Programming with Data Structures

CMPSCI 187
Spring 2016

- **Please find a seat**
  - **Try to sit close to the center (the room will be pretty full!)**
- **Turn off or silence your mobile phone**
- **Turn off your other internet-enabled devices**

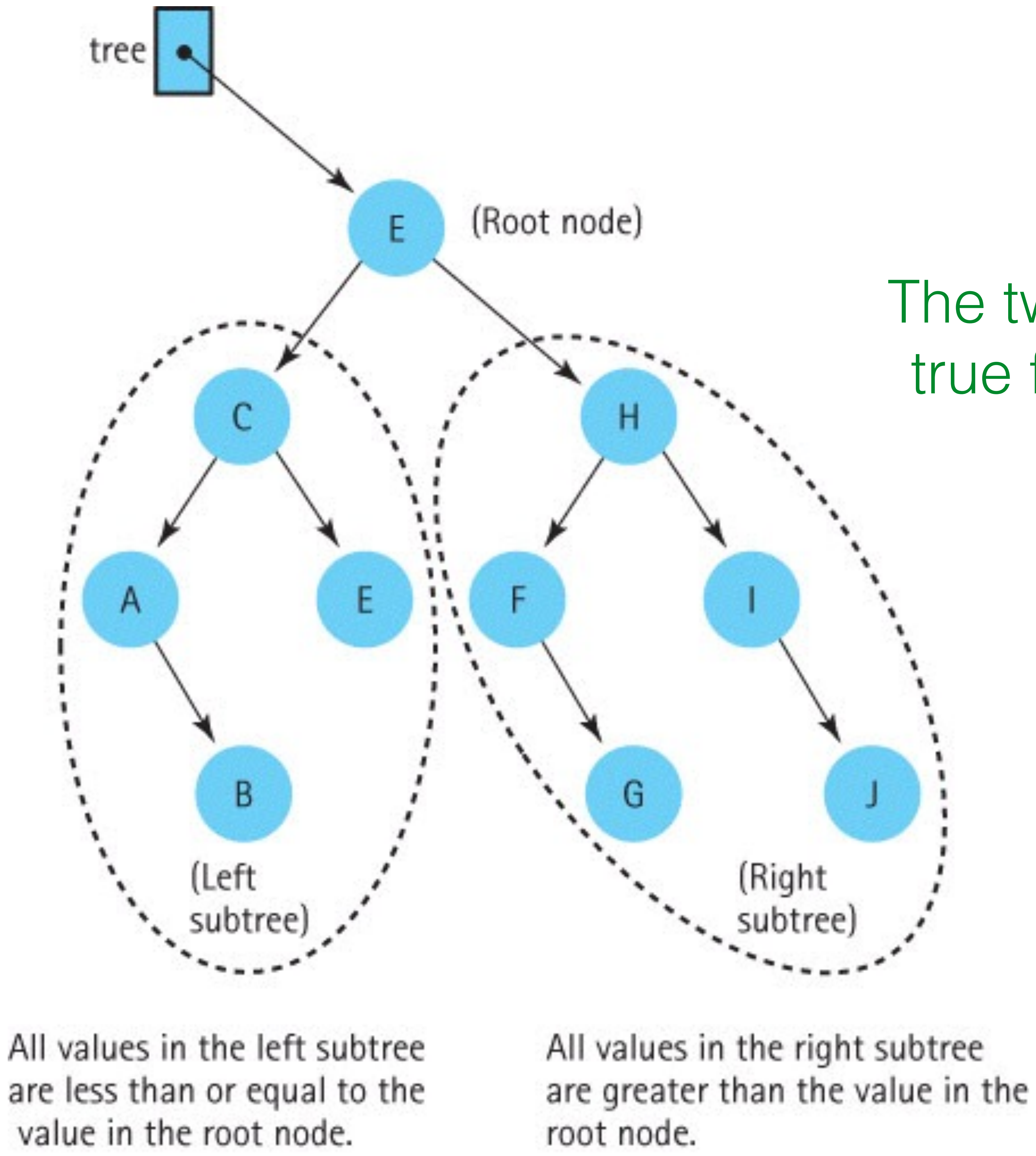
Reminders and Topics

- **Project 7 is due this Friday**
  - **reuse methods**
  - **write your own tests!**
- **The second midterm is Wed, March 30, 7-9pm**

- This lecture:
  - **Binary Search Tree**

Binary Search Tree (BST)

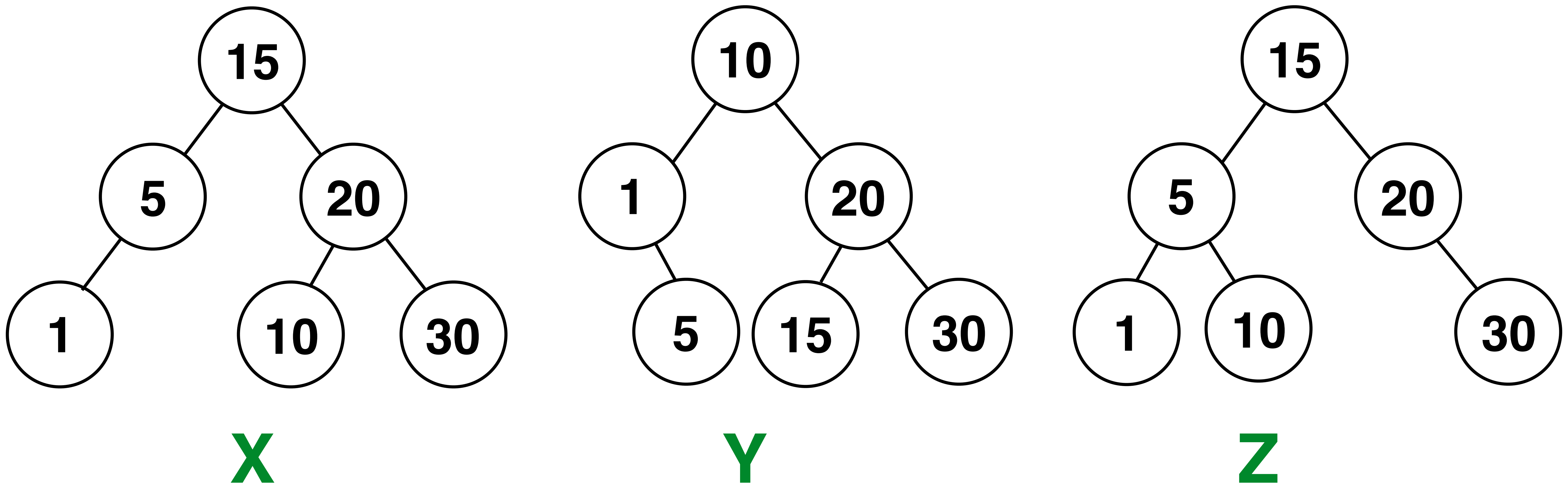
- **Binary search tree** A binary tree where at **ANY NODE**, its value is:
  - **greater than or equal to** the value of any node in its left subtree, and
  - **less than** the value of any node in its right subtree.


tree
E
(Root node)
C
H
A
E
F
I
B
G
J
(Left subtree)
(Right subtree)
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 two criteria have to be true for **any given node**!

Clicker Question #1

Are the following trees valid binary search trees?



a) Only X is

b) Only Y is

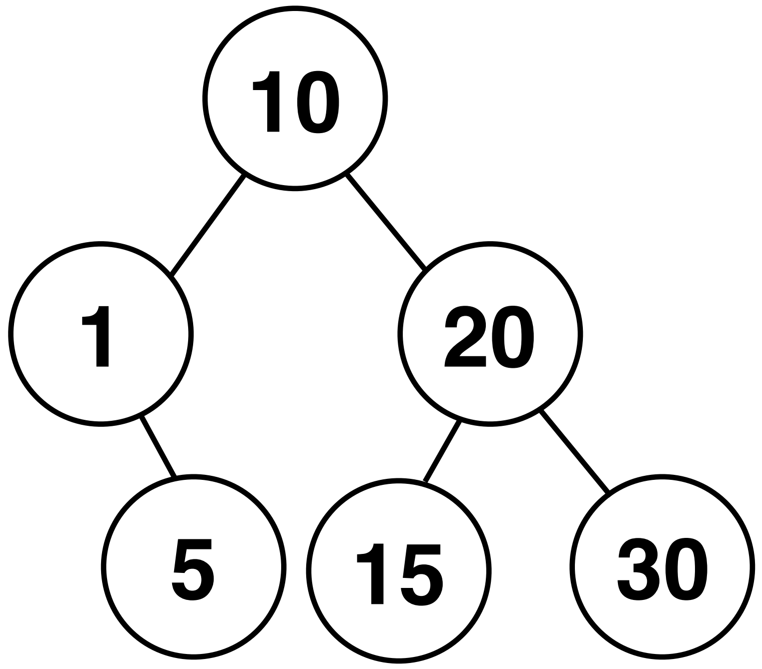
c) Only Z is

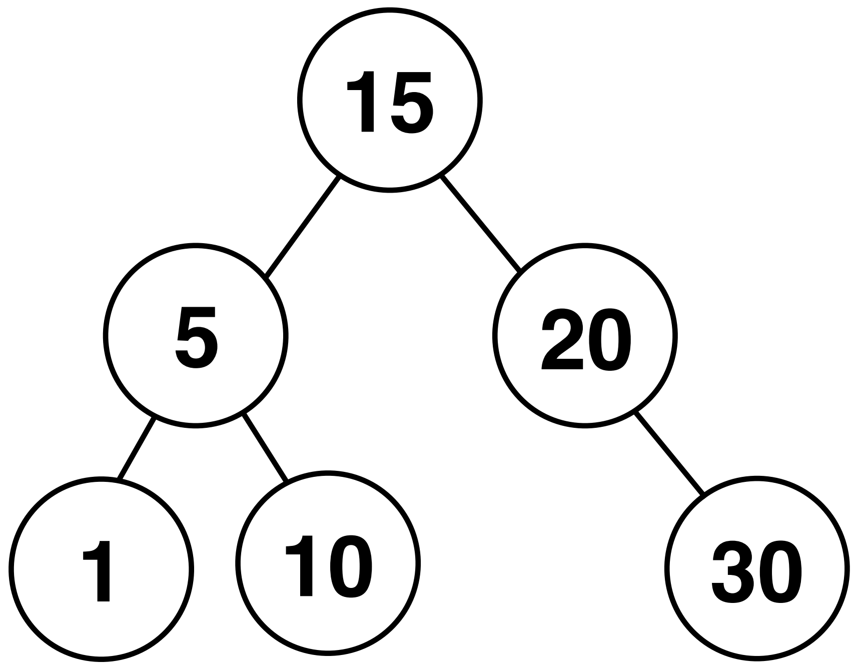
d) Both X and Y are

e) Both Y and Z are

In-Order Traversal of BST

- Let’s work out the **in-order traversal** results of the following two 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

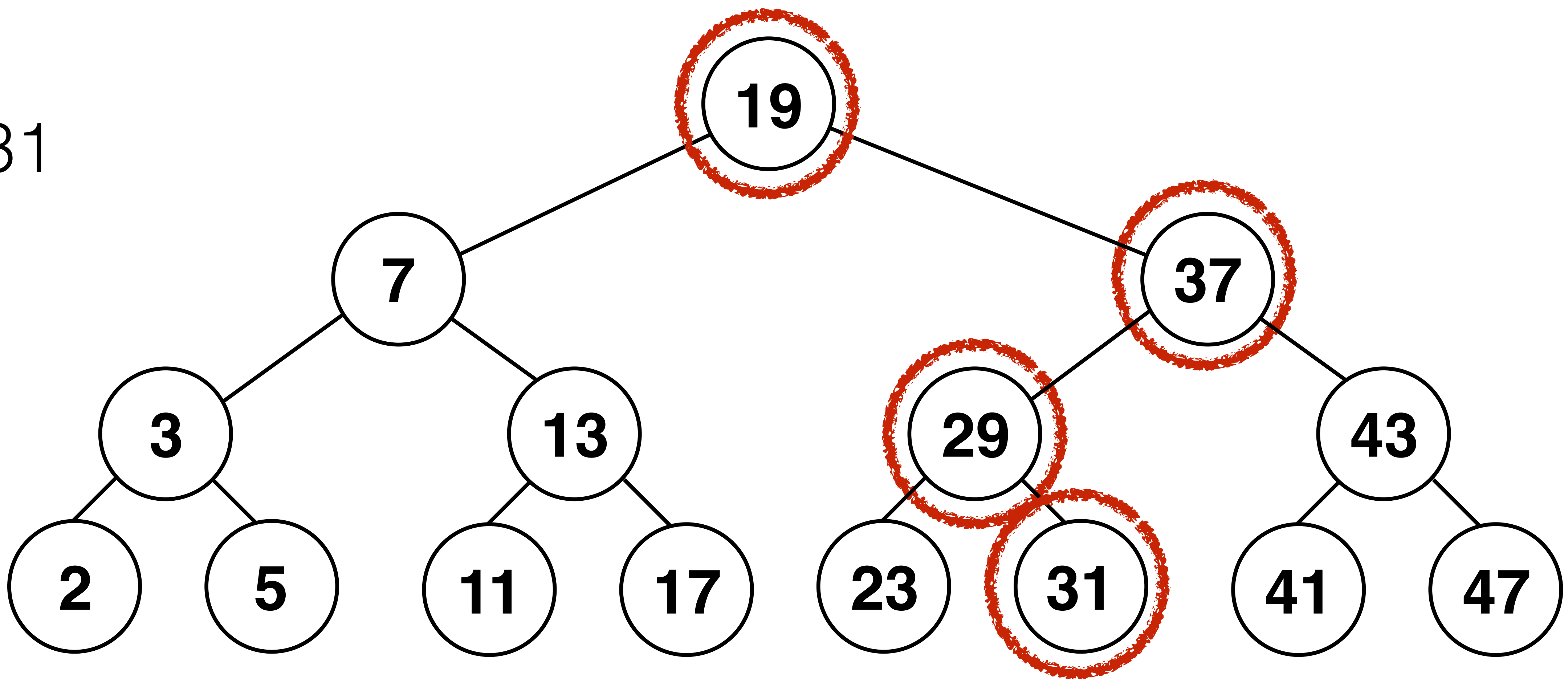
2	3	5	7	11	13	17	**19**	23	29	31	37	41	43	47
2	3	5	7	11	13	17	19	23	29	31	**37**	41	43	47
2	3	5	7	11	13	17	19	23	**29**	31	37	41	43	47
2	3	5	7	11	13	17	19	23	29	**31**	37	41	43	47

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

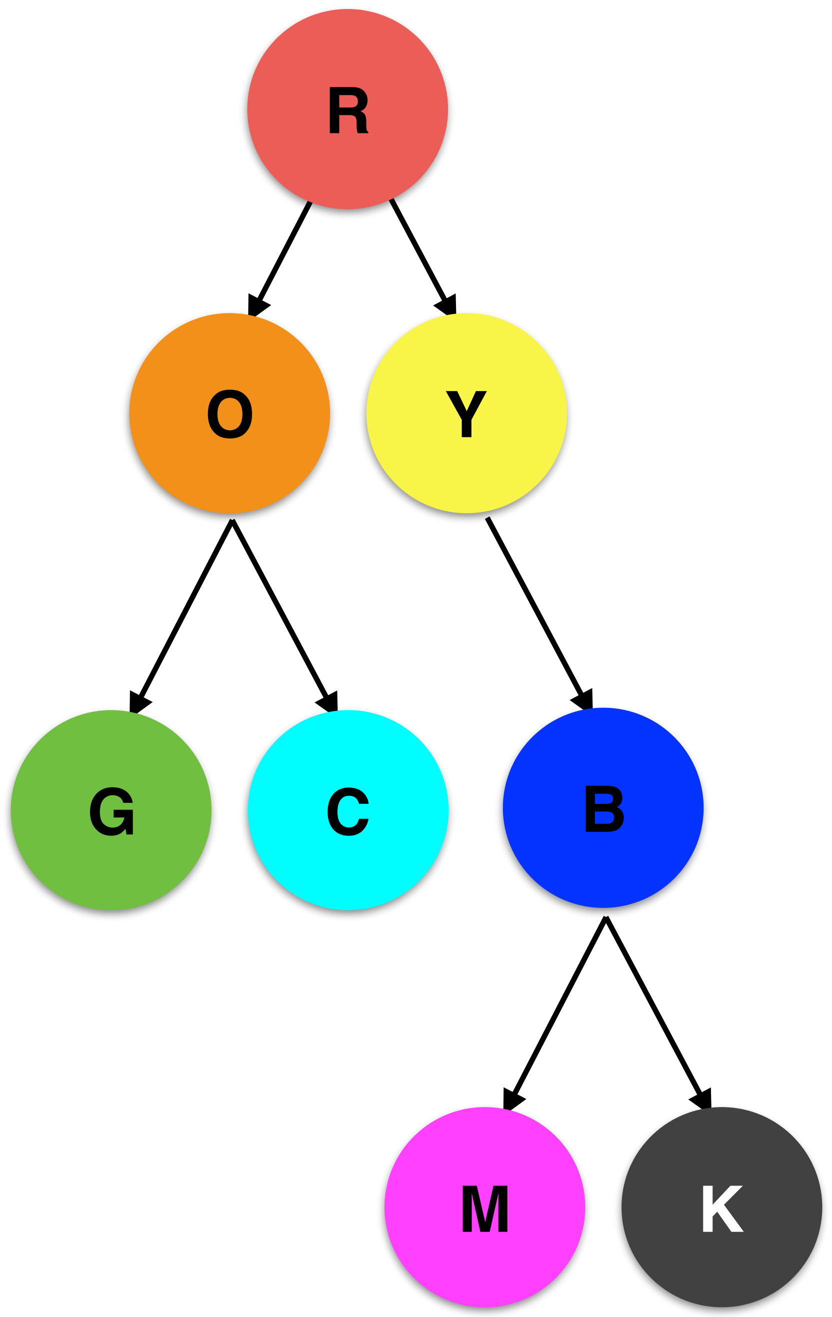


Search in a BST

- 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 Structure

- Here is a BST in which the node info has been hidden. Based on the structure of the BST alone…
    - Which node contains the **largest** element?
    - Which node contains the **smallest** element?



Clicker Question #2

- Here is a BST. Which node contains the element that comes **just before the root** (in sorted order), and which node contains the element that comes **just after the root** (in sorted order)?

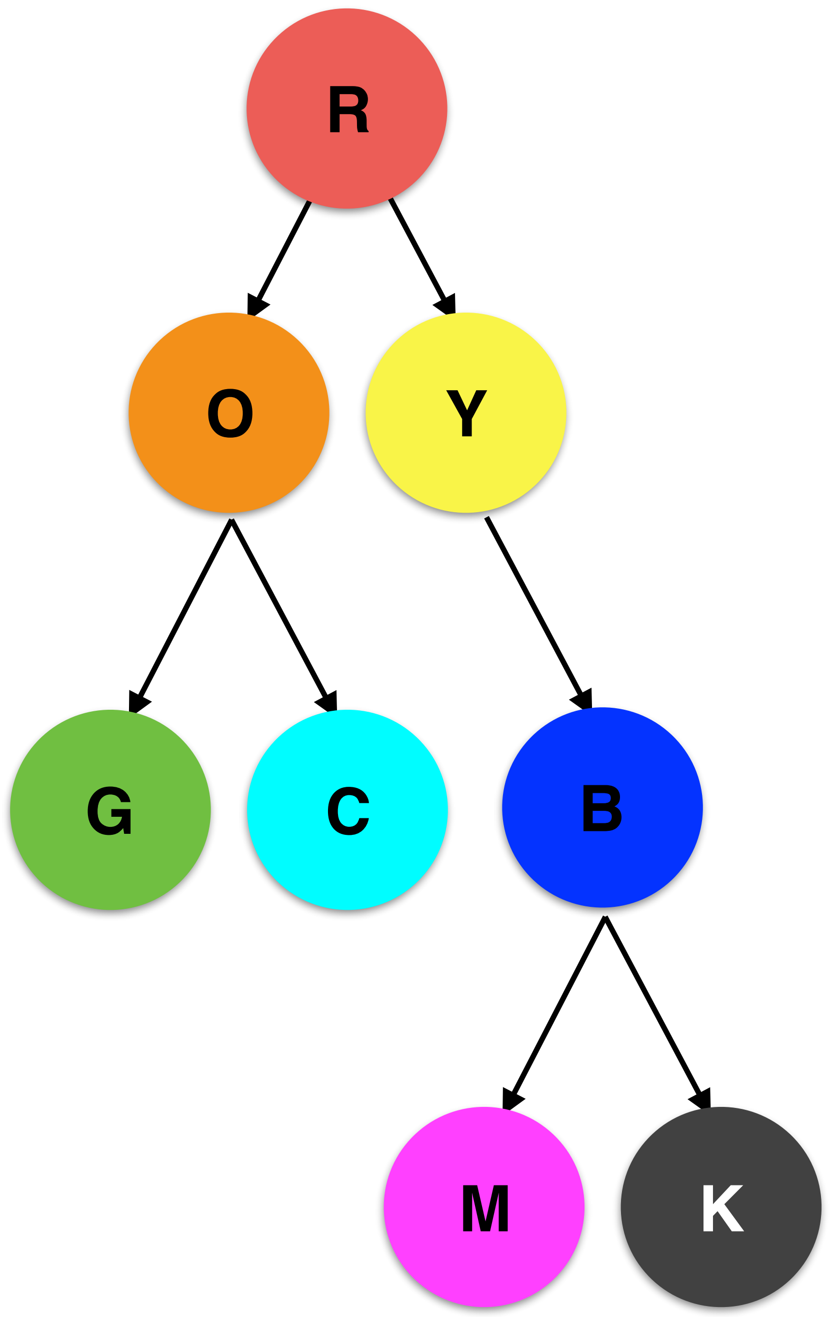
    a) green, cyan

    b) orange, yellow

    c) cyan, yellow

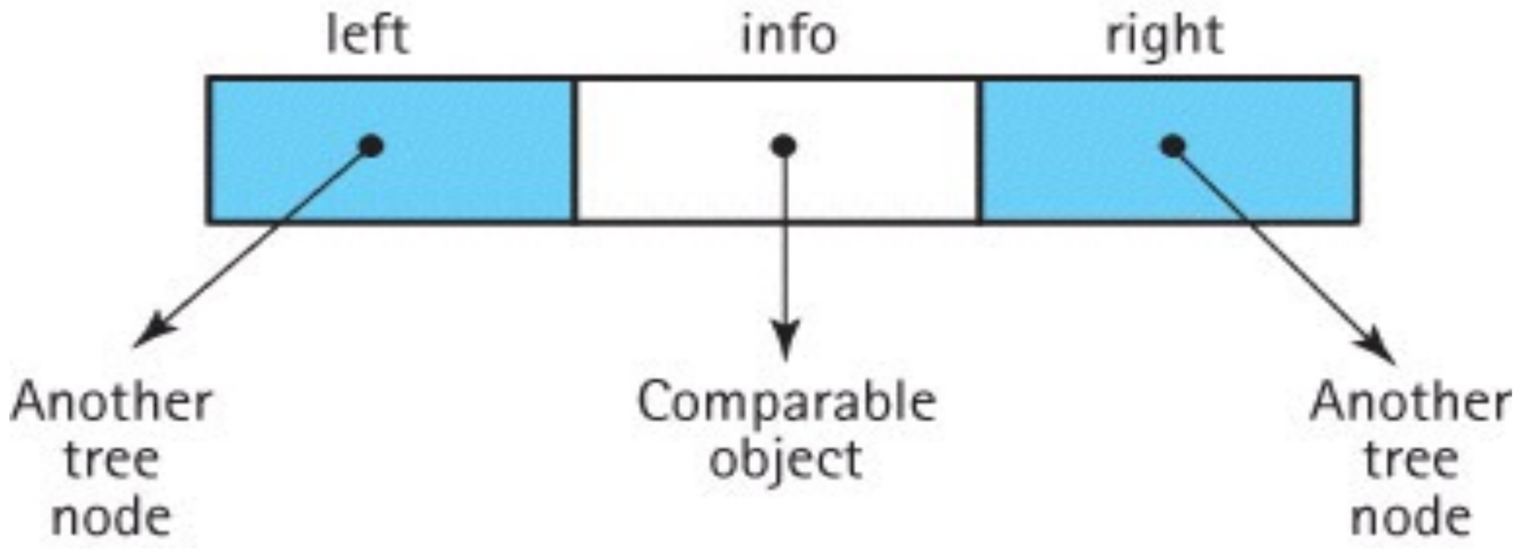
    d) orange, black

    e) cyan, magenta



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 node containing 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

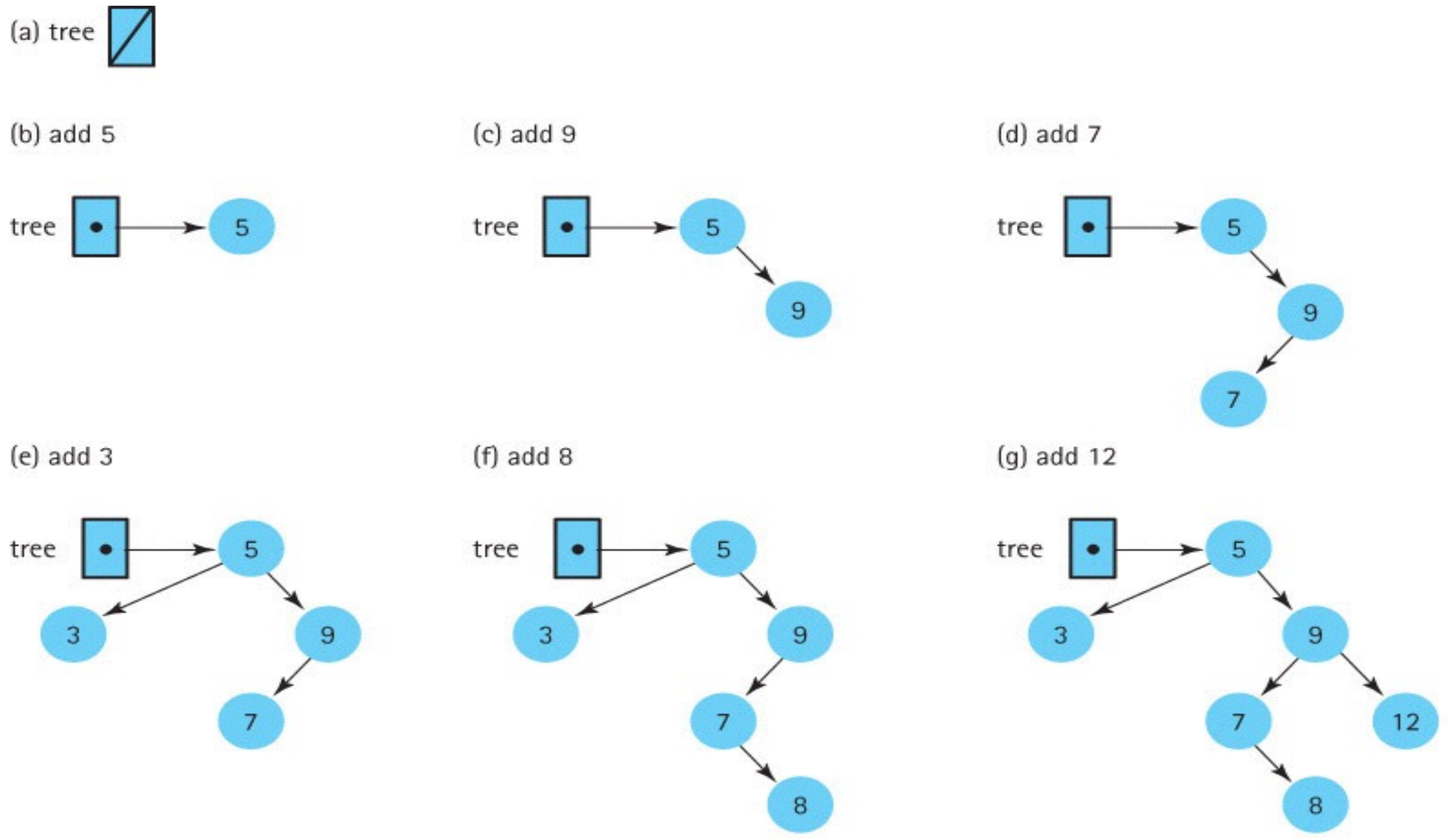
```
public class BinarySearchTree<T extends Comparable<T>>
                                implements BSTInterface<T>
{
  protected BSTNode<T> root;  // link to the root node

  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 BSTs

- 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

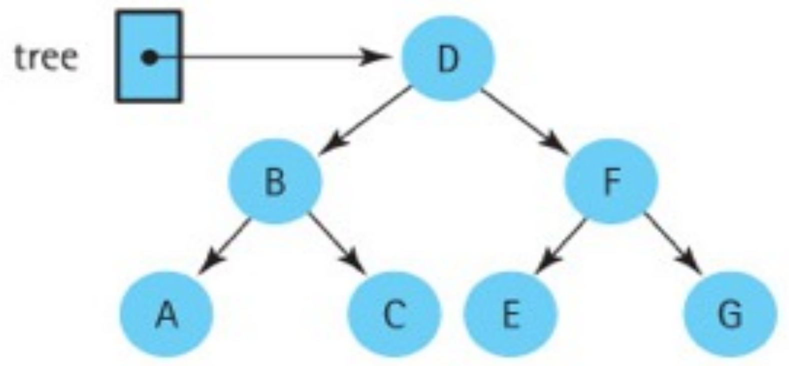


Summary of Insertion

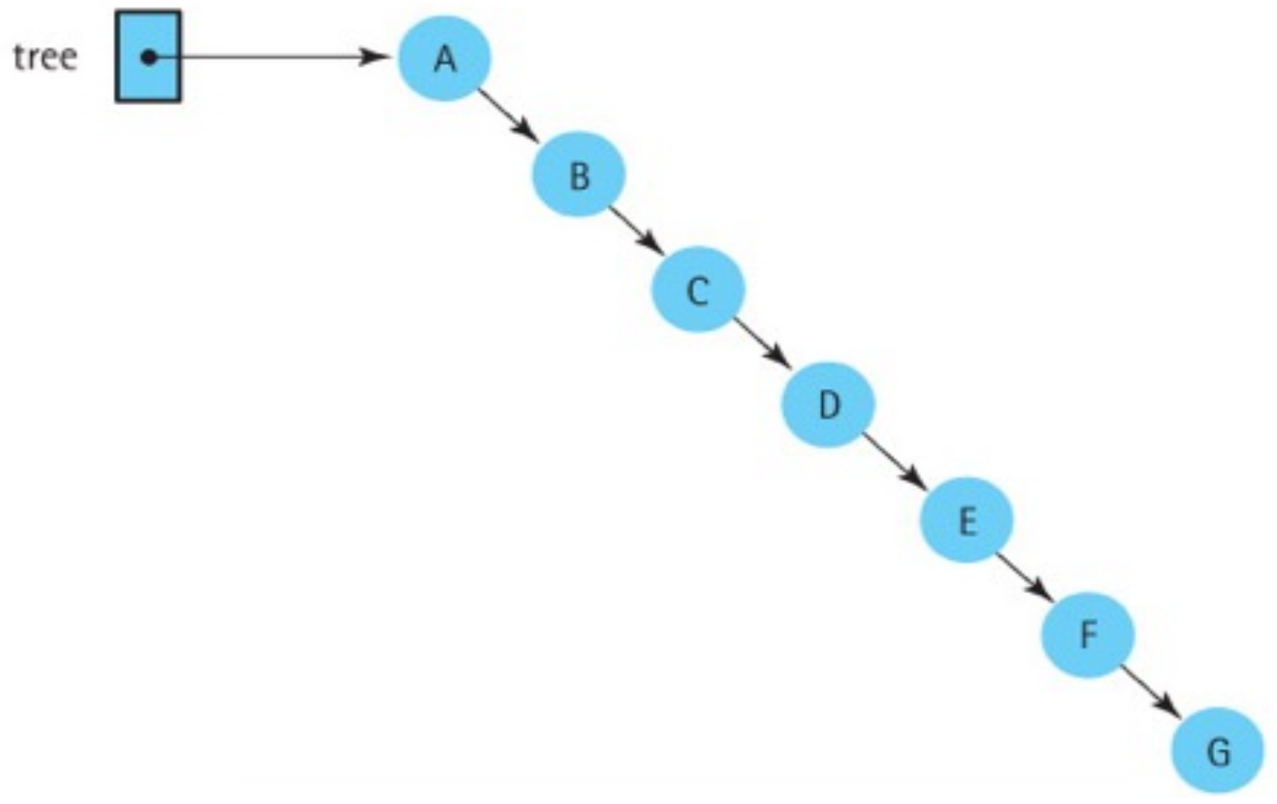
- First, find the node to insert the new element to. This is much the same process as trying to find an element that turns out not to exist.
- Once you’ve found the node, insert the new element as either its left child or right child, depending on the comparison result.
- Note that the new element is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

A Key Point

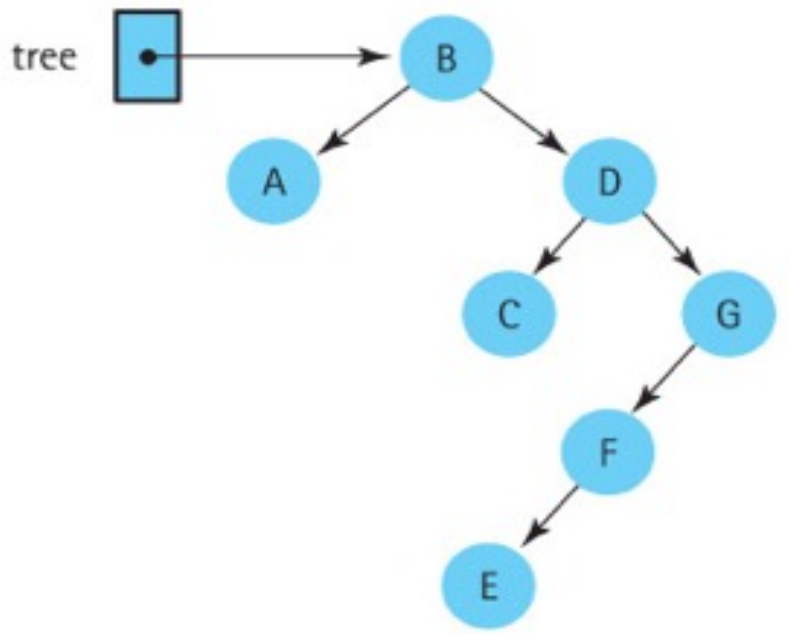
Insertion order determines shape of BST
(and we don't know the order items will come in)



D B F A C E G



A B C D E F G



B A D C G F E

Clicker Question #3

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

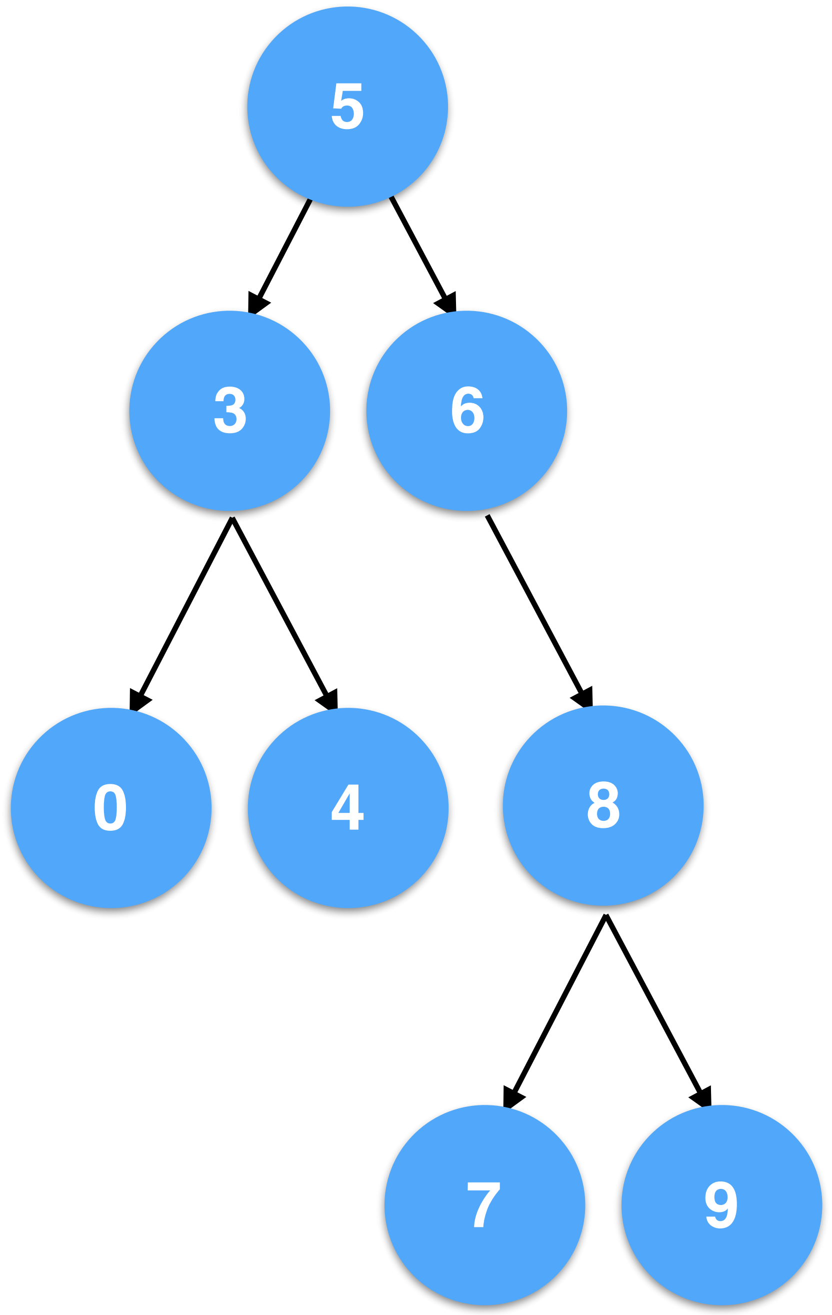
A) 5 6 8 4 3 0 7 9

B) 5 3 4 6 0 8 9 7

C) 5 3 6 0 4 7 8 9

D) All of the above

E) None of the above



The Remove Operation

remove(elem) : remove node containing elem

- The most complicated in BST operations.
- We must ensure when we remove an element that we maintain the binary search tree property.
- No need to memorize the code, but given a BST and a node to remove, you need to be able to 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.

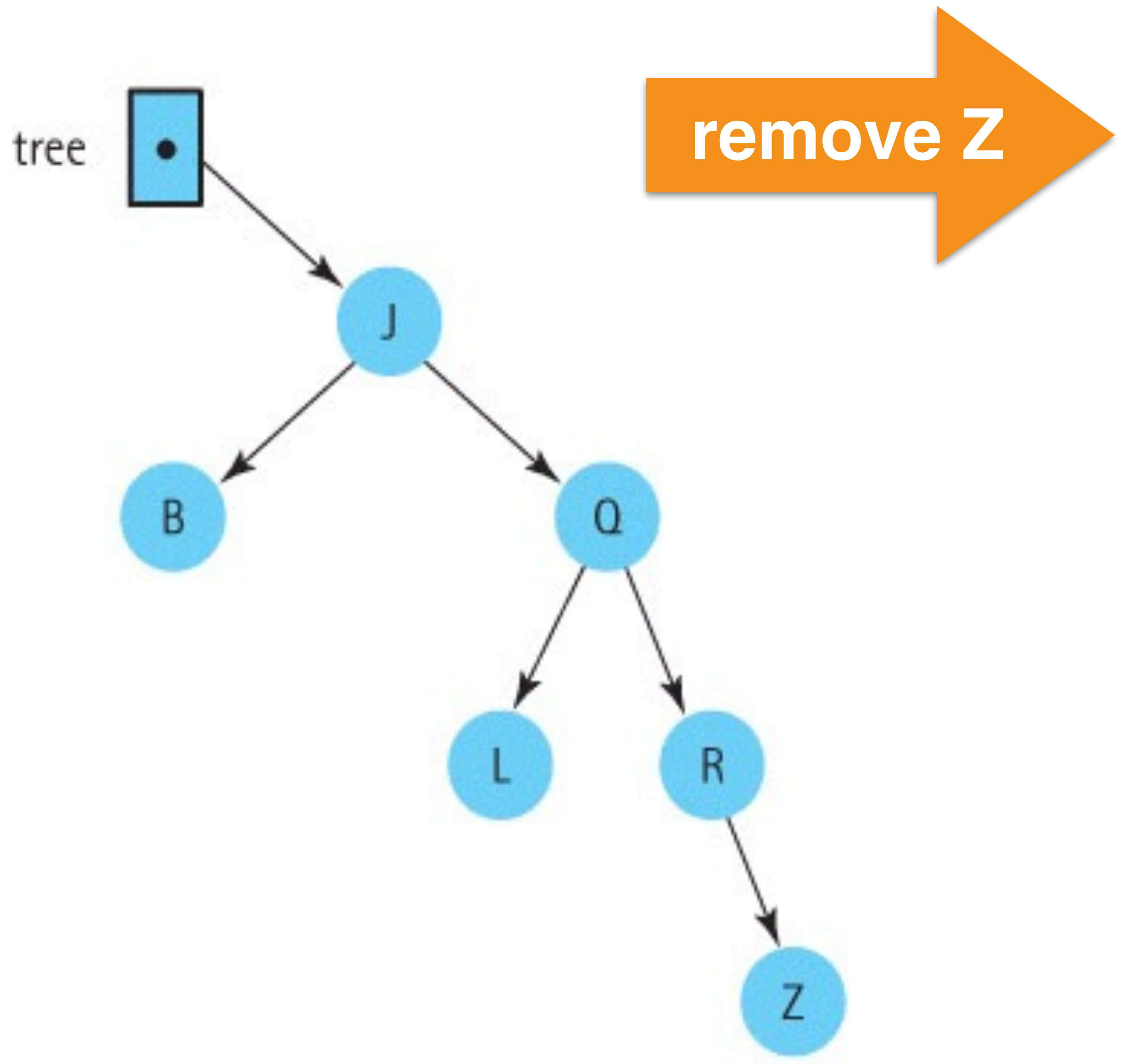
OK

- **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.

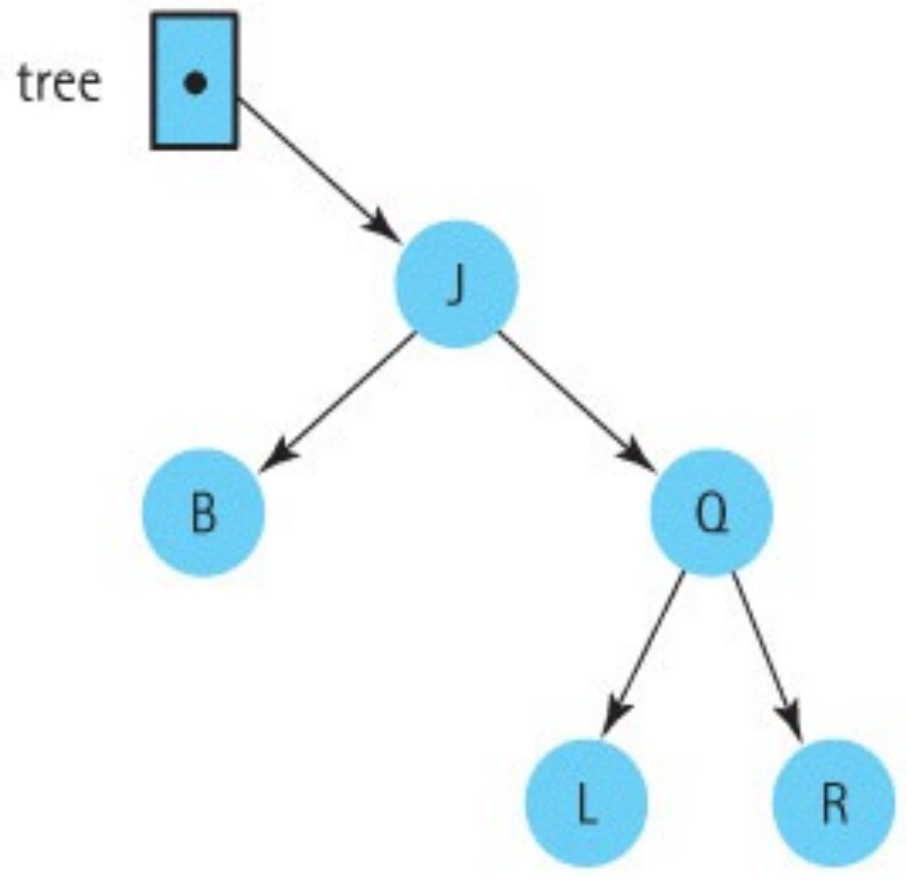
Tricky

- **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.

Removing a Leaf Node

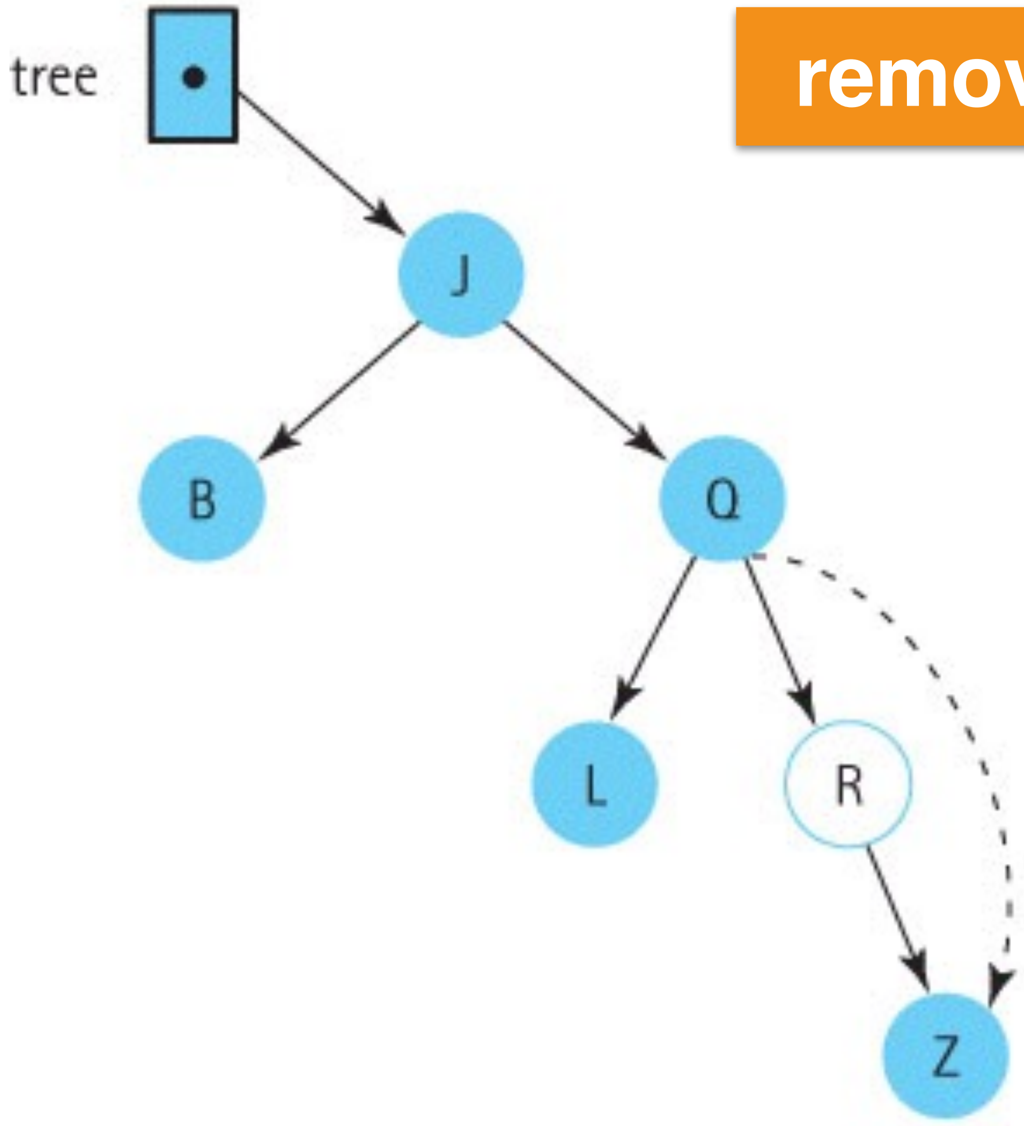


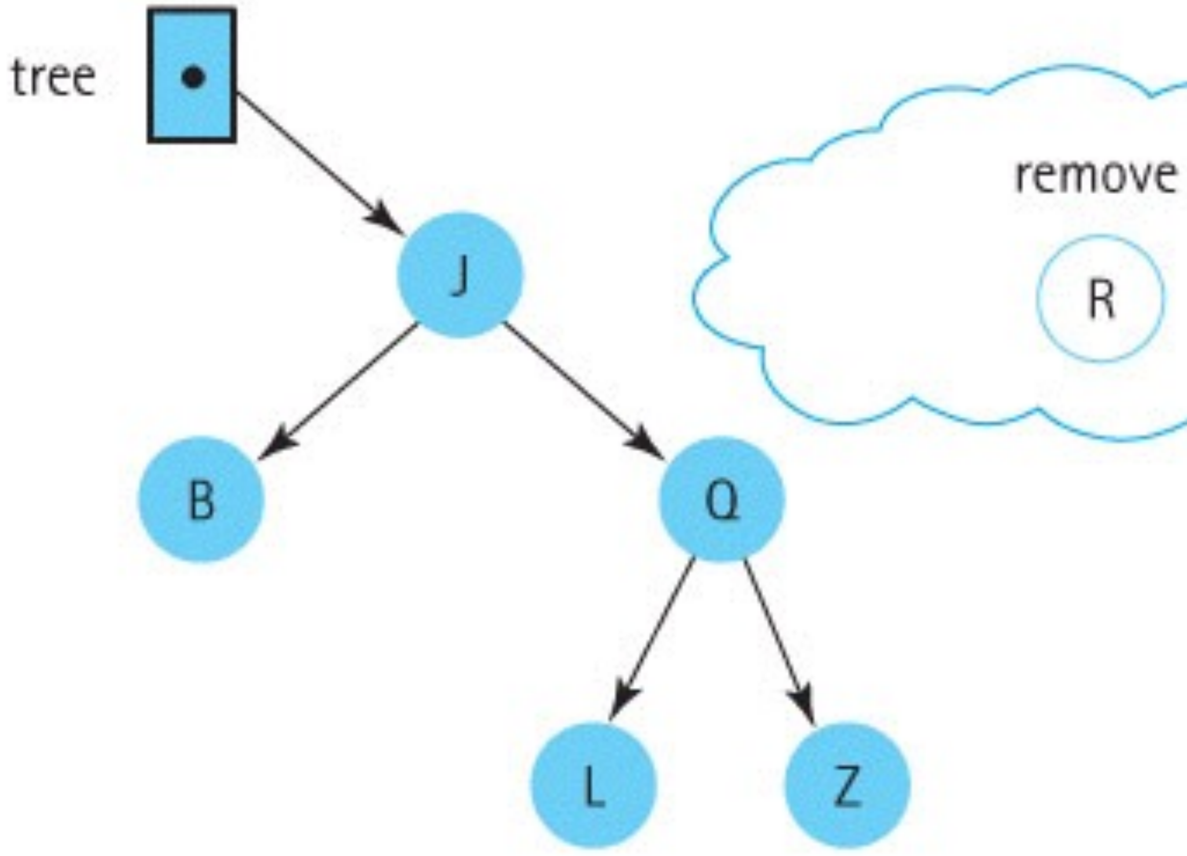
Before



After

Removing a Node with One Child



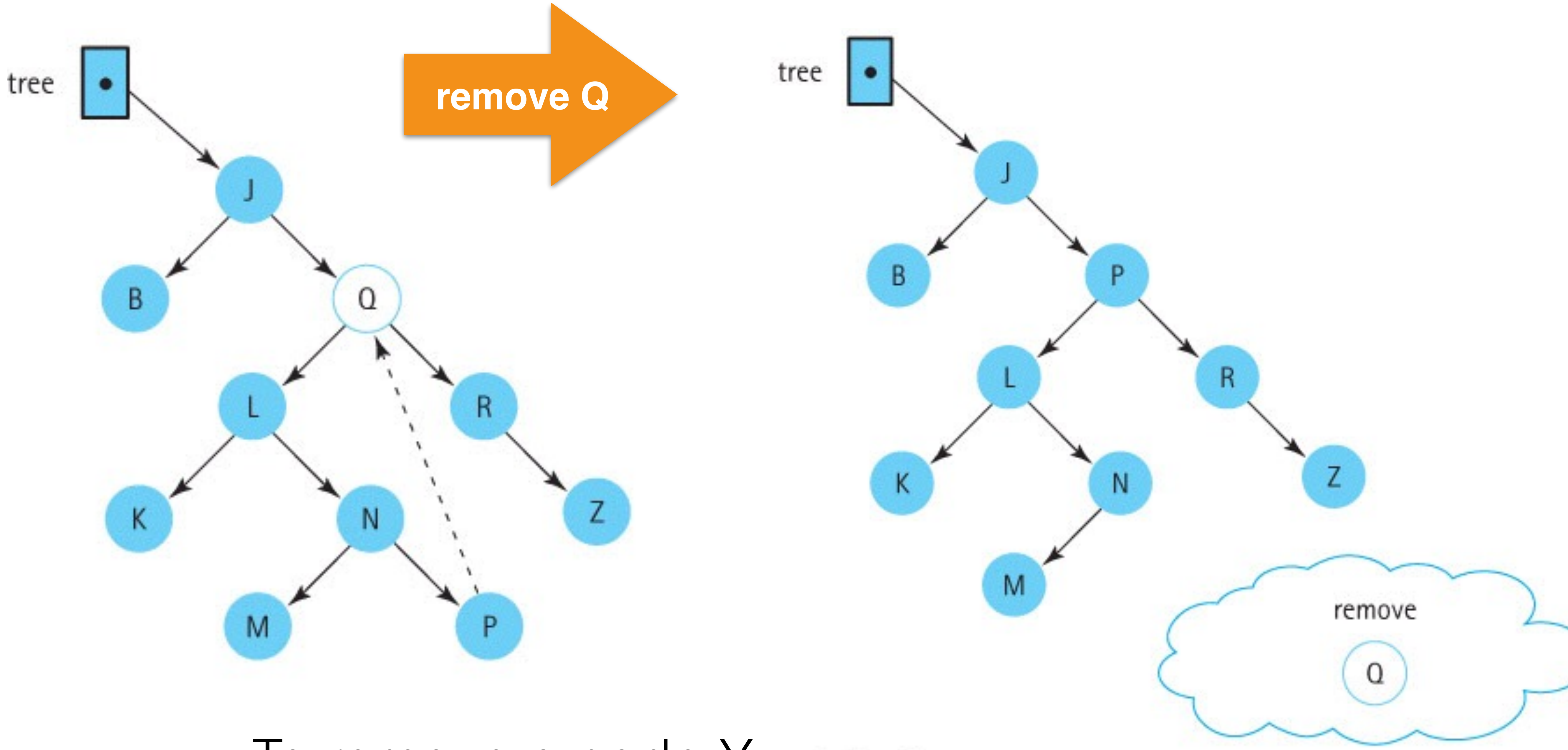


What if R has a left child S and no right child?

Before

After

Removing a Node with Two Children



To remove a node Y:

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

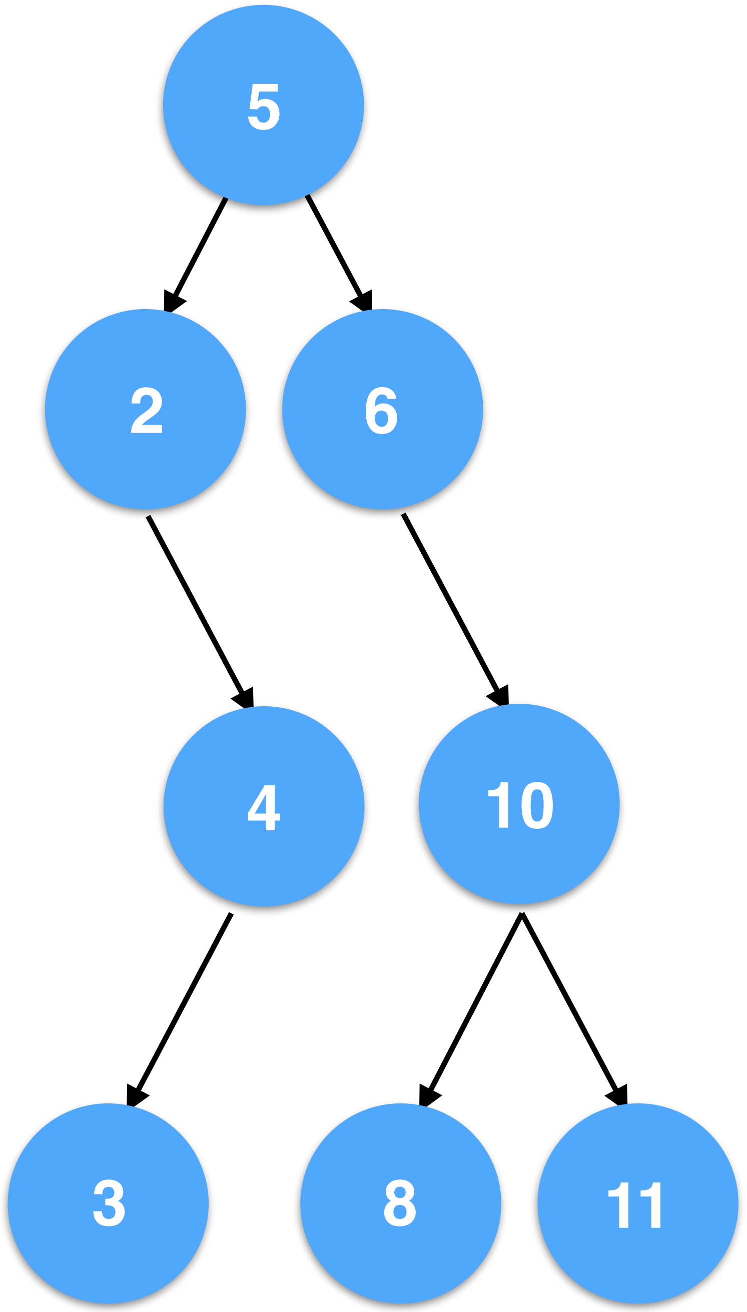
Removing a Node with Two Children

What happens when we remove 5?

What's 5's predecessor?

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

Is it possible that the predecessor has two children again, so there is yet another recursion?



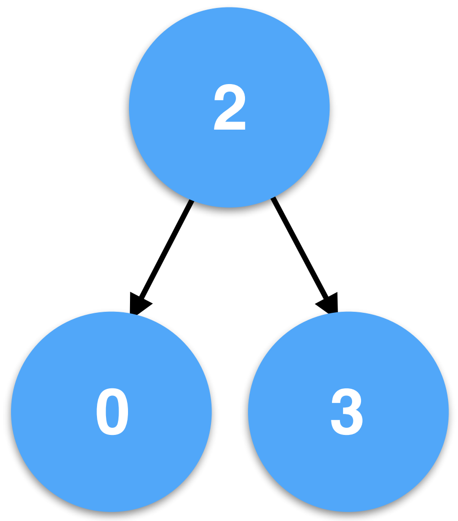
Removing a Node with Two Children

- How do we know the predecessor won't have two children itself?
    - Because it can't 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.
- In stead 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.

Do at Your Seats

- What does this BST look like after the following operations:
    - remove 2
    - add 2
    - add 5
    - add 1
    - remove 3



The **contains** method

```
public boolean contains (T element) {
  return recContains(element, root);
}
private boolean recContains(T element, BSTNode<T> tree) {
  // Returns true if tree contains an element e such that
  // e.compareTo(element) == 0; otherwise, returns false.
  if (tree == null)
    return false;        // element is not found
  else if (element.compareTo(tree.getInfo()) < 0)
    // Search left subtree
    return recContains(element, tree.getLeft());
  else if (element.compareTo(tree.getInfo()) > 0)
    // Search right subtree
    return recContains(element, tree.getRight());
  else
    return true;         // element is found
}
```

The **get** method

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

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> tree {
  if (tree==null) return 0;
  else
    return 1 + recSize(tree.getLeft())
             + recSize(tree.getRight());
}
```

Trade-offs: Ordered Structures

- **Sorted Array:**
  - SEARCH: can leverage binary search —> O(log n)
  - ADD: hard to maintain sorted order —> O(n) to insert
  - harder to make unbounded.
- **Binary Search Tree (BST):** (let **h** be the height of the tree)
  - SEARCH: efficient search —> O(h)
  - ADD: efficient maintenance —> O(h)
  - linked structure means its unbounded.

If the BST is reasonably balanced,
h is O(log n), hence BST wins.