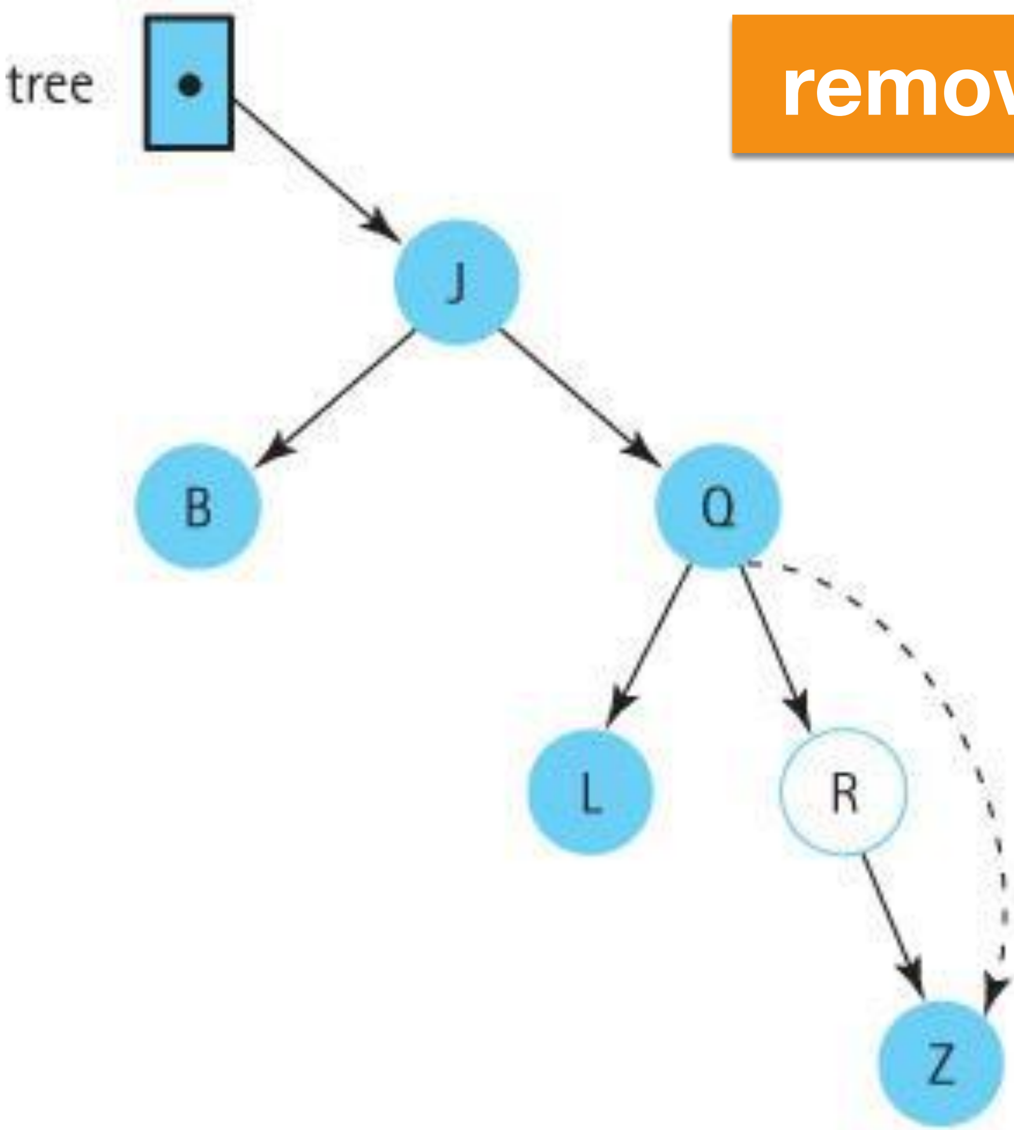
Case 1: Removing a Leaf Node

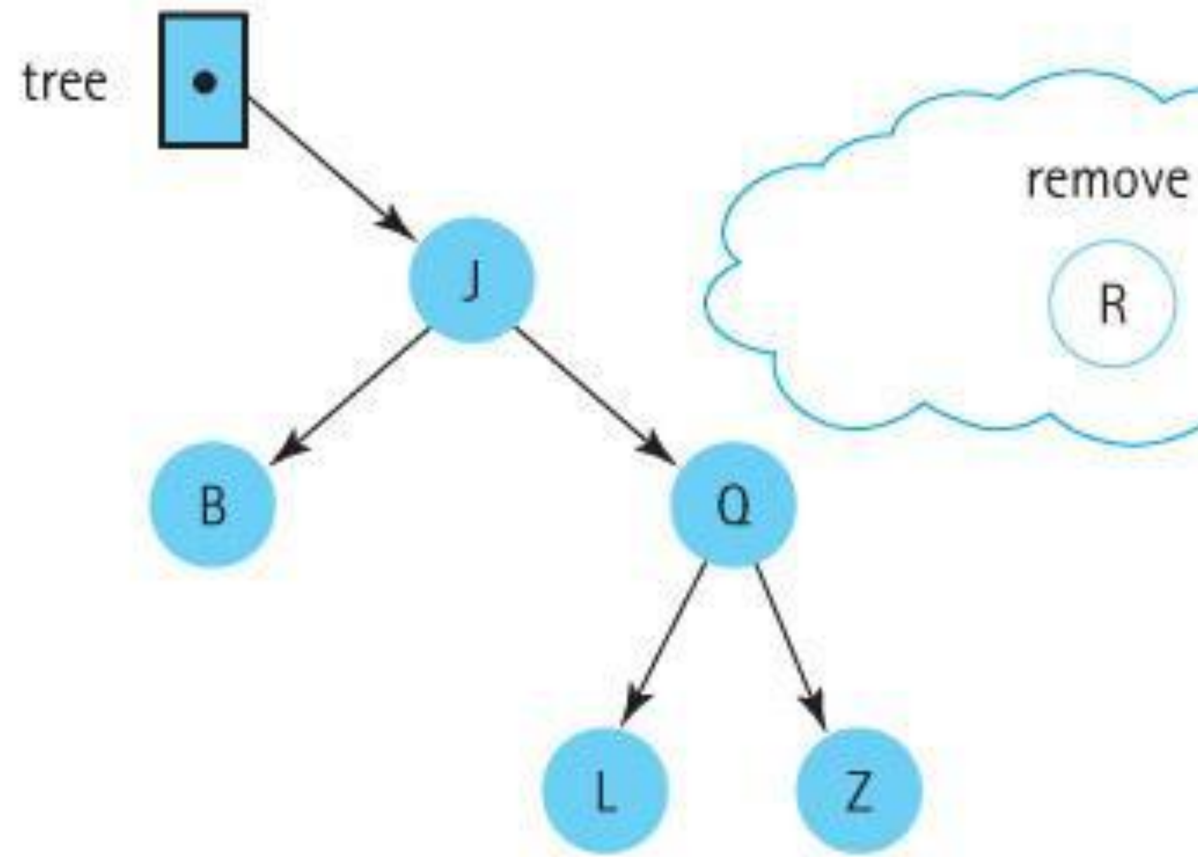


Before **After**

Case 2: Remove a Node with One Child

remove R



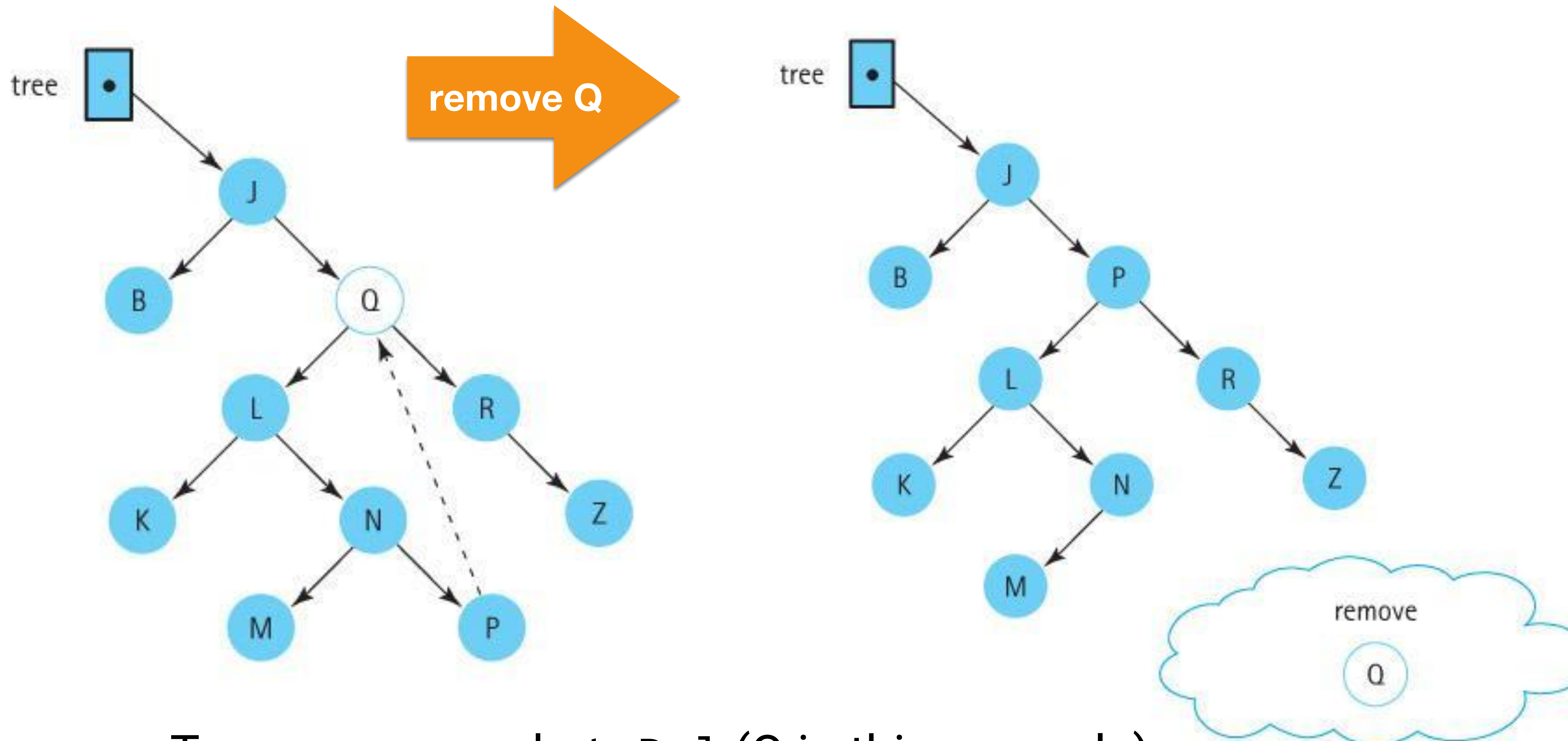


What if R has a left child
Instead of a right child?

Before

After

Case 3: Remove a Node with Two Children



To remove a node `toDel` (Q in this example):

1. Find the node's (in-order) **predecessor** `pre`
2. **Replace** `toDel.info` with `pre.info`
3. **Remove** `pre` (this sounds like a recursion)

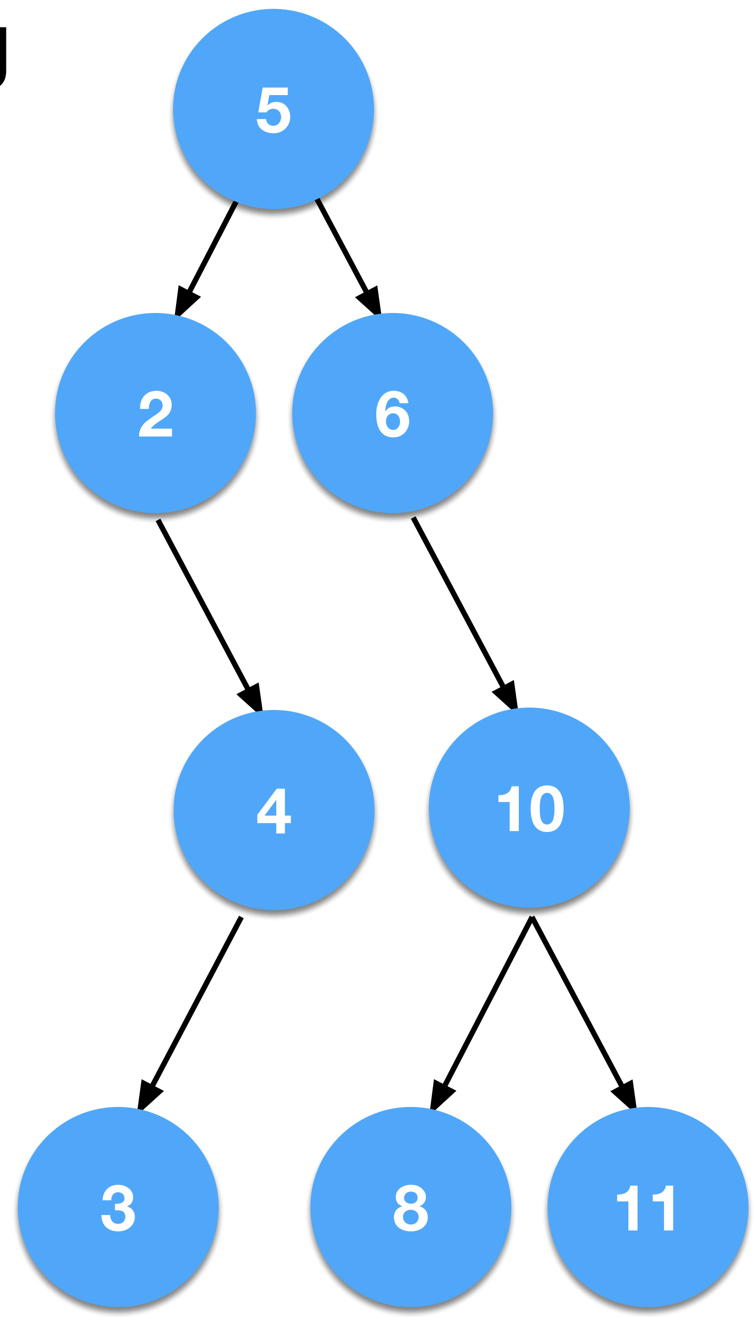
Case 3: Remove a Node with Two Children

Why does this work? Because replacing a node with its predecessor preserves the BST's ordering property. Why?

What happens when we remove the root (5) in the tree on the right? What's its predecessor?

Is the predecessor a leaf? If not, how do you remove it?

Is it possible that the predecessor has two children again, so removing it becomes another case 3 (difficult case)?



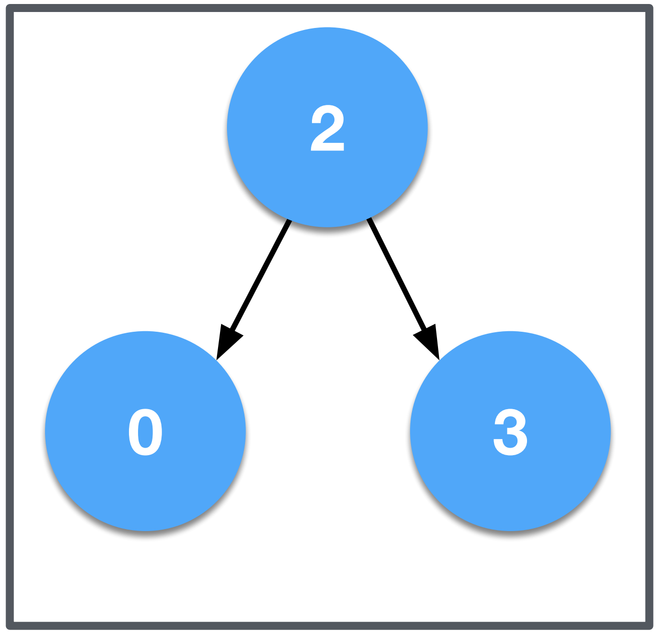
Removing a Node with Two Children

- How do we know the predecessor won't have two children itself?
    - Because it cannot have a right child (Why?), it is either a leaf node or it's a node with only one left child.
    - Hence we know removing the predecessor is one of the easy cases.
- Instead of the predecessor, is there another node we can use to replace the node to be removed?

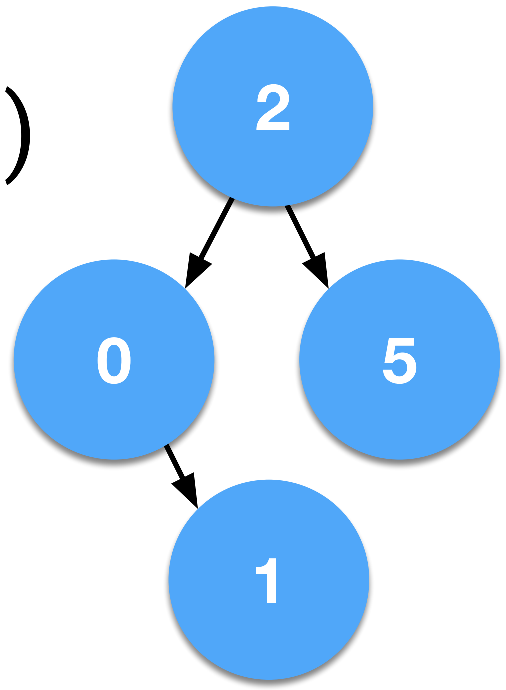
Yes! The (in-order) **successor**! The process is similar.

Clicker Question #3

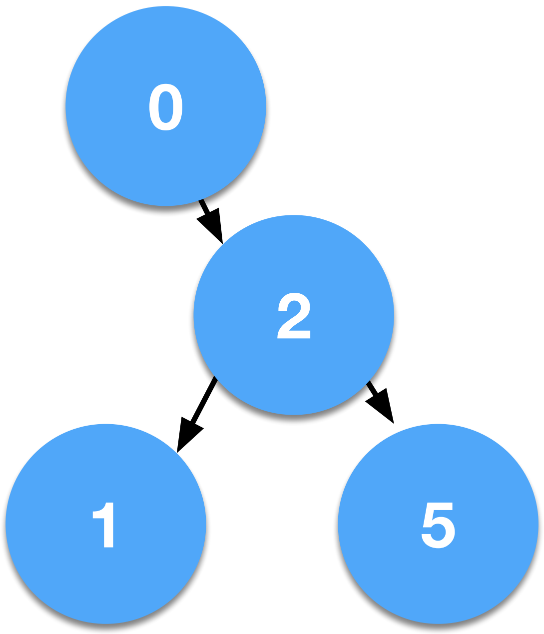
- What does this BST look like after the following operations (when removing a node, we replace it with its predecessor)? remove 2 -> add 2 -> add 5 -> add 1 -> remove 3



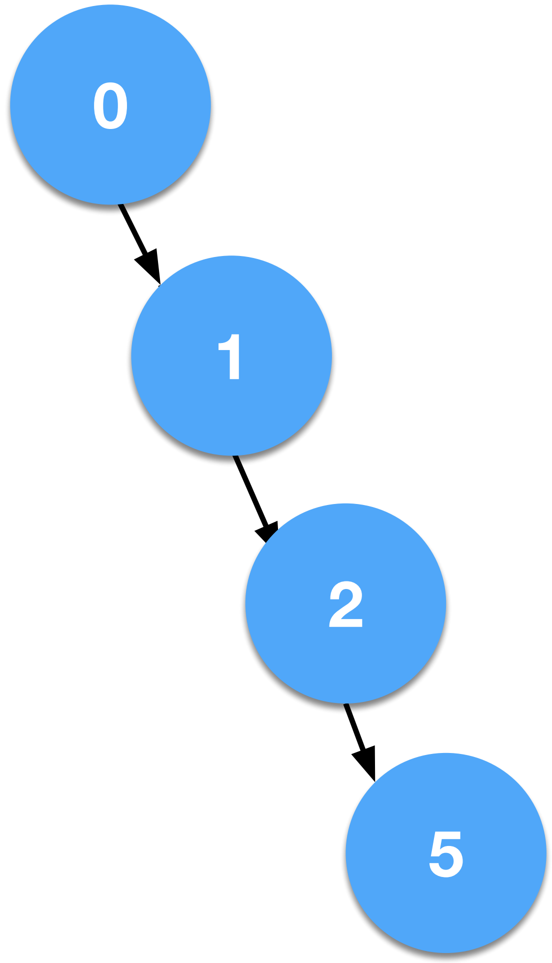
a)



b)



c)



The **`size()`** method

Think for a moment how you would implement the **`size()`** method. Can you do this recursively?

```
public int size() {
  return recSize(root);
}
private int recSize(BSTNode<T> node) {
  if (node==null) return 0;
  else
    return 1 + recSize(node.getLeft())
             + recSize(node.getRight());
}
```

The **get** method

```
public T get(T element) {
  return recGet(element, root);
}

private T recGet(T element, BSTNode<T> node) {
  // Returns element e such that e.compareTo(element) == 0;
  // if no such element exists, returns null.
  if (node == null)  return null;  // element is not found
  else if (element.compareTo(node.getInfo()) < 0)
    // get from left subtree
    return recGet(element, node.getLeft());
  else if (element.compareTo(node.getInfo()) > 0)
    // get from right subtree
    return recGet(element, node.getRight());
  else
    return node.getInfo();  // element is found
}
```

Activity

https://kahoot.it

Game id: 9078269