



Middlebury

CSCI 201: Data Structures

Fall 2024

Lecture 9T: Binary Search Trees

Goals for today:

- **Motivation:** how to know if a collection **contains** an item (efficiently)? How to maintain a unique, ordered set of items?
- Use a **TreeSet** to maintain an ordered set of keys.
- Use a **TreeMap** to maintain an ordered set of key-value pairs.
- Implement your own **BinarySearchTree**:
 - **add** an item to a binary search tree.
 - **remove** an item from a binary search tree.
 - Check if a BST **contains** a key.
 - Use a BST to **get** the value associated with a key.



Motivation: does a collection **contains** an item?

- **ArrayList:**  $O(n)$

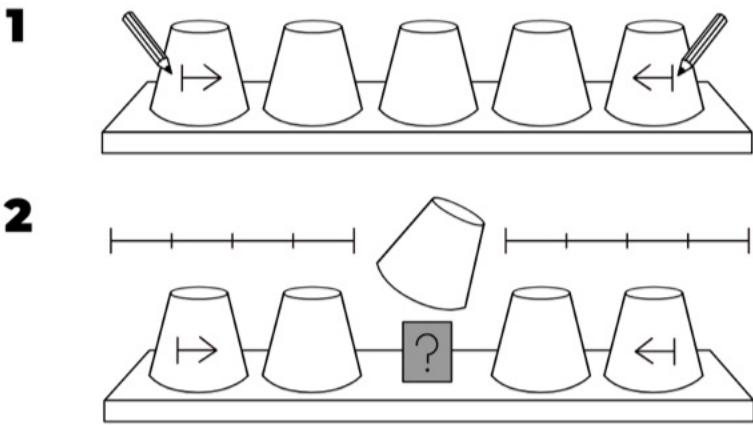
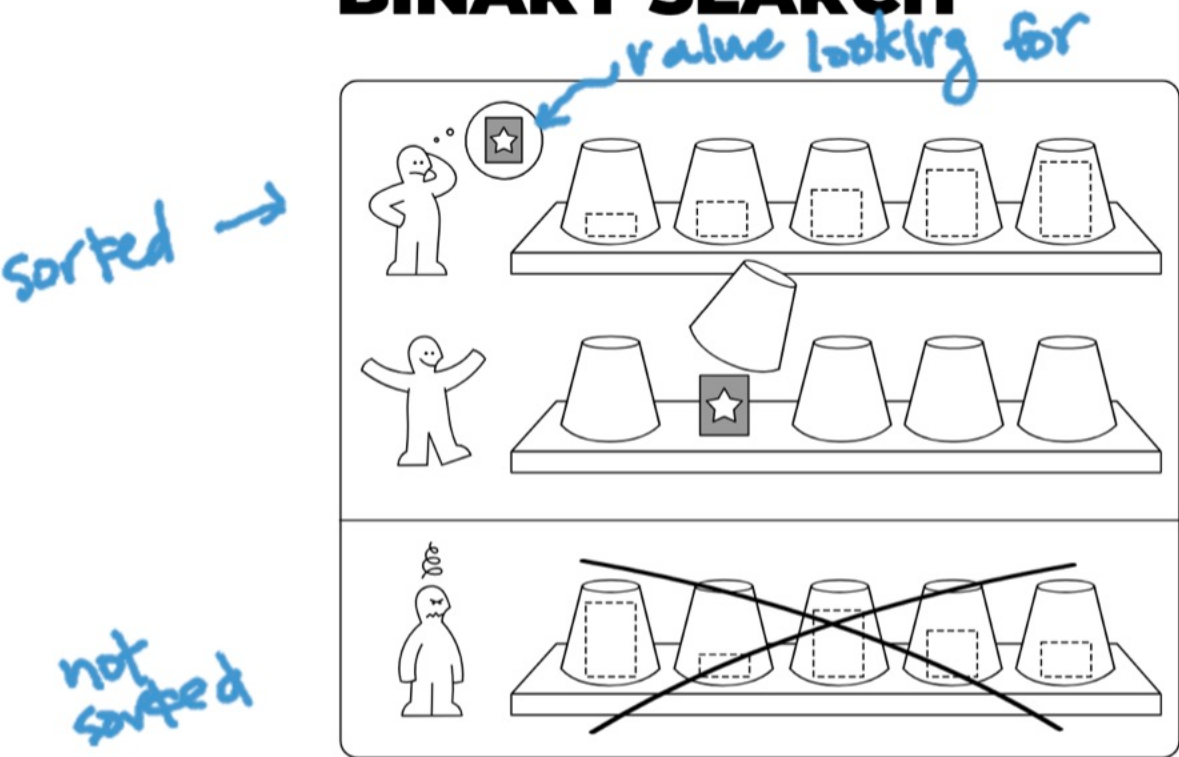
- **LinkedList:**



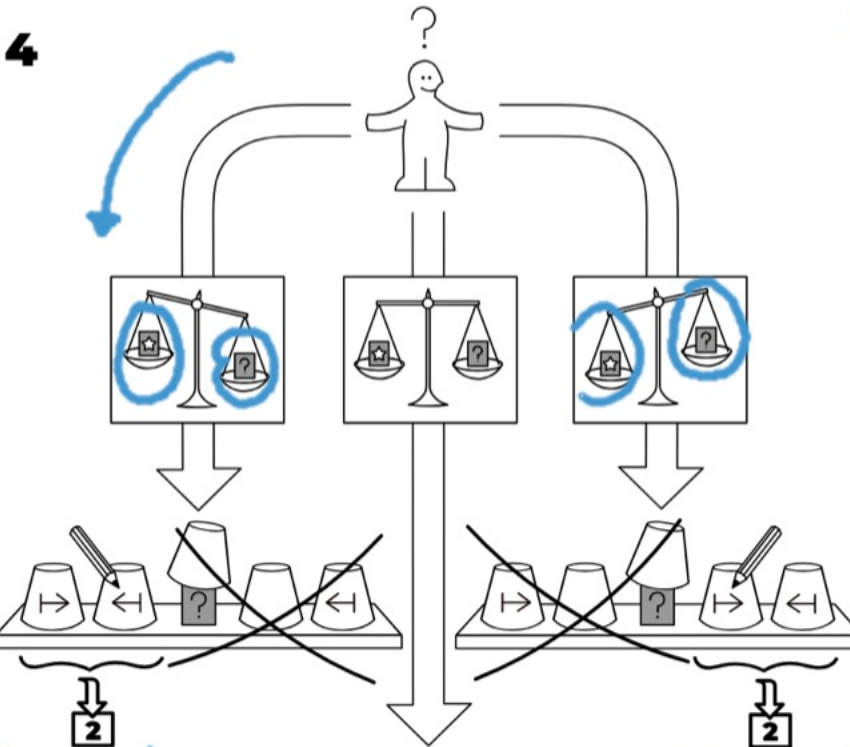
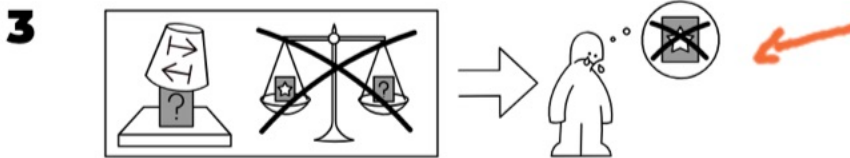
💡 **Idea:** maintain sorted (ordered) items, perform binary search.

Inspiration: binary search to see if a sorted array has an item.

BINÄRY SEARCH



idea-instructions.com/binary-search/
v1.1, CC by-nc-sa 4.0 **IDEA**



search left part of array

search right part of array.

complexity of binary search?
 $O(\log n)$

Binary search on our favorite data structures?

With the requirement that we can **add** or **remove** items.

- **ArrayList:**  add: put item in right place in sorted array $O(n)$

- **LinkedList:**



add: put item in right place
(requires finding right place, node) $O(n)$

💡 **Idea:** linked structure, maintain order between nodes.

How **Java** does it: trees!

Main things we want:

- Determine if this key exists in the BST (**contains**, **containsKey**).
- Ability to **add** (or **put**) a new item into a tree.
- Ability to **remove** an item from the tree.

```
1 import java.util.*;
2
3 public class TreeSetMapExamples {
4
5     public static void main(String[] args) {
6         TreeSet<Integer> set = new TreeSet<>();
7
8         set.add(12);
9         set.add(8);
10        set.add(14);
11        set.add(5);
12        set.add(9);
13        set.add(20);
14
15        System.out.println(set);
16        for (int item : set) {
17            System.out.println(item);
18        }
19        System.out.println(set.contains(12));
20        System.out.println(set.contains(13));
21    }
22 }
```

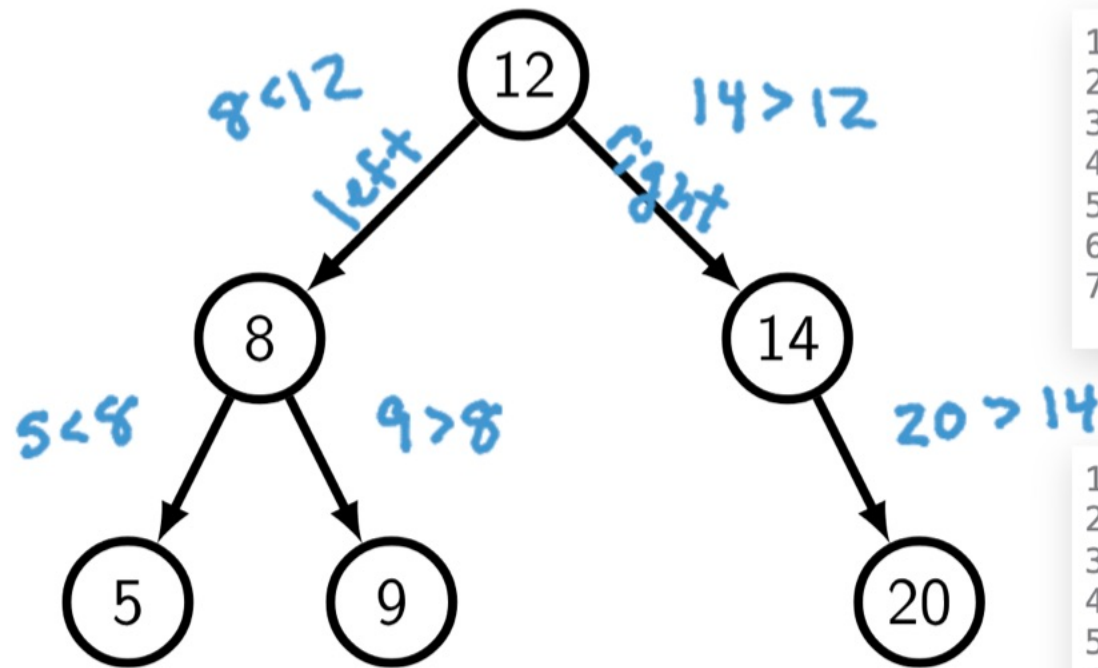
```
1 import java.util.*;
2
3 public class TreeSetMapExamples {
4
5     public static void main(String[] args) {
6         TreeMap<Character, Integer> map
7             = new TreeMap<>();
8
9         String phrase = "bananabread";
10        for (int i = 0; i < phrase.length(); i++) {
11            Character c = phrase.charAt(i);
12            if (map.containsKey(c)) {
13                map.put(c, map.get(c) + 1);
14            } else {
15                map.put(c, 1);
16            }
17        }
18        System.out.println(map);
19        System.out.println(map.containsKey('b'));
20        System.out.println(map.containsKey('f'));
21    }
22 }
```

Binary Search Trees (BST): maintain order between left and right child nodes (subtrees) in relation to each node.

Main things we want:

- Determine if this key exists in the BST (**contains**, **containsKey**).
- Ability to **add** (or **put**) a new item into a tree.
- Ability to **remove** an item from the tree.

Main idea: all items (keys) in left subtree are $<$ item (key) at current node; all items (keys) in right subtree are $>$ item (key) at current node.



```
1 class BSTSet<E extends Comparable<E>> {
2   class Node {
3     public E key;
4     public Node left;
5     public Node right;
6   }
7 }
```

```
1 class BSTMap<K extends Comparable<K>, V> {
2   class Node {
3     public K key;
4     public V value;
5     public Node left;
6     public Node right;
7   }
8 }
```

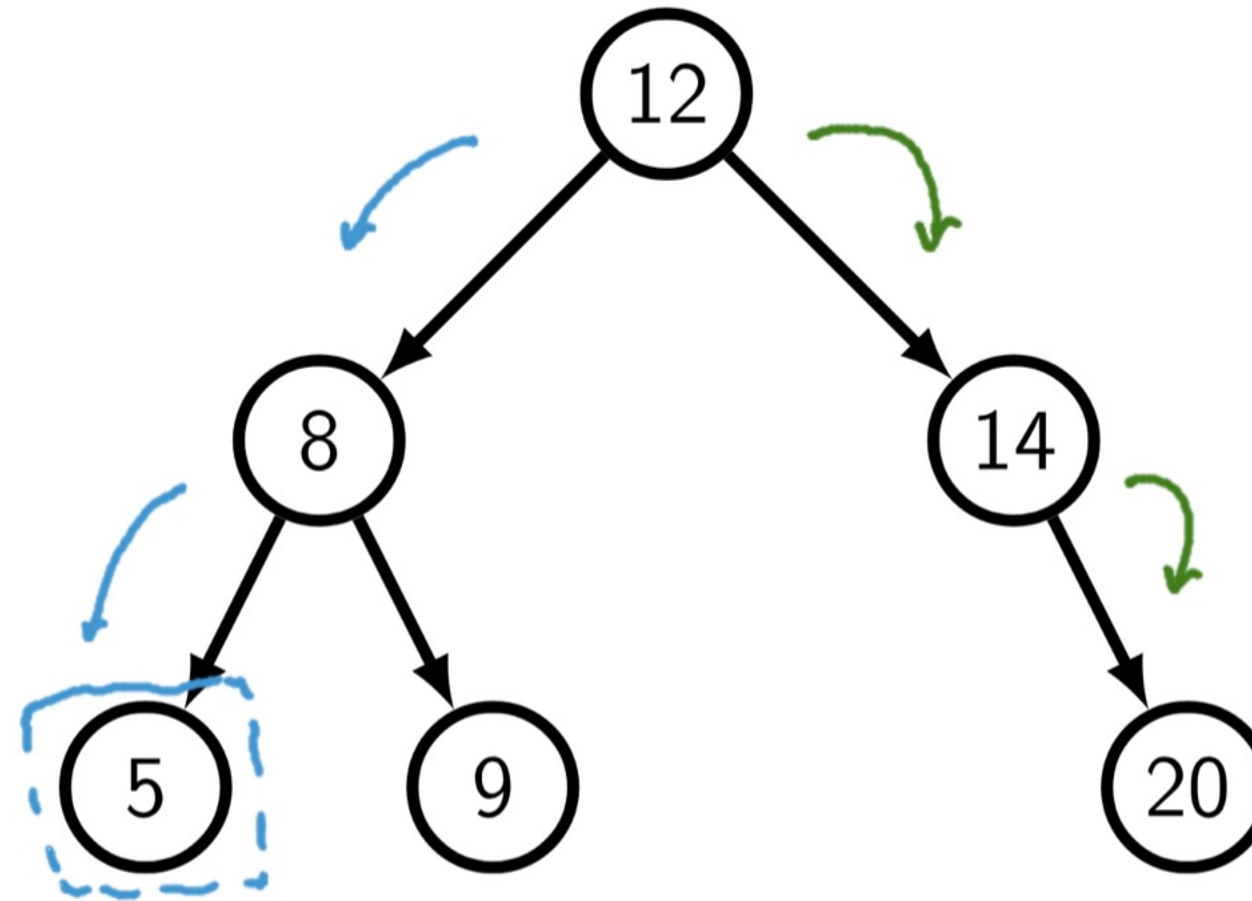
We need
to be able
to compare
keys/items/elements

$<$, $>$, $==$

compareTo

$<$ $>$

Retrieving the minimum & maximum item/key in a BST.

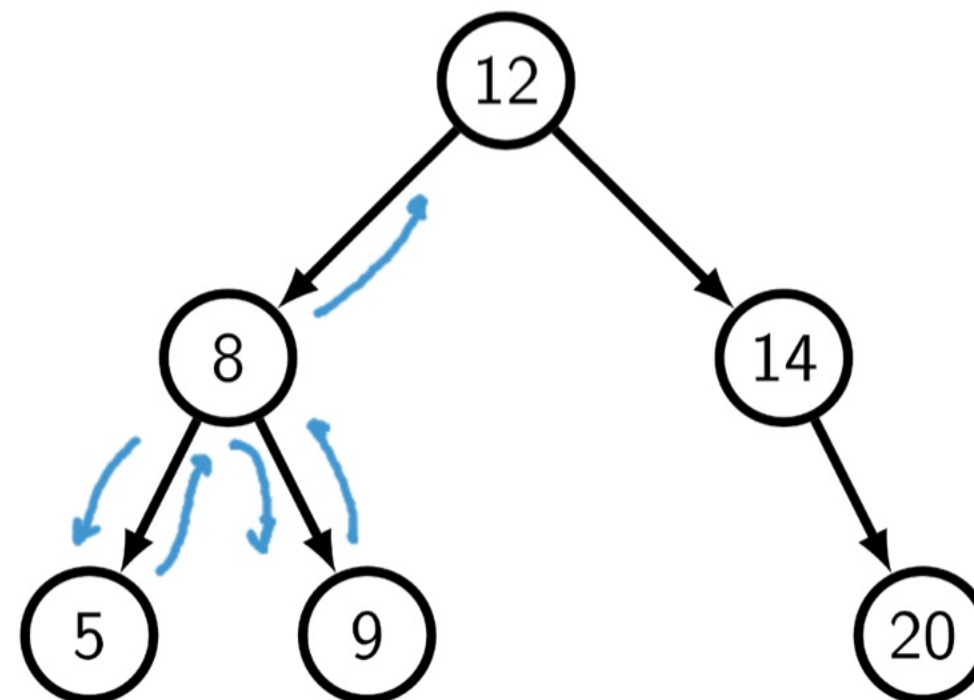
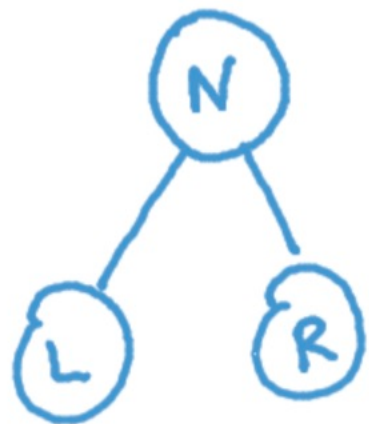


```
1 public E getMin() {  
2     Node node = root;  
3     while (node.left != null) {  
4         node = node.left;  
5     }  
6     return node.key;  
7 }
```

```
1 public E getMax() {  
2     Node node = root;  
3     while (node.right != null) {  
4         node = node.right;  
5     }  
6     return node.key;  
7 }
```

≡ Which of the following would process the items (keys) from a BST in sorted order (smallest to largest)? (slido.com #3128533)

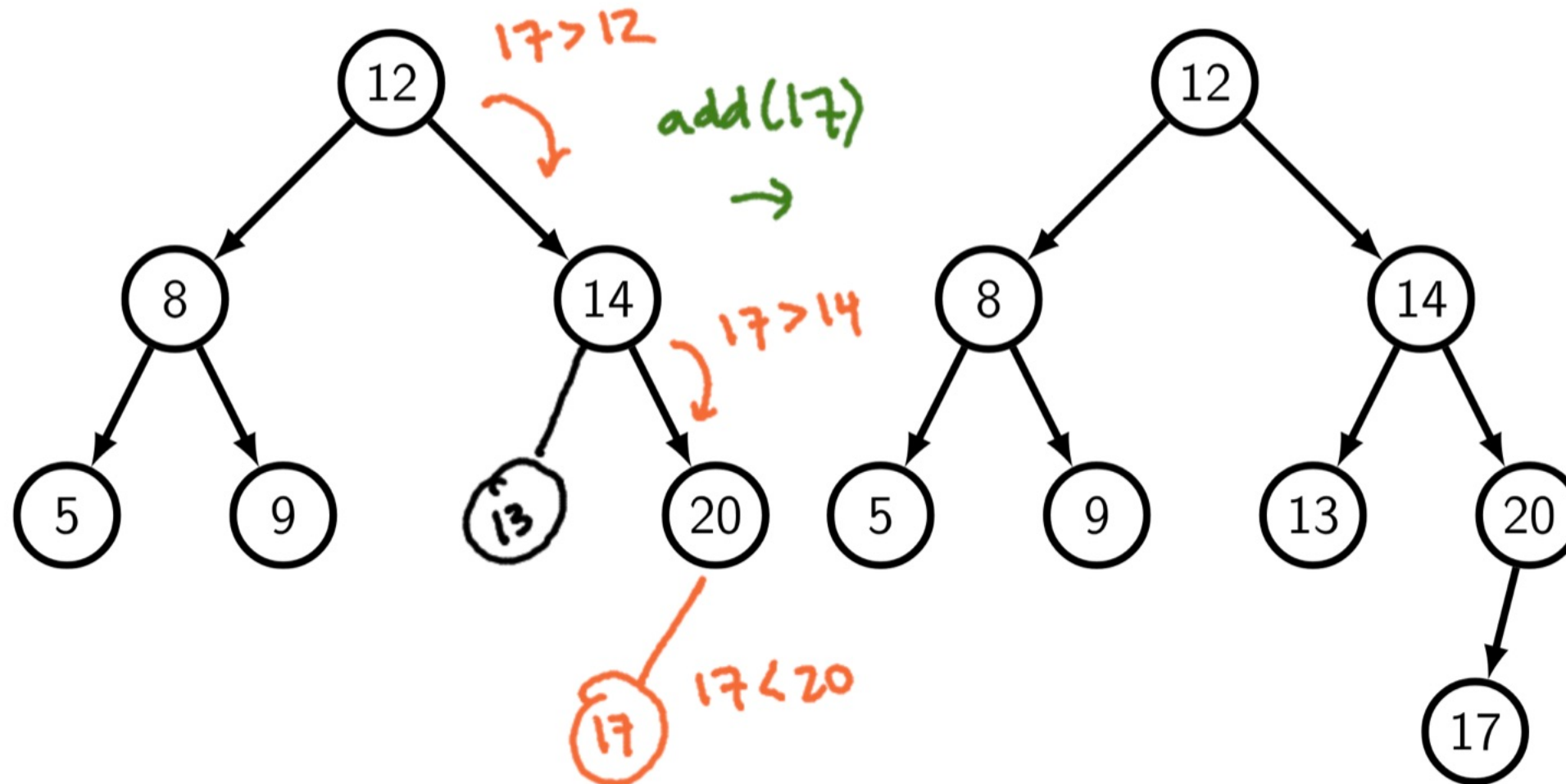
19 ⦿

☐ Pre-order traversal.☒ In-order traversal.☐ Post-order traversal.☐ Level-order traversal.

5, 8, 9, 12, 14, 20

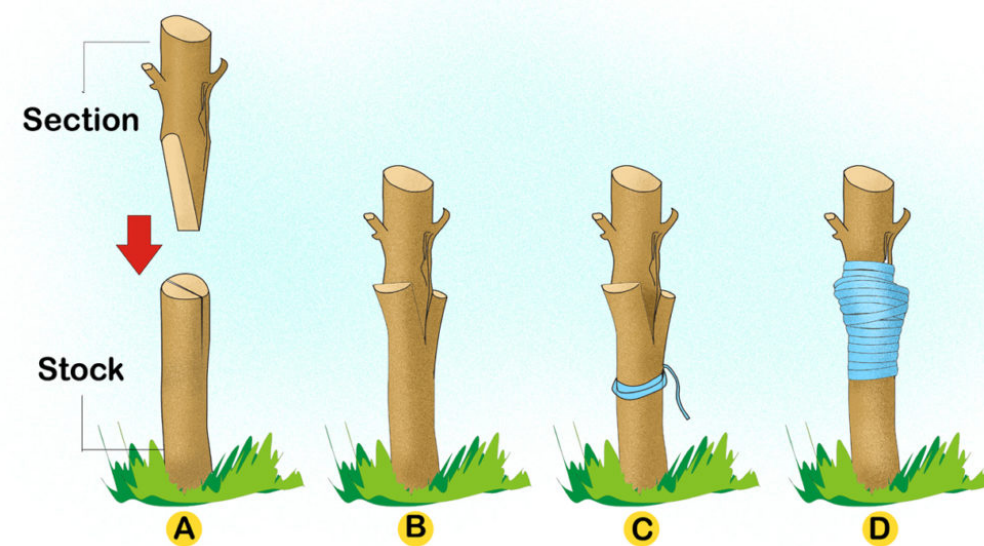
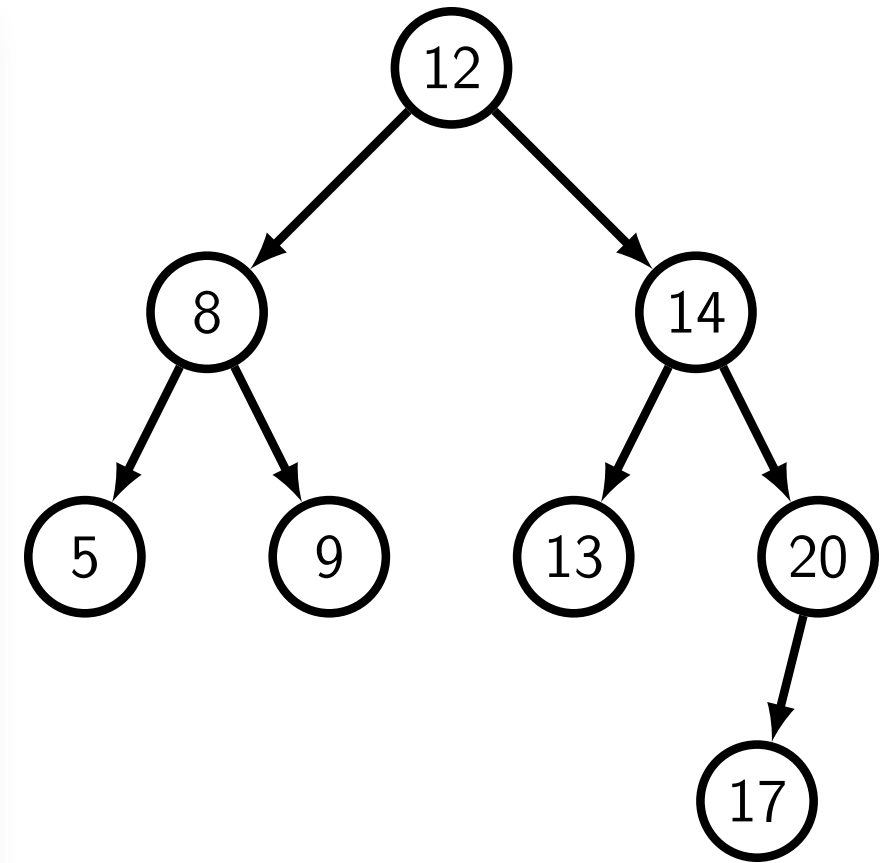
Adding a **key** to a BST (**add**, **put**).

- Start at **node = root**.
- If **key < node.key**, need to put in left subtree (**node.left**).
- If **key > node.key**, need to put in right subtree (**node.right**).
- If **key == node.key**, key is already in tree! Done.
- If **node == null**, create a new **Node** to hold this **key**.



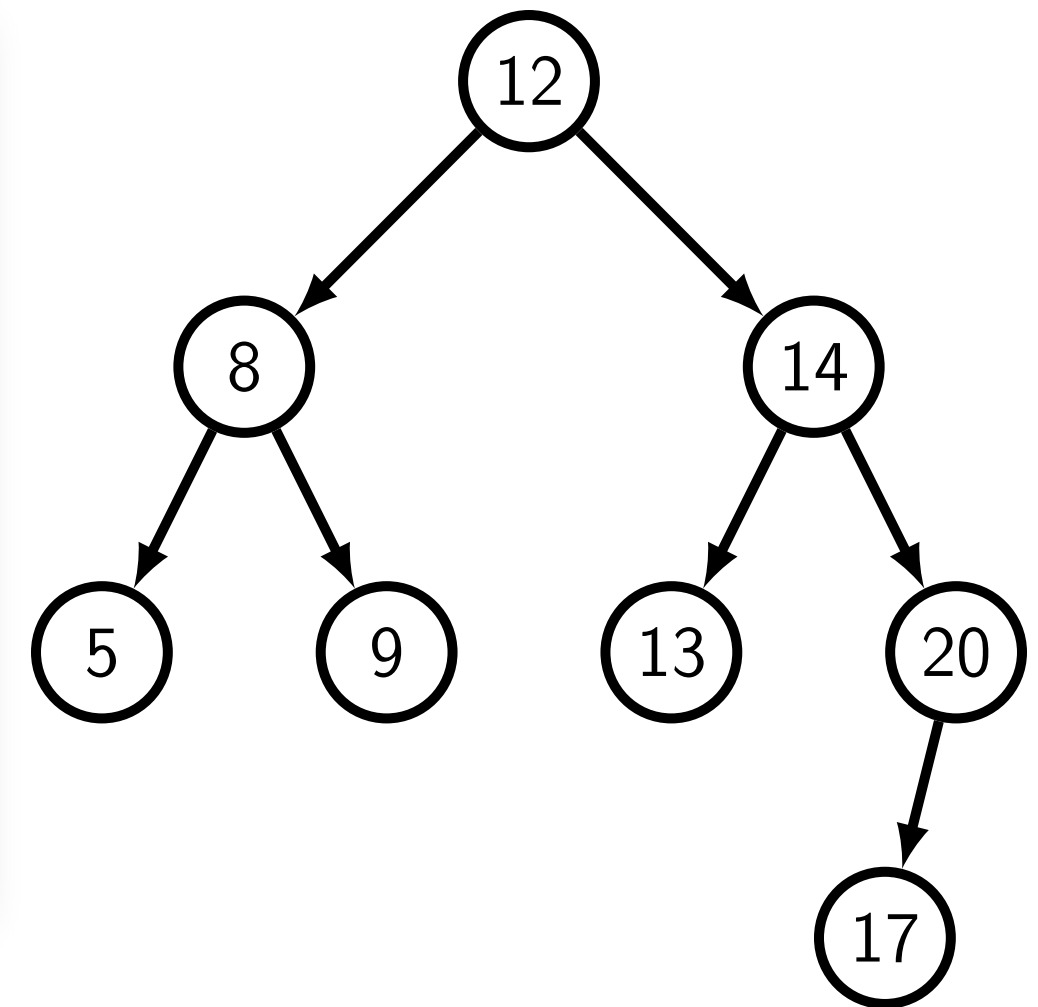
Implementing the **add** method.

```
1 public void add(E key) {  
2     root = add(root, key);  
3 }  
4  
5 private Node add(Node node, E key) {  
6     if (node == null) {  
7         // last node was a leaf, create the new leaf node  
8         return new Node(key);  
9     }  
10  
11     int compareResult = key.compareTo(node.key);  
12     if (compareResult < 0) {  
13         node.left = add(node.left, key);  
14     } else if (compareResult > 0) {  
15         node.right = add(node.right, key);  
16     }  
17     return node;  
18 }
```



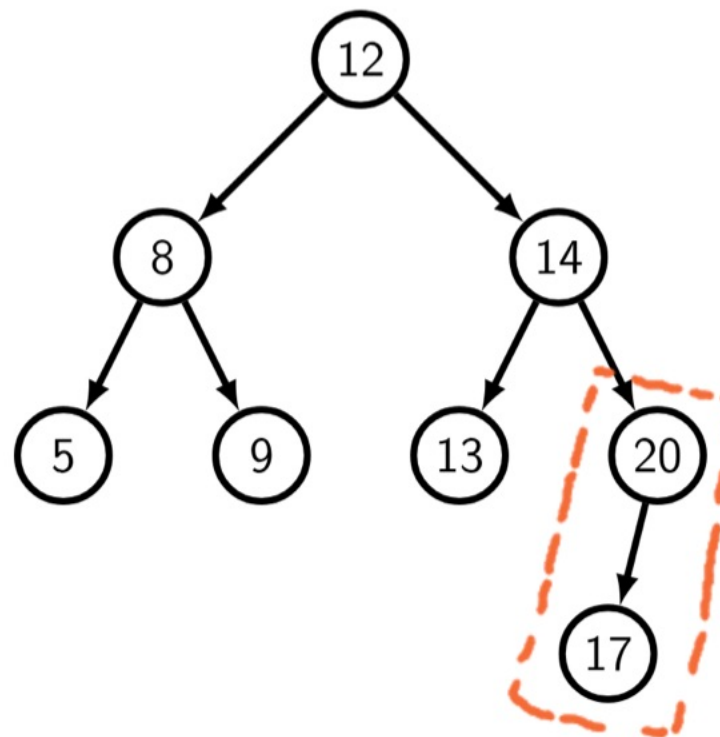
Exercise: implement the **contains** method and test it!

```
1 public boolean contains(E key) {  
2     return contains(root, key);  
3 }  
4  
5 private boolean contains(Node node, E key) {  
6     if (node == null) return false;  
7  
8     int compareResult = key.compareTo(node.key);  
9     if (compareResult == 0) {  
10         return true;  
11     } else if (compareResult < 0) {  
12         return contains(node.left, key);  
13     } else {  
14         return contains(node.right, key);  
15     }  
16 }
```



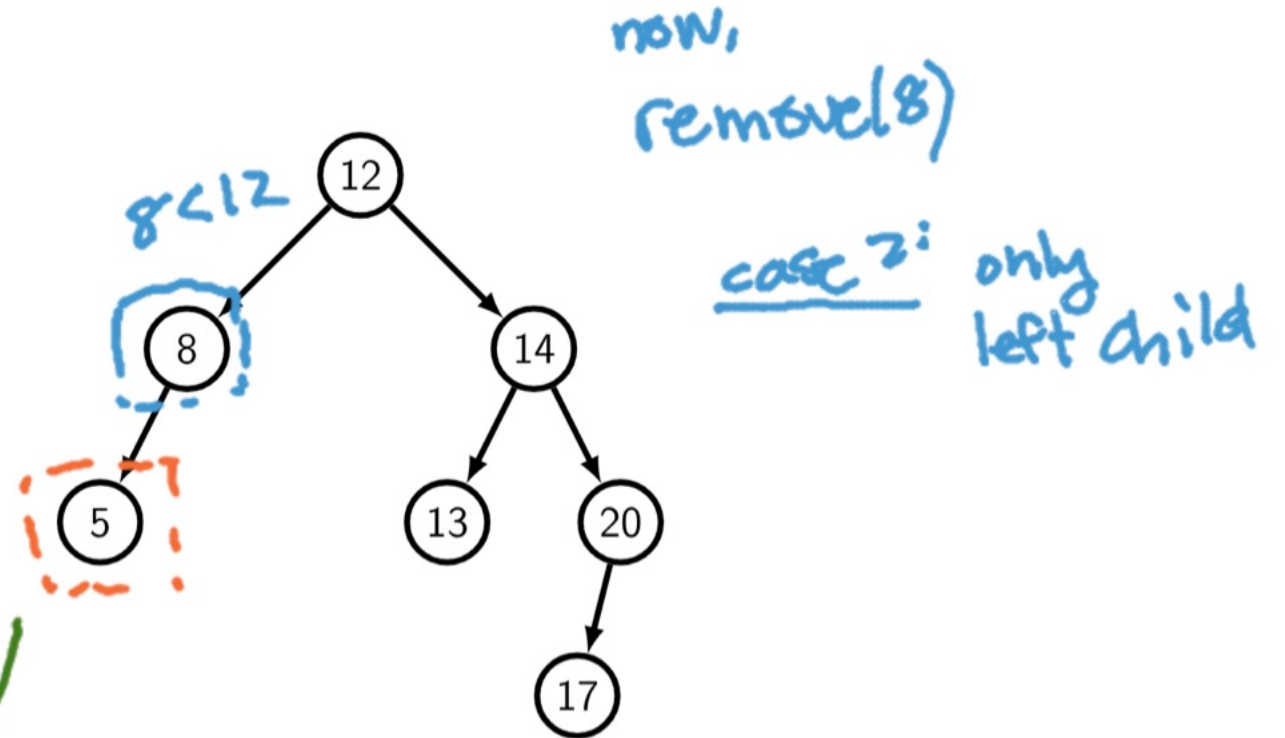
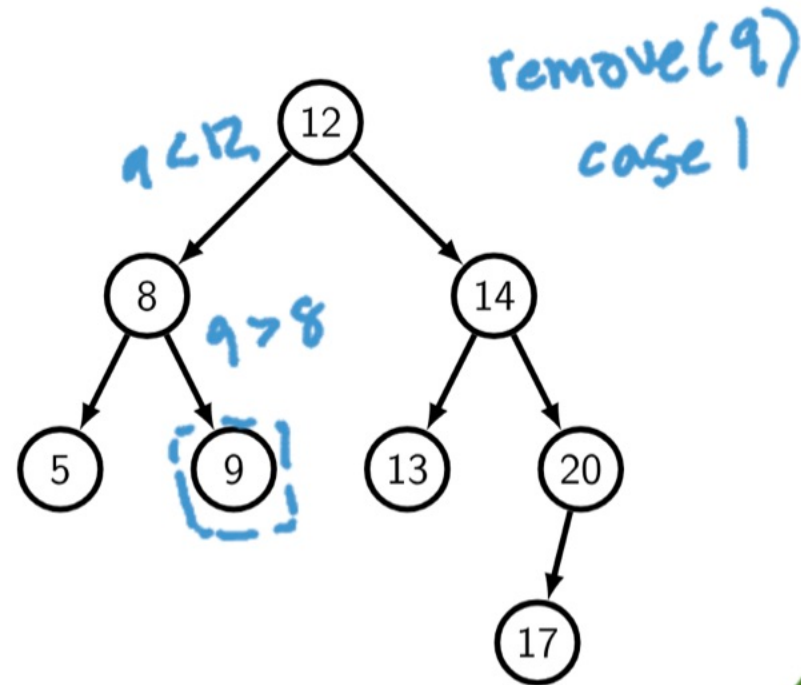
Removing an item/**key** from a BST:

- Start at **node = root**.
- If **key < node.key**, need to remove node in left subtree (**node.left**).
- If **key > node.key**, need to remove node in right subtree (**node.right**).
- If **key == node.key**, three cases to consider:
 1. Is this a leaf? Delete node.
 2. (a) If **node.left == null**, "graft" **node.right** to the **node**.
(b) If **node.right == null**, "graft" **node.left** to the **node**.
 3. Two (non-**null**) child nodes?
 - Find node (**minNode**) with minimum **key** in **node.right** (right subtree).
 - Replace **node.key** with **minNode.key**.
 - Now we remove **minNode** from right subtree.

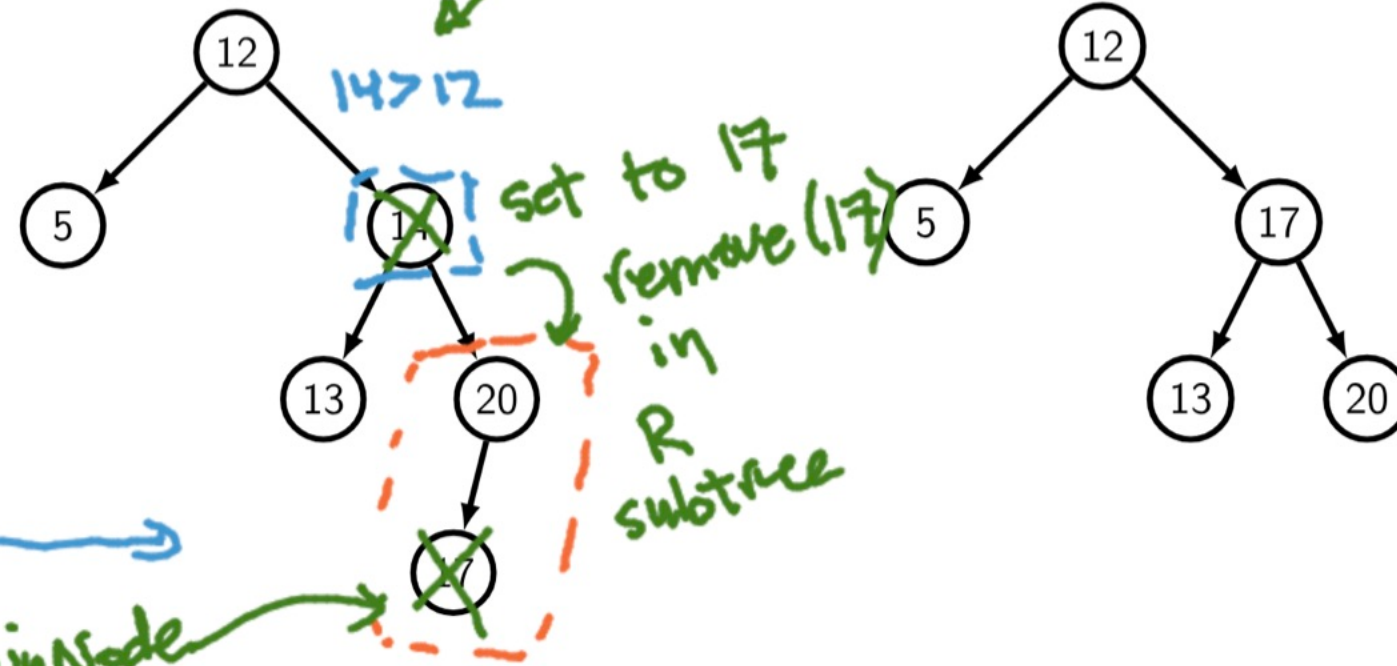


case 3:
removal in right
subtree is
either case 1 or
case 2 removal.

remove examples.



case 3:
now remove (14)
1. find min in right subtree



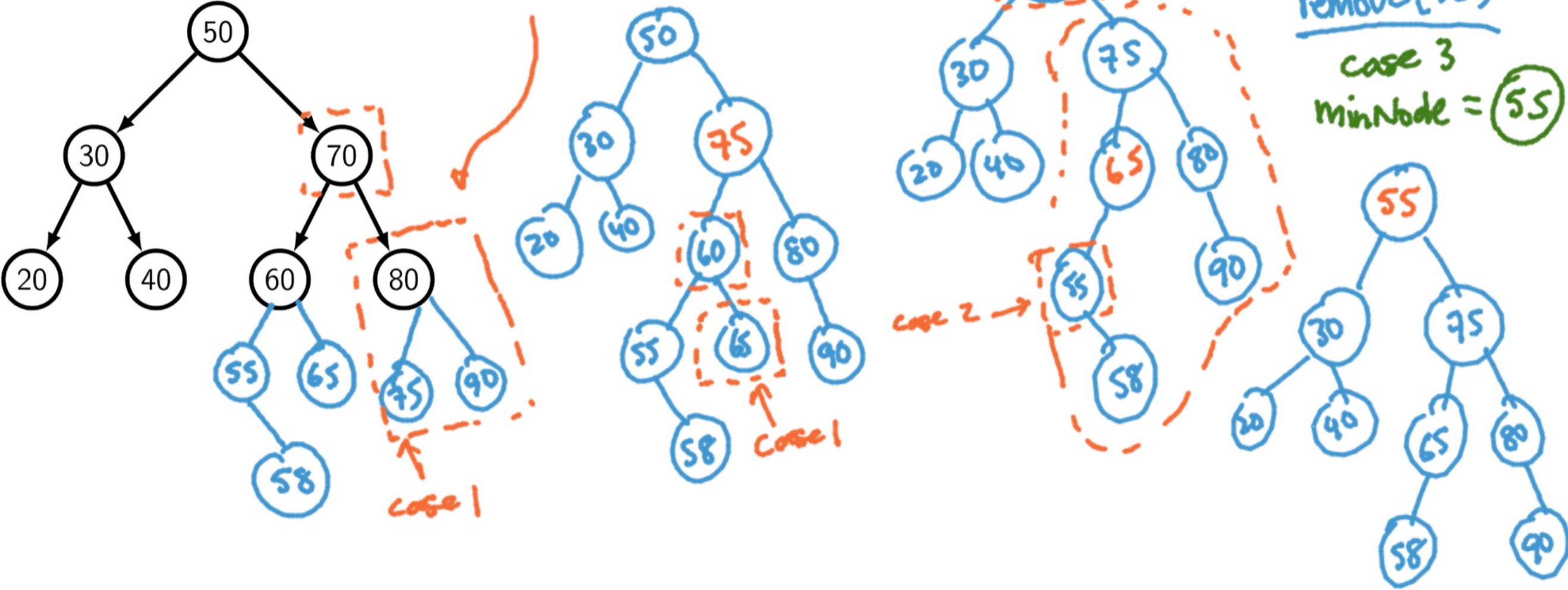
always a case 1 or 2 removal

minNode

Exercise: draw the tree after the following calls to **add** and **remove**.

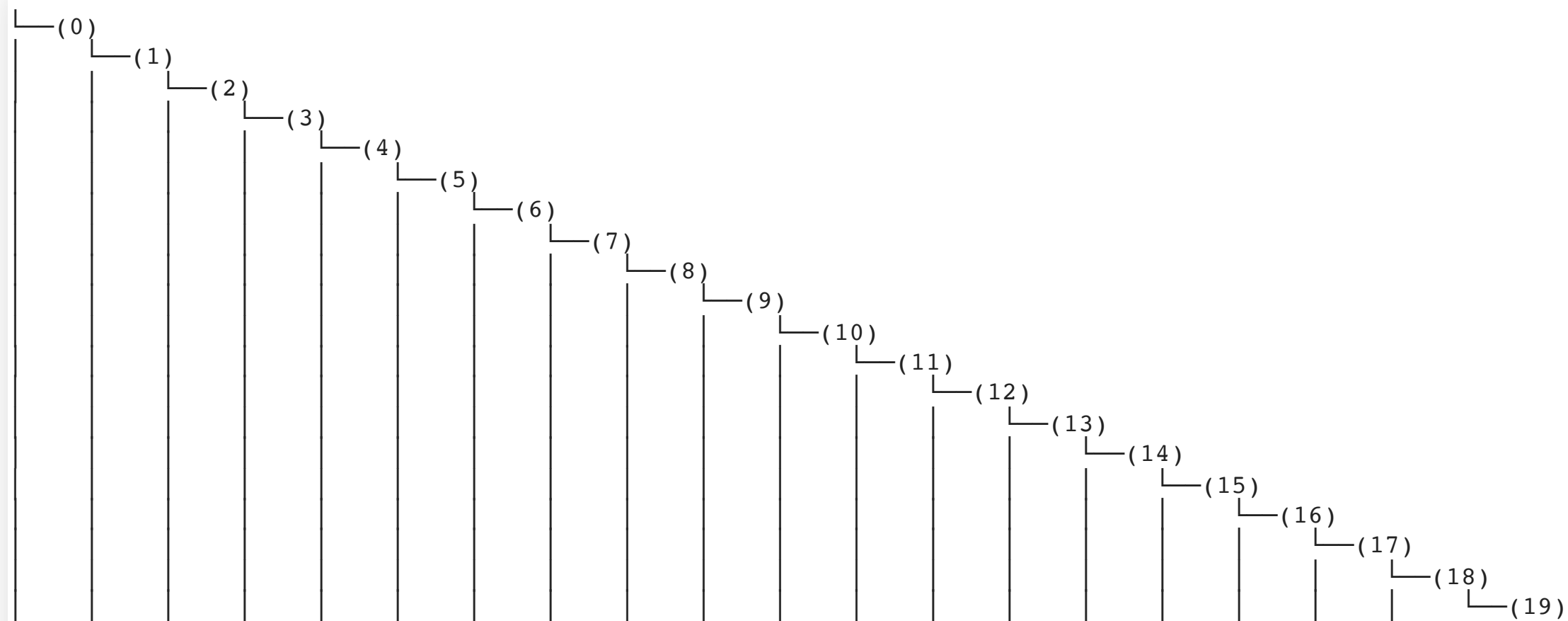
Work on the whiteboards and take a picture of your work for today's exit ticket.

- **add**: 55, 65, 58, 75, 90
- then **remove**: 70, 60, 50



What happens if we do this??

```
1 int n = 20;  
2 for (int i = 0; i < n; i++) {  
3     tree.add(i);  
4 }  
5 System.out.println(tree);
```



Additional notes:

- Techniques to *balance* trees next class!
- **Homework 7** due on Friday 11/8: implement a max-heap.
- There is also an optional Homework 7 challenge problem.
- See Oracle's documentation for **TreeSet**:
<https://docs.oracle.com/javase/8/docs/api/java/util/TreeSet.html>
- See Oracle's documentation for **TreeMap**:
<https://docs.oracle.com/javase/8/docs/api/java/util/TreeMap.html>
- Submit ET 9T by the end of today: submit a picture of your work for **add/remove** exercise.