

CS350: Data Structures

Binary Search Trees

James Moscola

Department of Engineering & Computer Science

York College of Pennsylvania

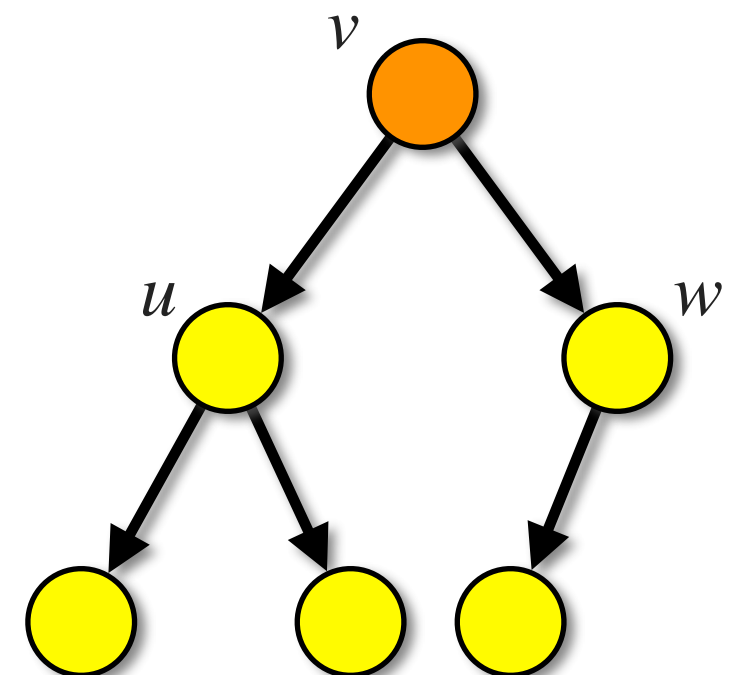


Introduction to Binary Search Trees

- **A binary search tree is a binary tree that stores keys (or key-element pairs) in such a way as to satisfy the following:**
 - For every node X in the tree, the values of all the keys in the left subtree are smaller than the key in X
 - For every node X in the tree, the values of all the keys in the right subtree are larger than the key in X

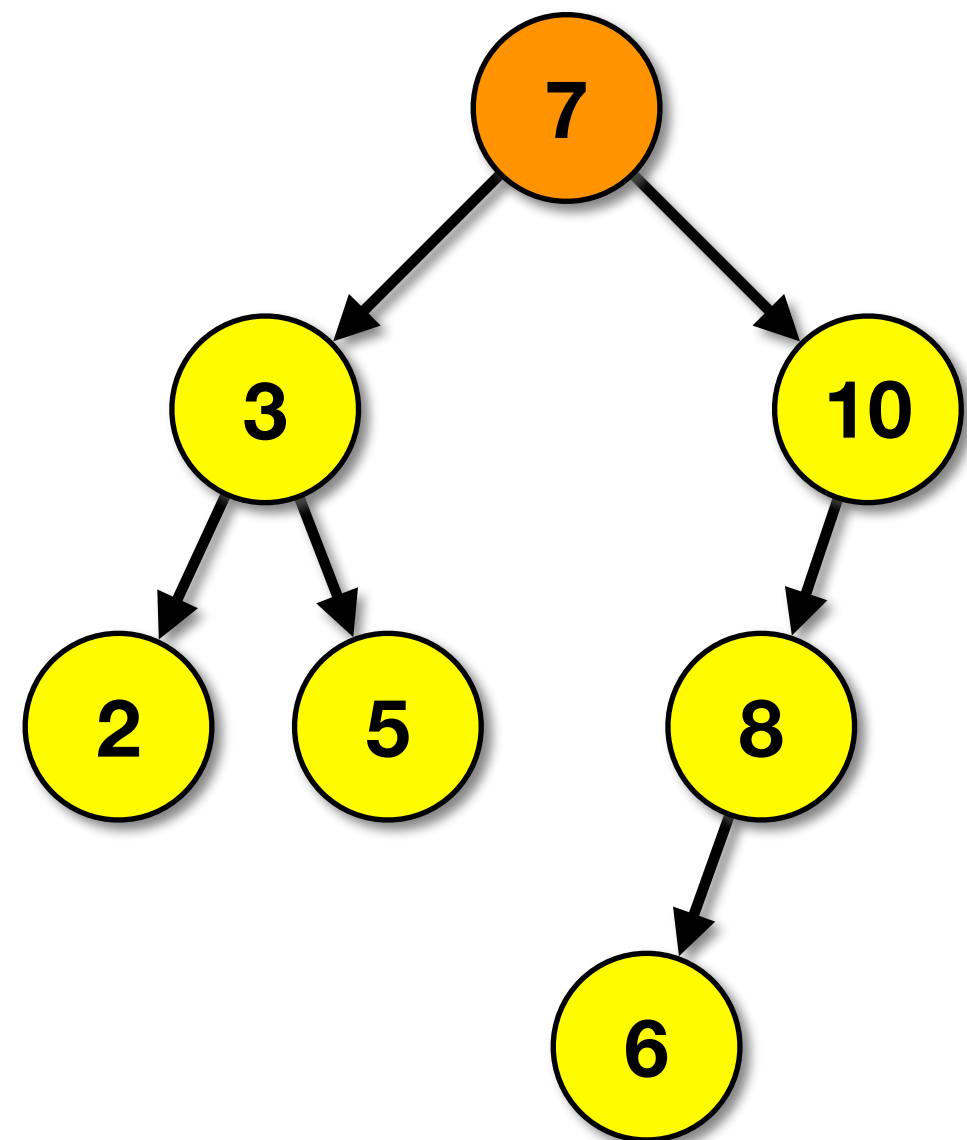
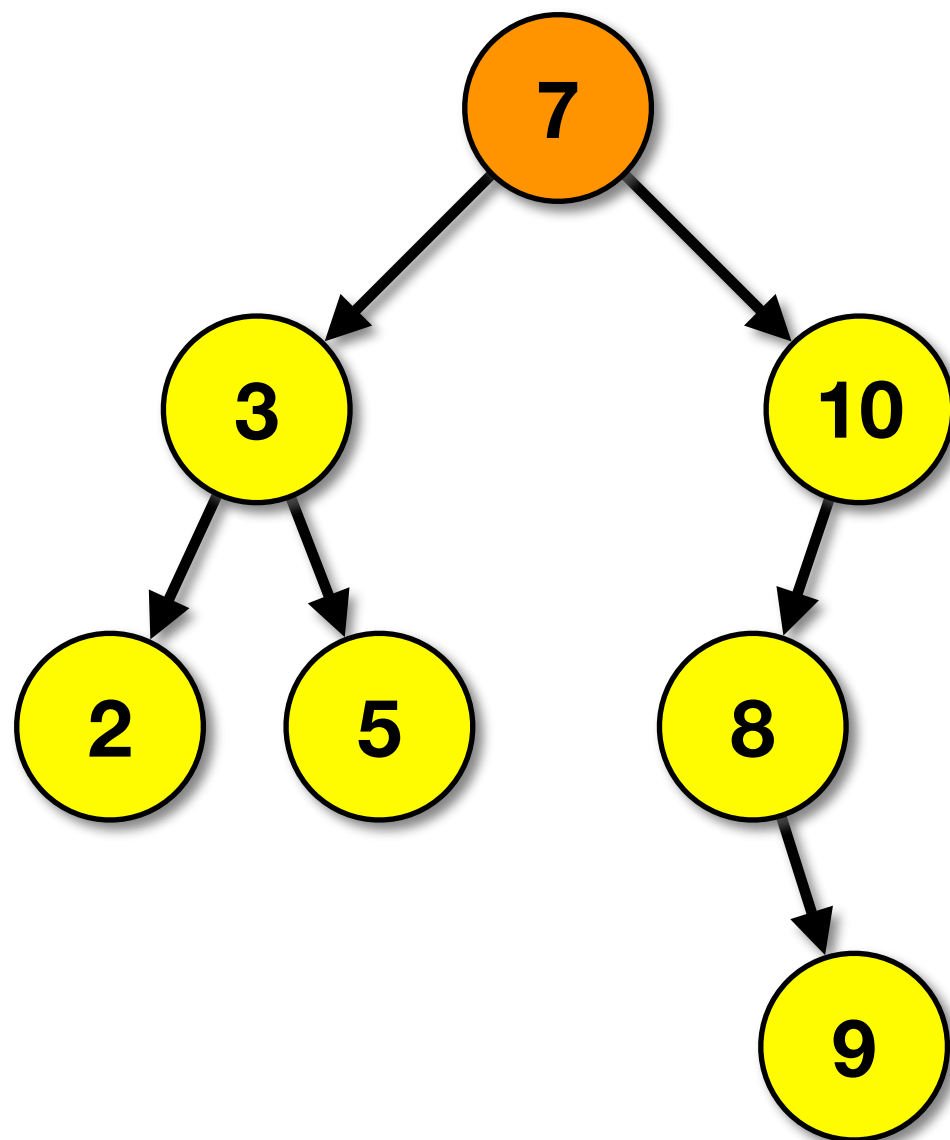
\forall nodes n in subtree u , $key(n) < key(v)$

\forall nodes n in subtree w , $key(n) > key(v)$



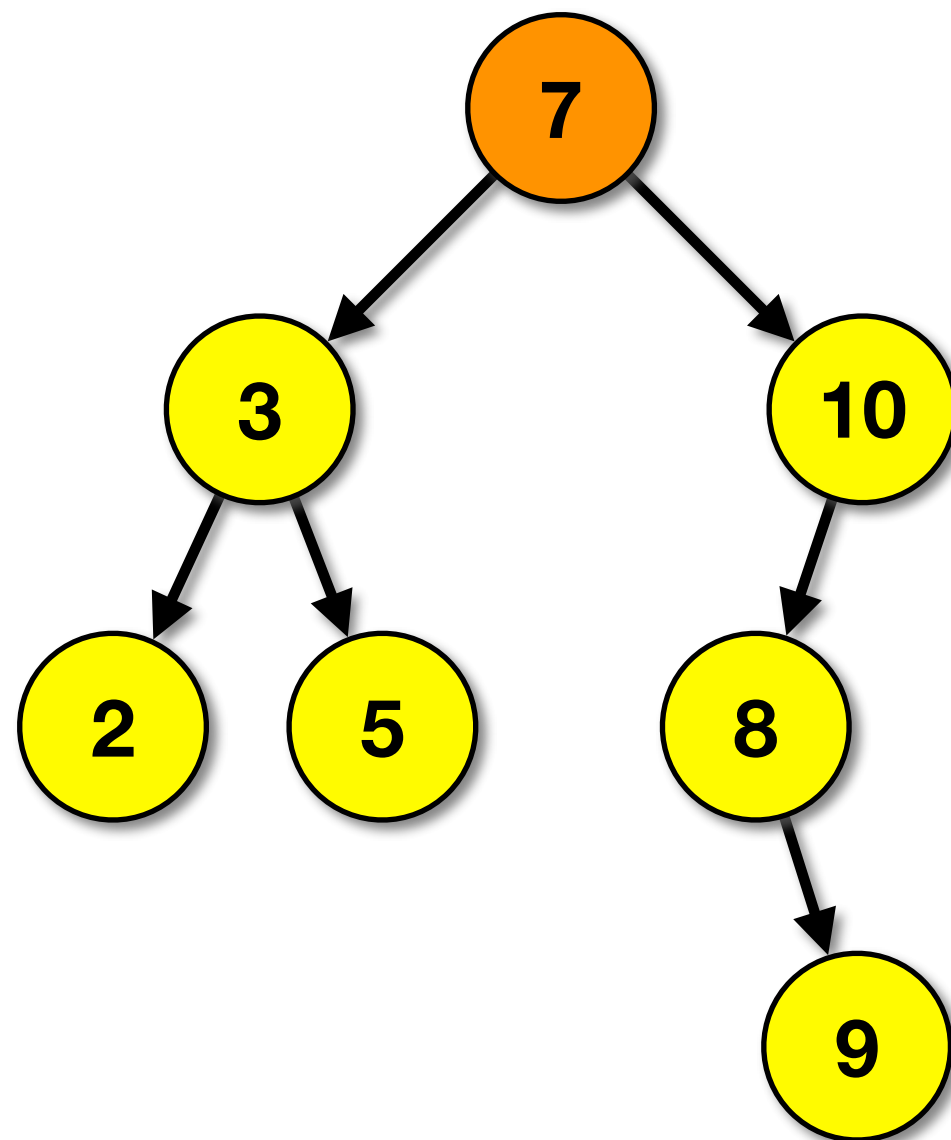
Binary Search Trees

- Which of these is not a BST? Why?

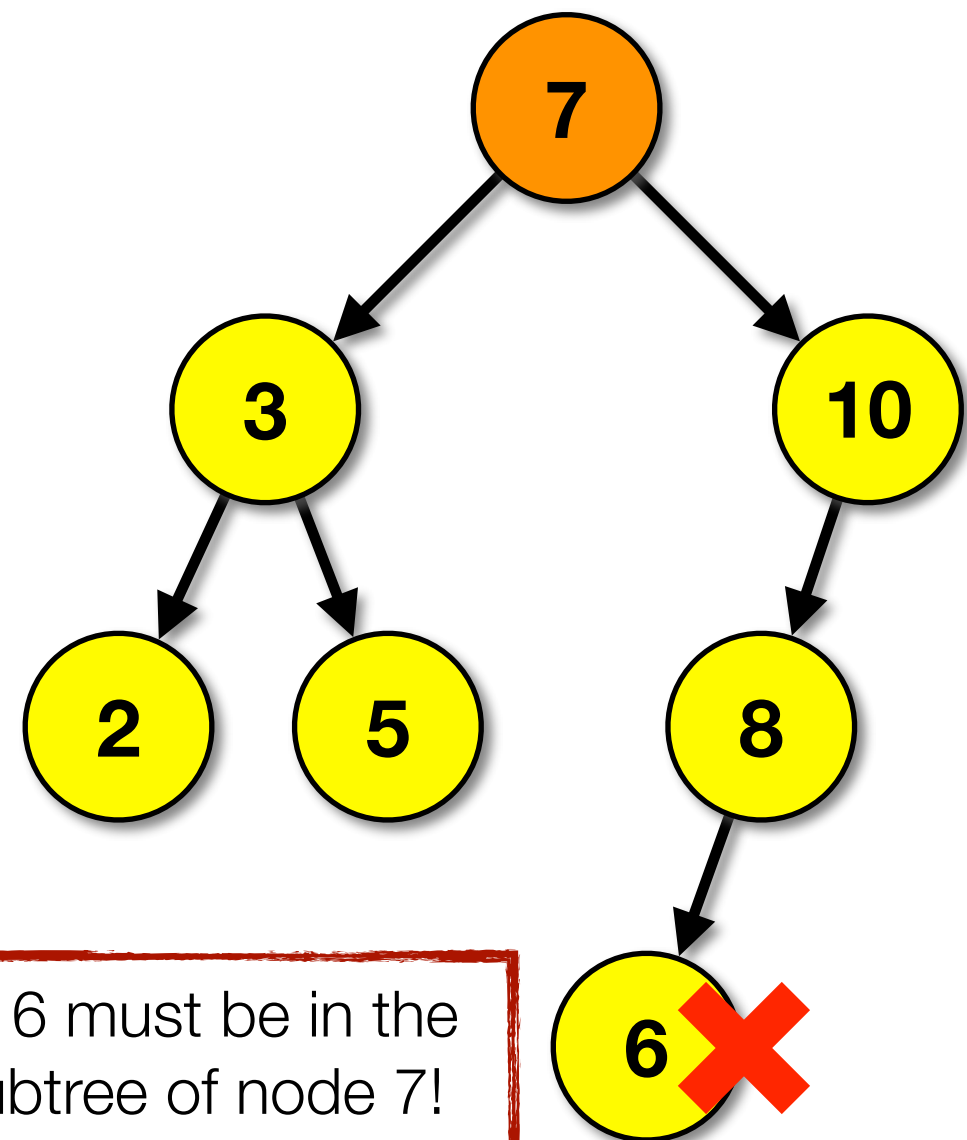


Binary Search Trees

- Which of these is not a BST? Why?



This tree is NOT a BST



Node 6 must be in the left subtree of node 7!

Typical Binary Search Tree Operations

- **isEmpty** - checks to see if the BST is empty
- **makeEmpty** - empties the BST
- **find** - searches for and returns the node with a specified key in the BST
- **findMin** - returns the node with the smallest key (or the key itself)
- **findMax** - returns the node with the largest key (or the key itself)
- **insert** - inserts a new node into the tree while maintaining the properties of a BST; all nodes are inserted as leaves
- **remove** - removes a node from the tree while maintaining the properties of a BST

Implementation of a Binary Search Tree

- **BST nodes are implemented similarly to other tree nodes**
 - Contains pointers to left and right subtrees
 - Contains a data element
 - Some implementations may contain a key-element pair where the key is used to determine where to insert the node in the tree

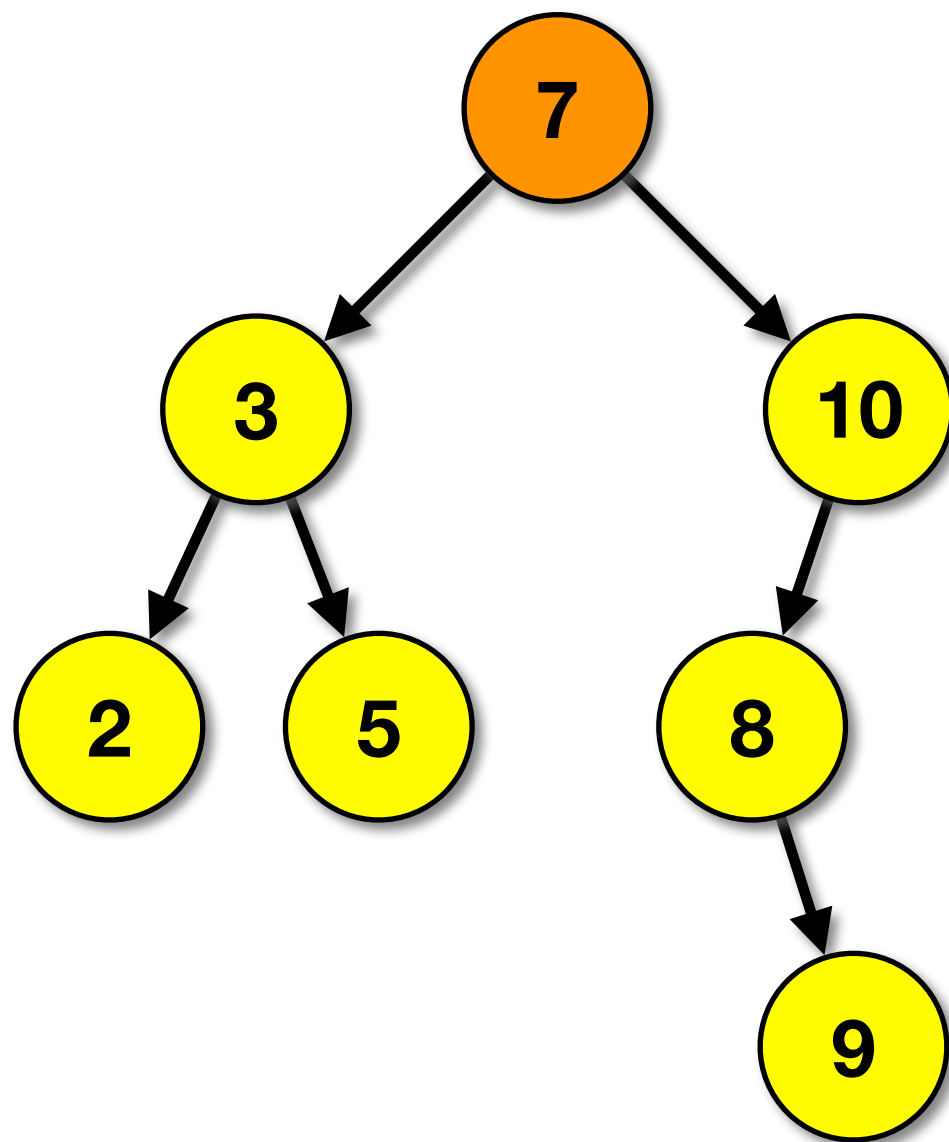
```
class BinTreeNode<E> {  
    E element;  
    BinTreeNode<E> left, right;  
  
    BinTreeNode(E newElement) {  
        element = newElement;  
    }  
}
```

The **find** Operation

- **To find a node with key k**
 - Start at the root node
 - Compare k with the key at the node
 - Move to left child if k is $<$ the key at the node
 - Move to right child if k is $>$ the key at the node
 - Repeat until either the desired key is found or until a leaf node is found
 - If a leaf node is found and the desired key has not been found, then the desired key does not exist in the tree, return `null`

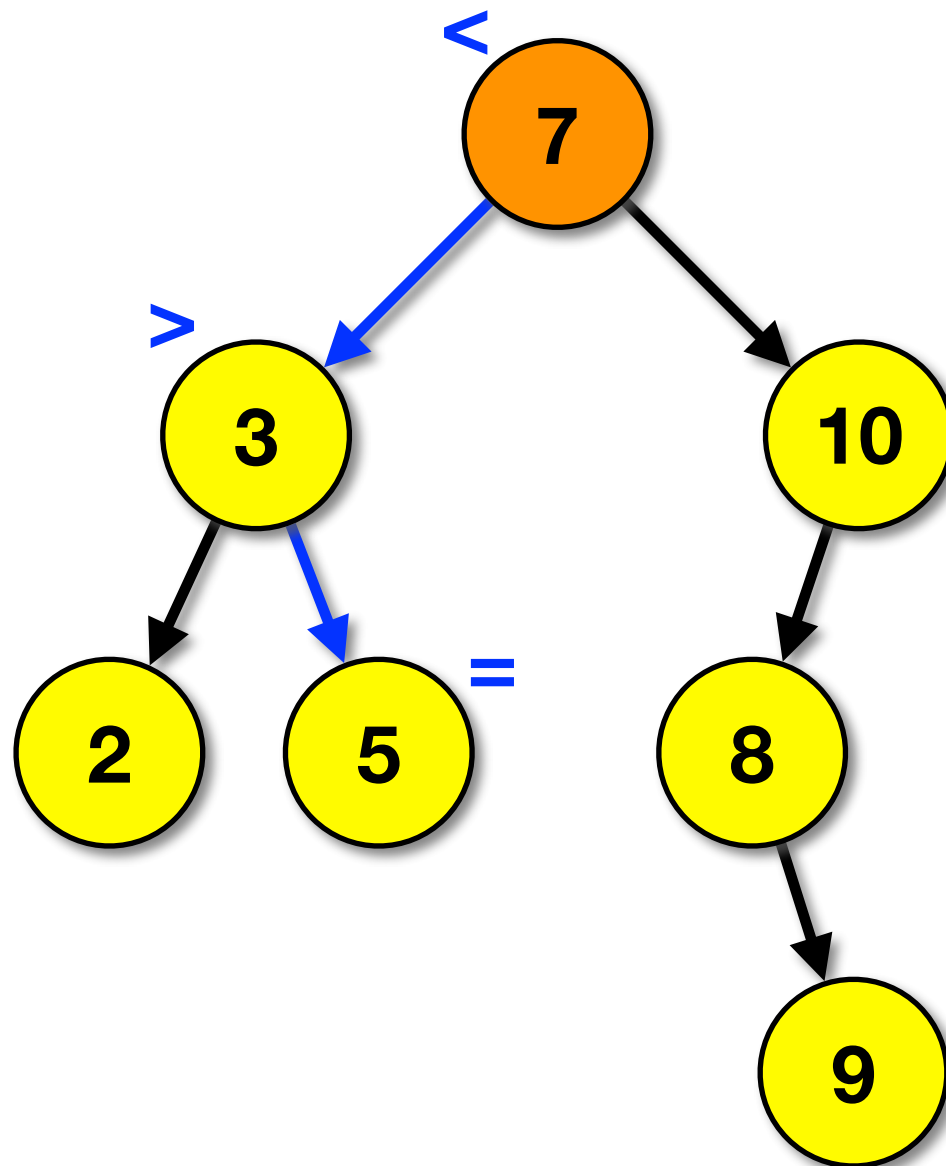
The **find** Operation

- Example of **find** operation - **find(5)**



The **find** Operation

- Example of **find** operation - **find(5)**



- (1) start at root node (node 7)
- (2) compare 5 to 7
- (3) $5 < 7$ so move left
- (4) compare 5 to 3
- (5) $5 > 3$ so move right
- (6) compare 5 to 5
- (7) found node

A Recursive Implementation of **find**

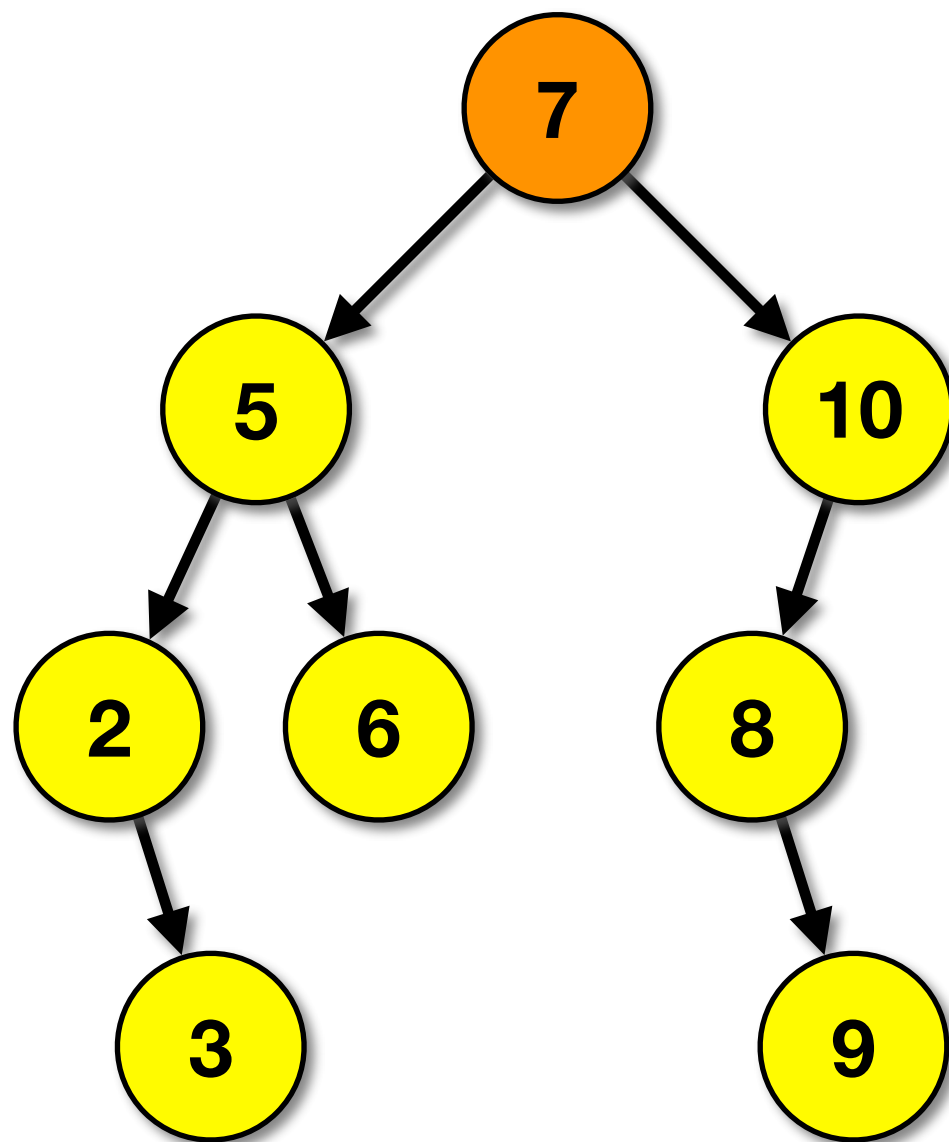
```
BinTreeNode<E> find(BinTreeNode<E> node, E element) {  
    if (node == null) {  
        return node;  
    } else if (element == node.element) {  
        return node;  
    } else if (element < node.element) {  
        return find(node.left, element);  
    } else {  
        return find(node.right, element);  
    }  
}
```

The **findMin** Operation

- **How can the node with the smallest key be found?**
- **To find the node in the tree with the smallest key k**
 - Start at the root node
 - Repeatedly move to the left child until there are no more left children
 - Return the node

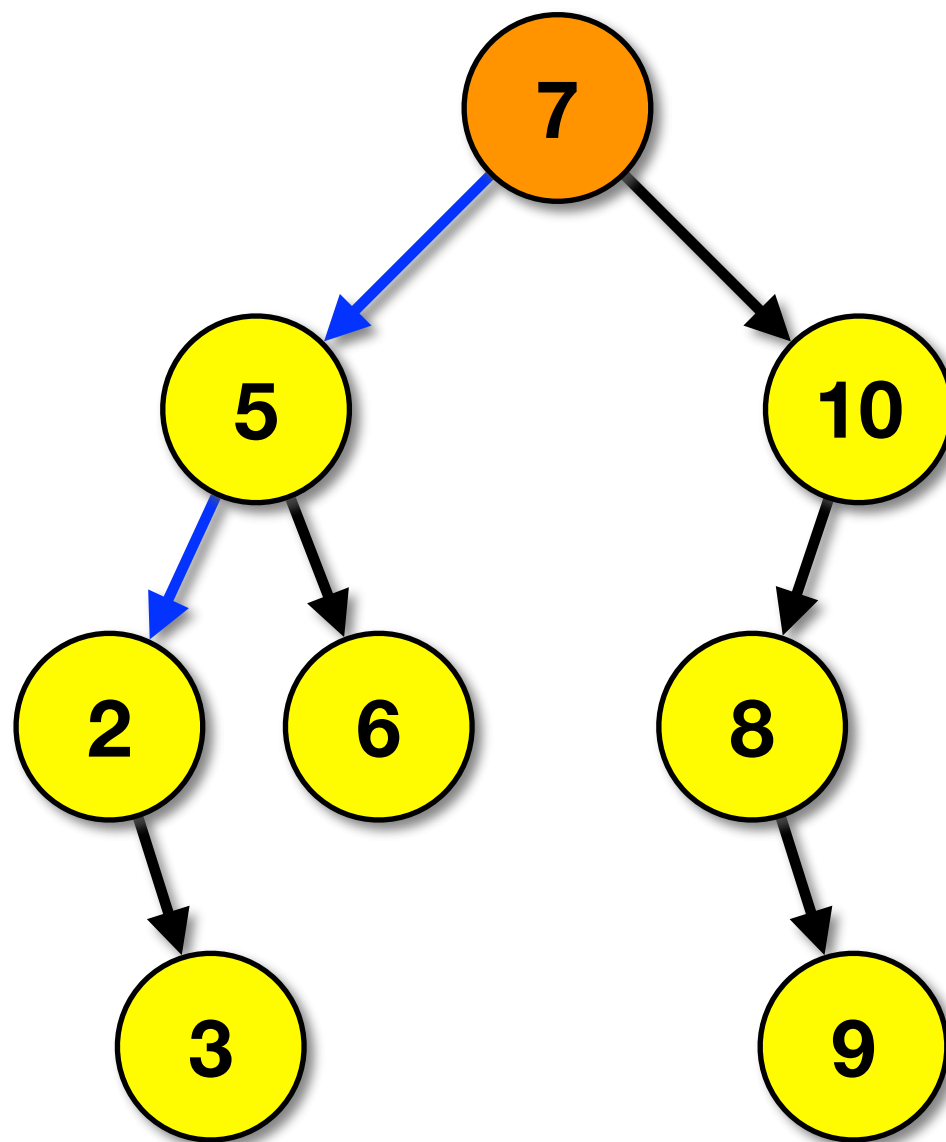
The **findMin** Operation

- Example of **findMin** operation - **findMin()**



The **findMin** Operation

- **Example of findMin operation - findMin()**



- (1) start at root node (node 7)
- (2) move to left child (node 5)
- (3) move to left child (node 2)
- (4) no more left children, so return node 2 as minimum key in tree

A Recursive Implementation of **findMin**

```
BinTreeNode<E> findMin(BinTreeNode<E> node) {  
    if (node.left == null) {  
        return node;  
    } else {  
        return findMin(node.left);  
    }  
}
```

The **findMax** Operation

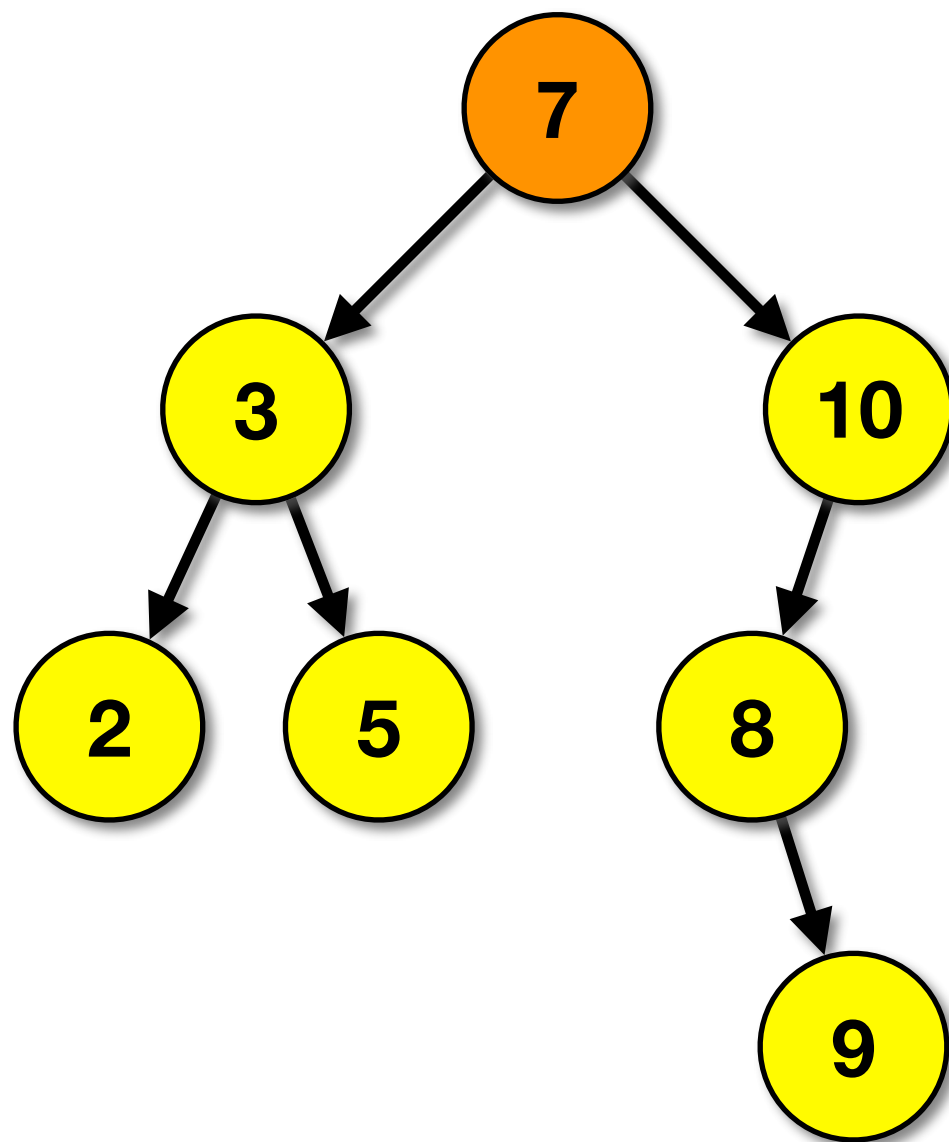
- How can the node with the largest key be found?
- To find the node in the tree with the largest key k
 - Start at the root node
 - Repeatedly move to the right child until there are no more right children
 - Return the node

The **insert** Operation

- **To insert a node with key k**
 - Start at the root node
 - Compare k with the key at the node
 - If the node is null, insert new node at current location in BST
 - Move to left child if k is $<$ the key at the node
 - Move to right child if k is $>$ the key at the node
 - Repeat until a null location is found in which to insert the new node
 - All insertions create a new leaf node

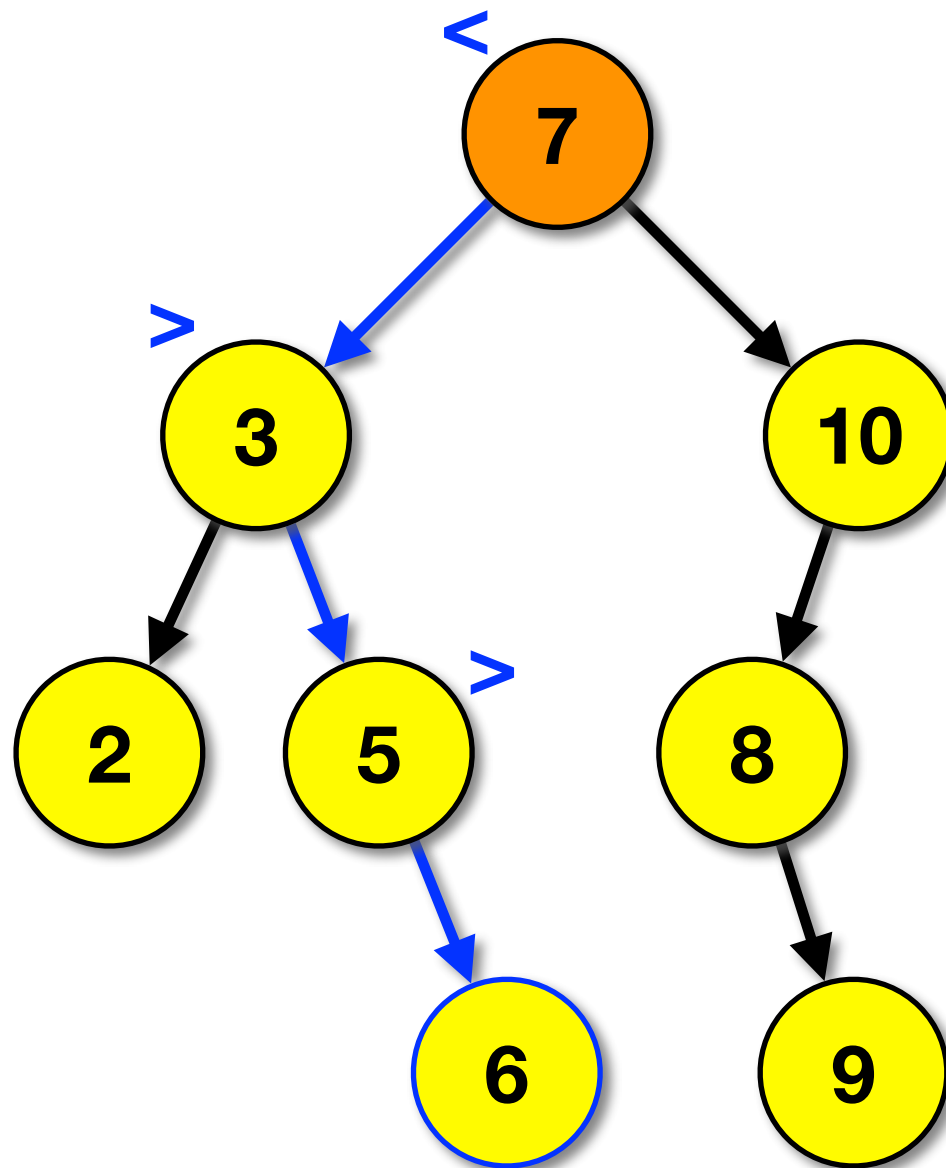
The **insert** Operation

- Example of **insert** operation - **insert(6)**



The **insert** Operation

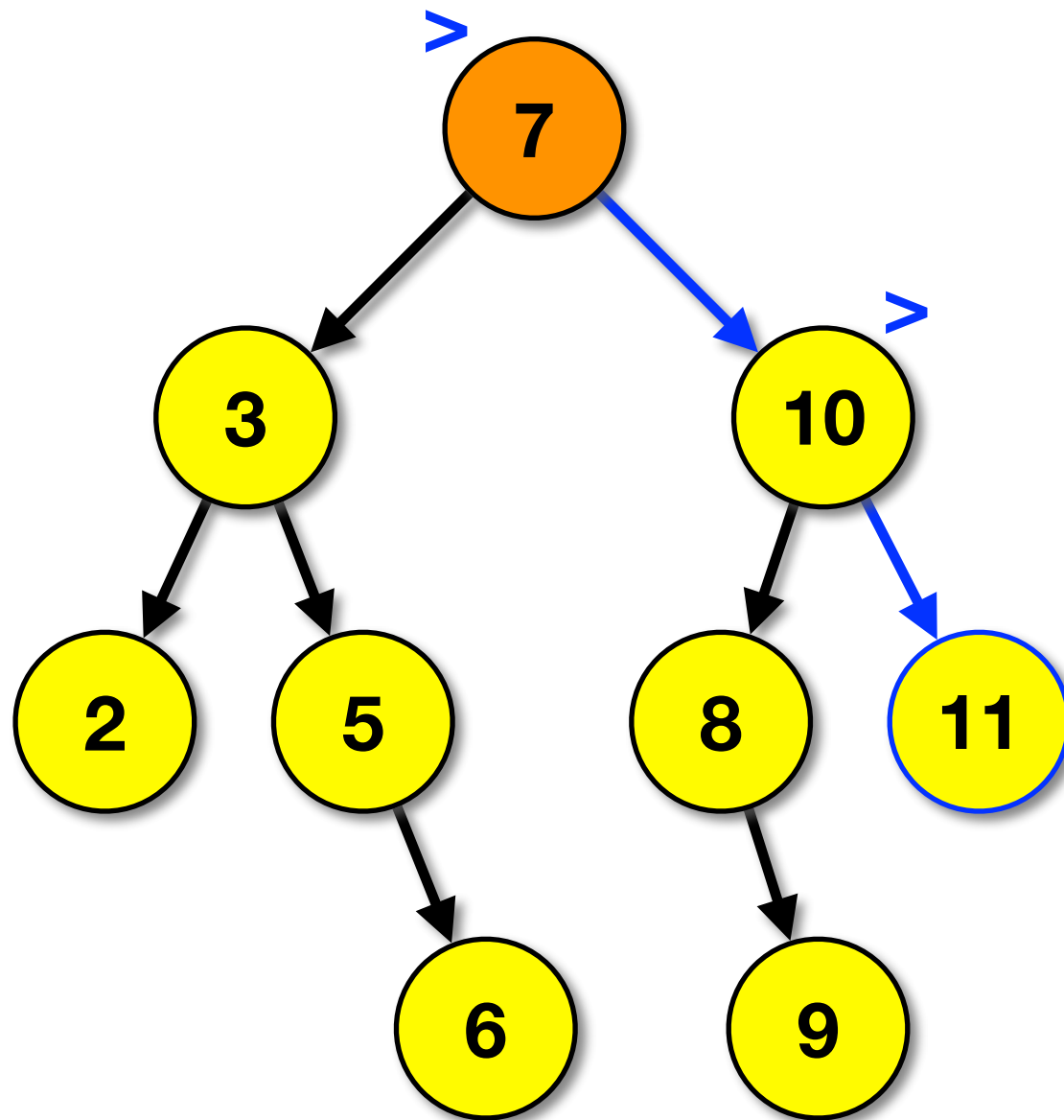
- Example of **insert** operation - **insert(6)**



- (1) start at root node (node 7)
- (2) compare 6 to 7
- (3) $6 < 7$ so move left
- (4) compare 6 to 3
- (5) $6 > 3$ so move right
- (6) compare 6 to 5
- (7) $6 > 5$ so move right
- (8) found null location, so add new node 6 there

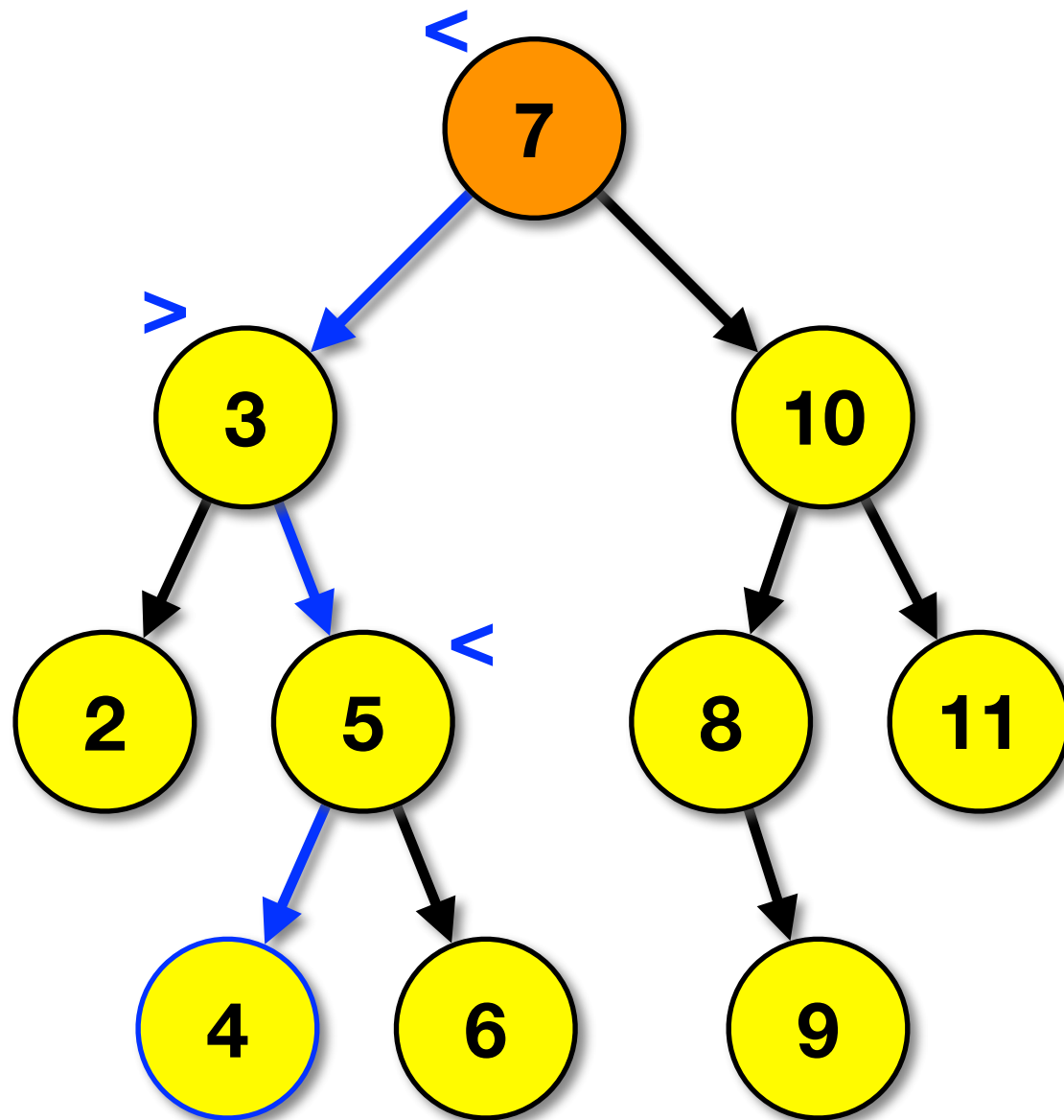
The **insert** Operation

- Example of **insert** operation - **insert(11)**



The **insert** Operation

- Example of **insert** operation - **insert(4)**



A Recursive Implementation of **insert**

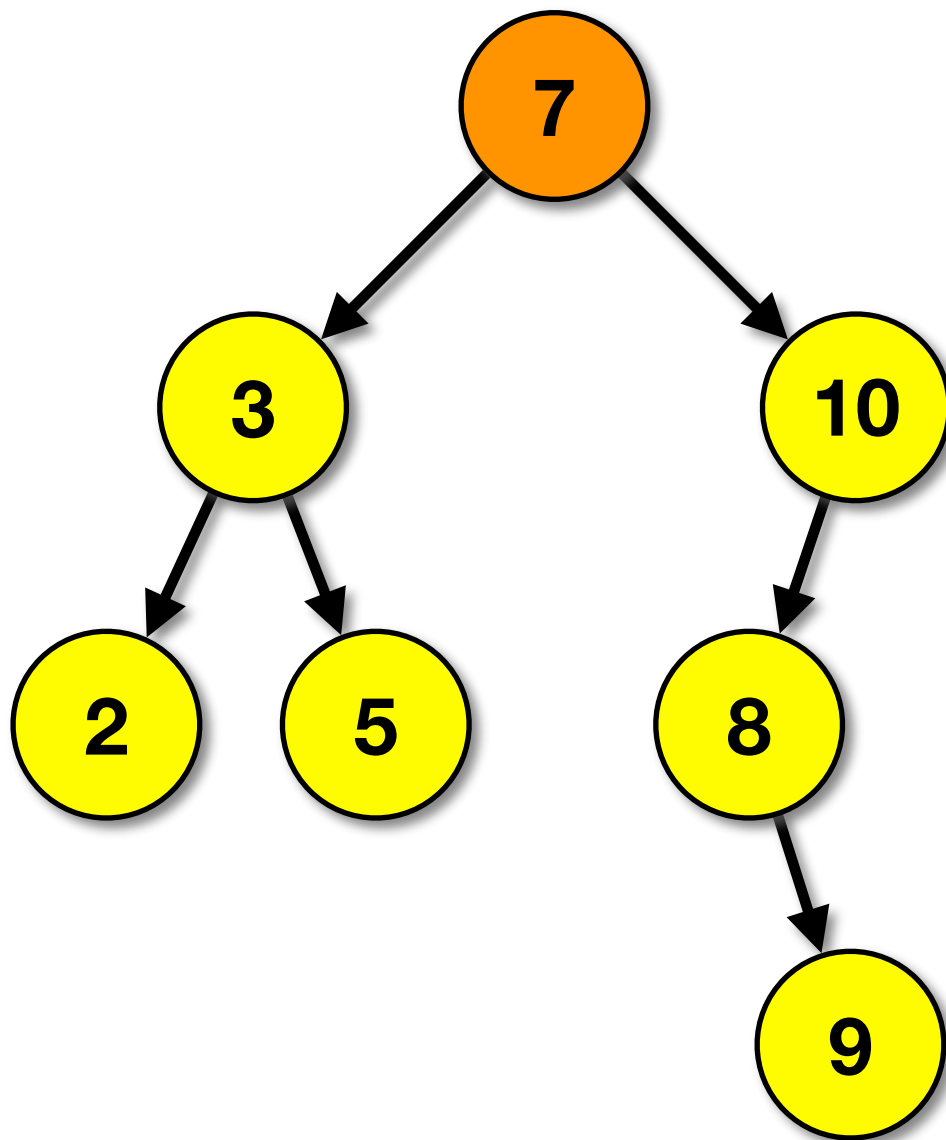
```
BinTreeNode<E> insert(BinTreeNode<E> node, E element) {  
    if (node == null) {  
        node = new BinTreeNode(element);  
    } else if (element < node.element) {  
        node.left = insert(node.left, element);  
    } else {  
        node.right = insert(node.right, element);  
    }  
    return node;  
}
```

The **remove** Operation

- **Find a node N with key k and remove it from the tree**
 - Start at the root node
 - Find the node requested for removal
 - Remove the node (there are several different cases that need to be considered when removing a node):
 - 1) Node N is not found -- do nothing
 - 2) Node N is a leaf node -- simply remove node N from tree
 - 3) Node N has only a single child -- remove node N and replace it with its child
 - 4) Node N has two children -- do not delete node N, instead find its in-order successor node (or in-order predecessor) (node R) and replace the values in the node N with those from node R. Then delete node R.

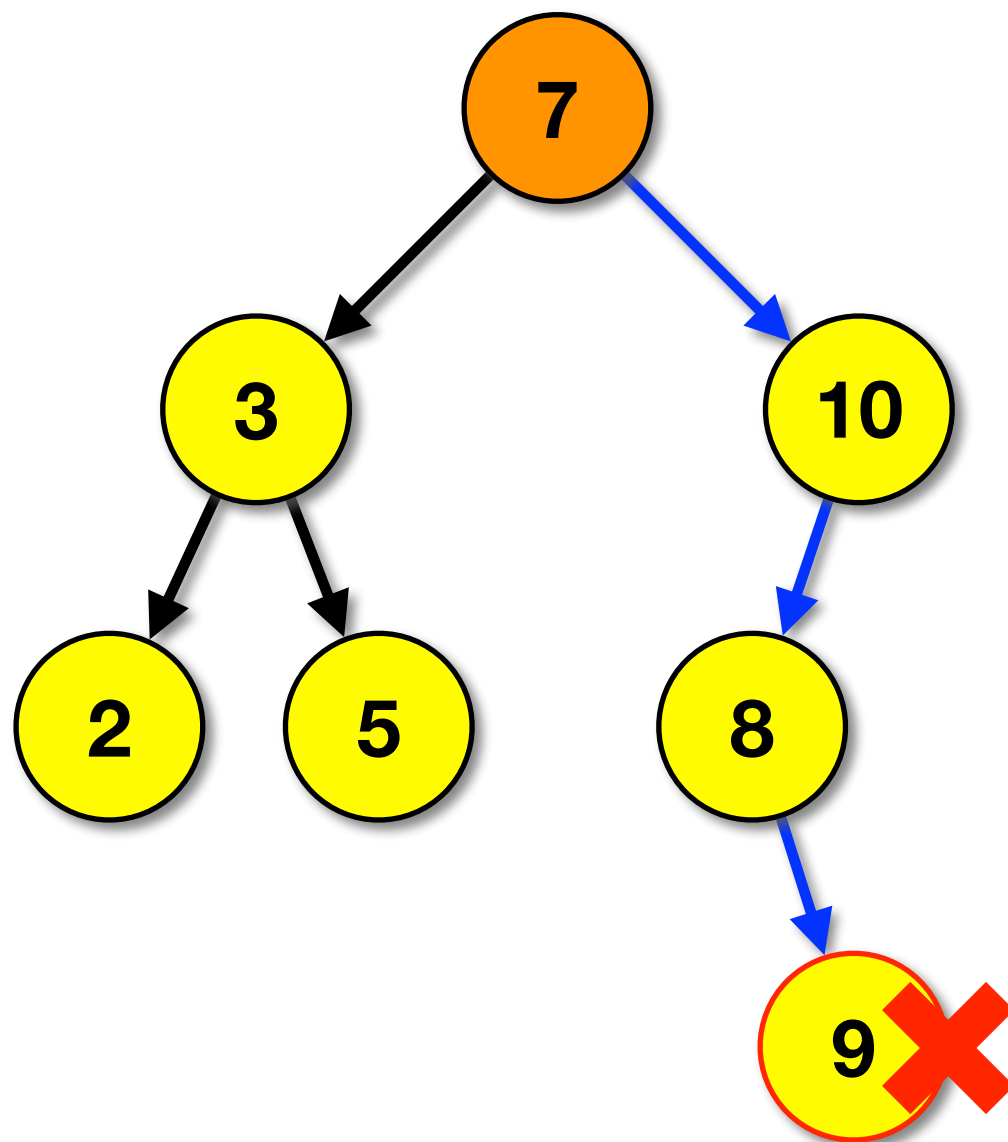
The **remove** Operation: Case #2 - Leaf Node

- Example of **remove** operation on leaf node - **remove(9)**



The **remove** Operation: Case #2 - Leaf Node

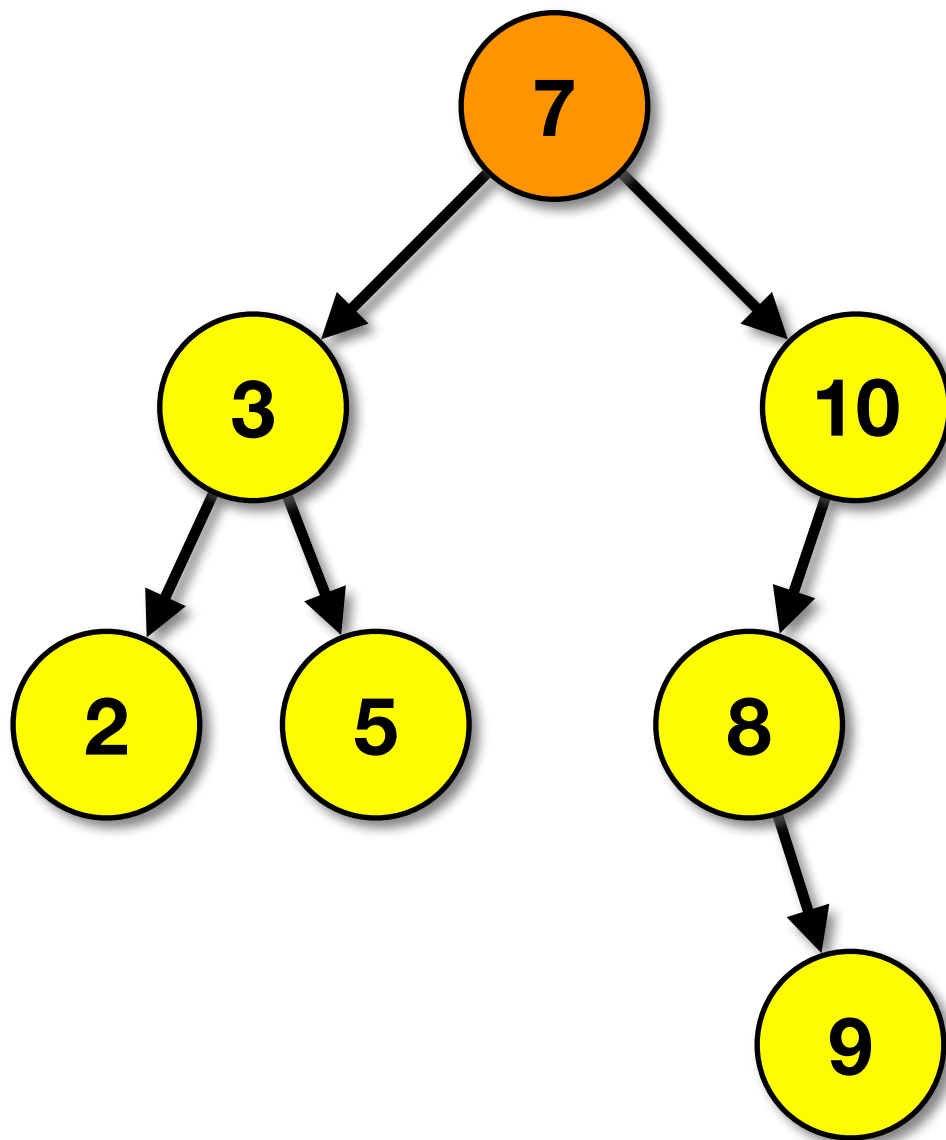
- Example of **remove** operation on leaf node - **remove(9)**



- (1) start at root node (node 7)
- (2) find node 9
- (3) node 9 is a leaf node
- (4) simply delete node 9 from the tree (i.e. set node 8's right child to null)

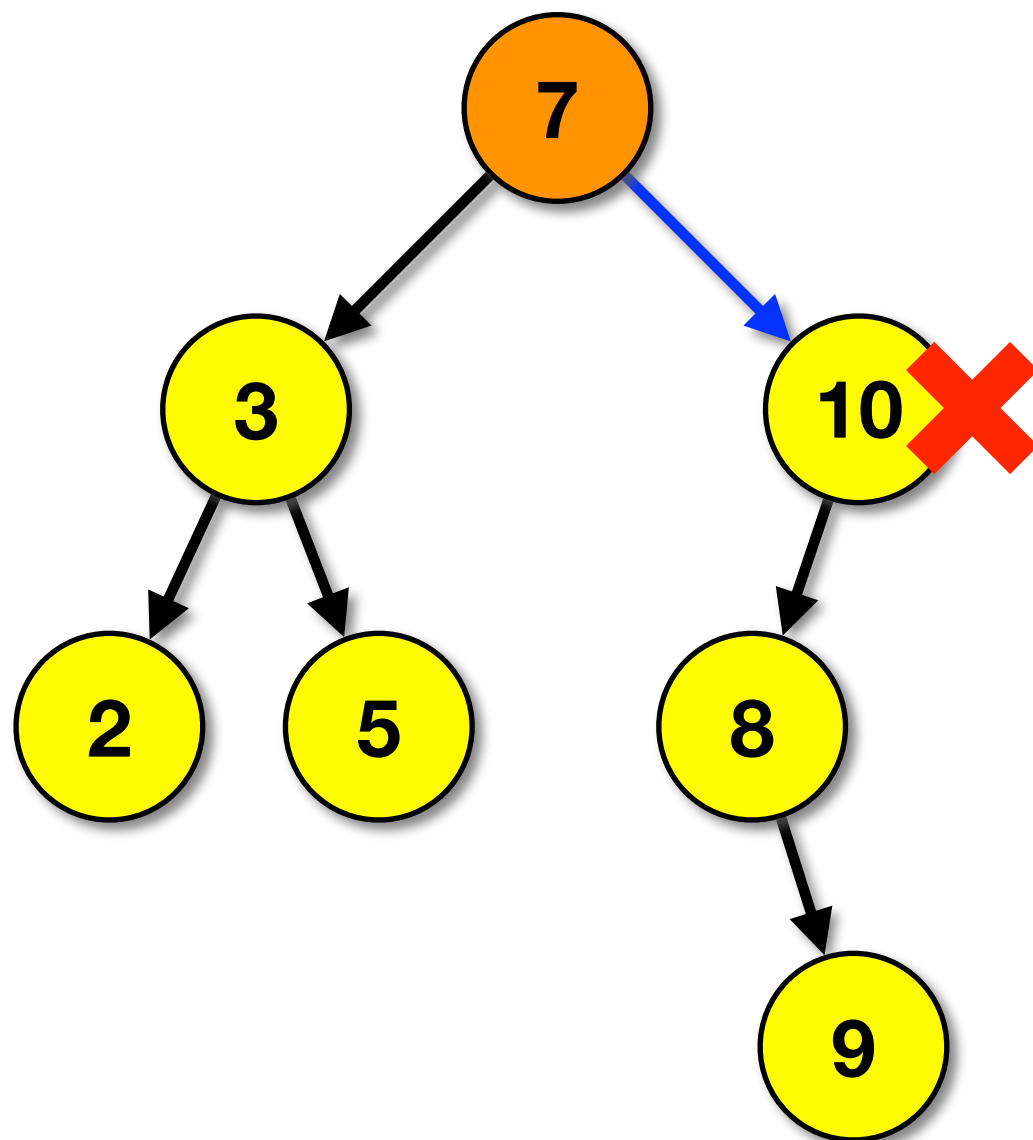
The **remove** Operation: Case #3 - Single Child

- Example of **remove** operation with a single child - **remove(10)**



The **remove** Operation: Case #3 - Single Child

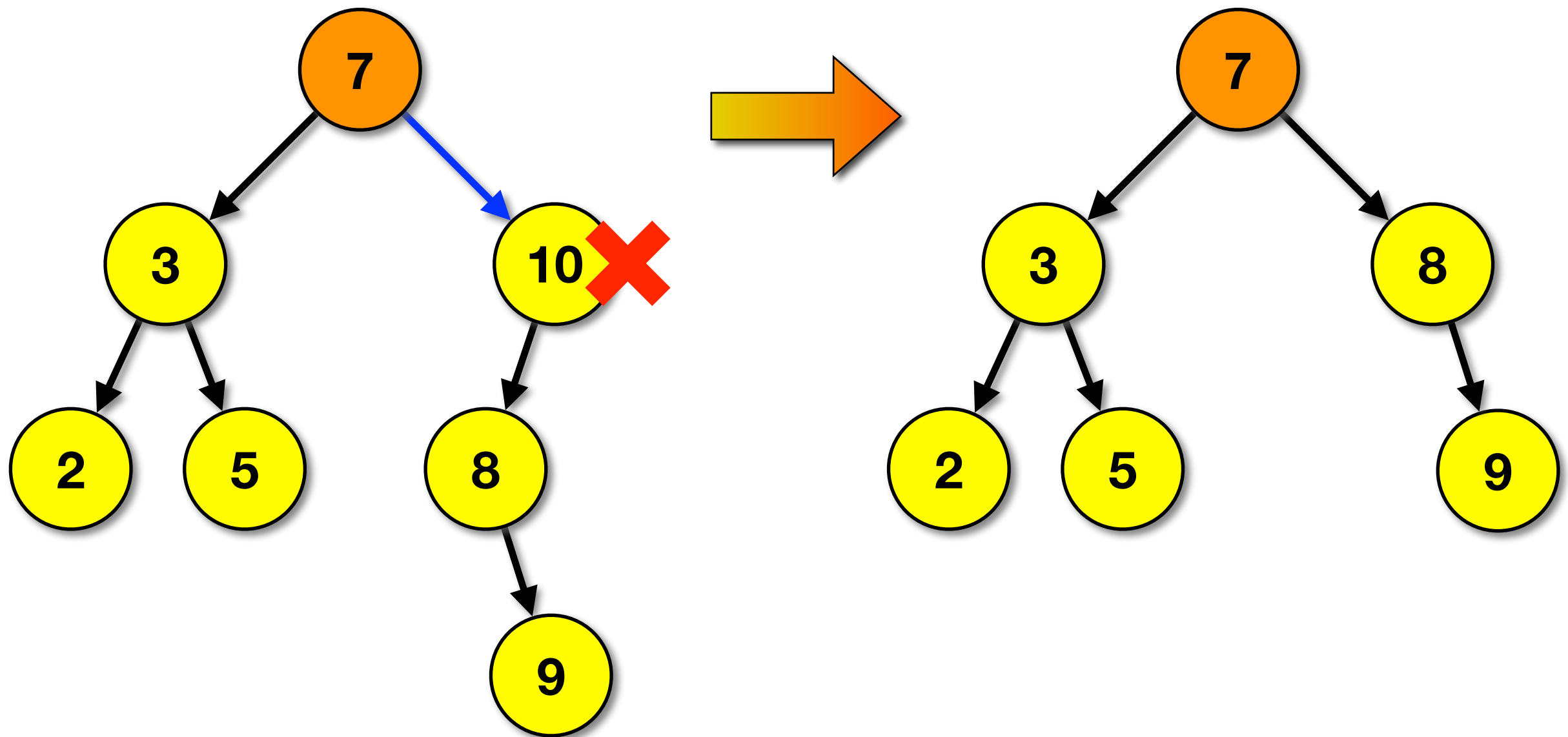
- Example of **remove** operation with a single child - **remove(10)**



- (1) start at root node (node 7)
- (2) find node 10
- (3) node 10 has a single child
- (4) remove node 10, and replace it with its only child

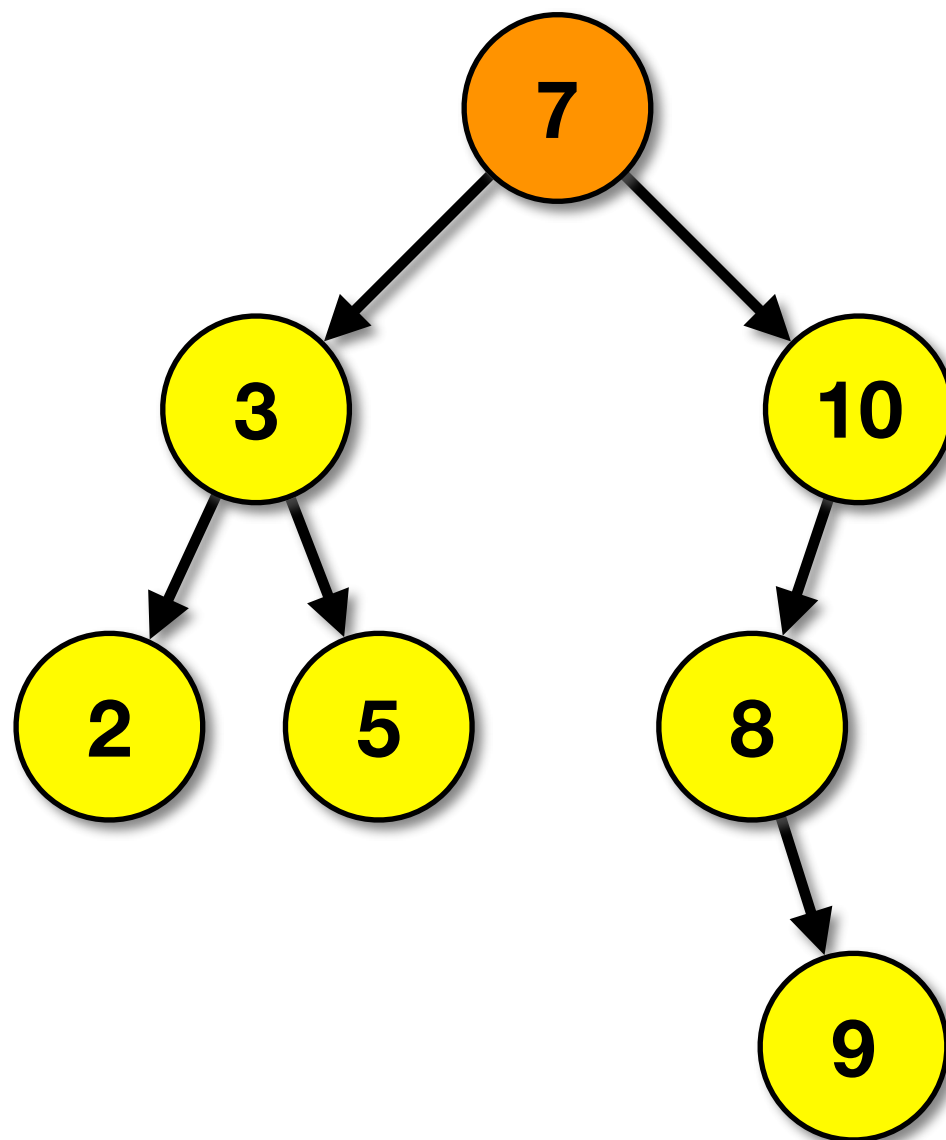
The **remove** Operation: Case #3 - Single Child

- Example of **remove** operation with a single child - **remove(10)**



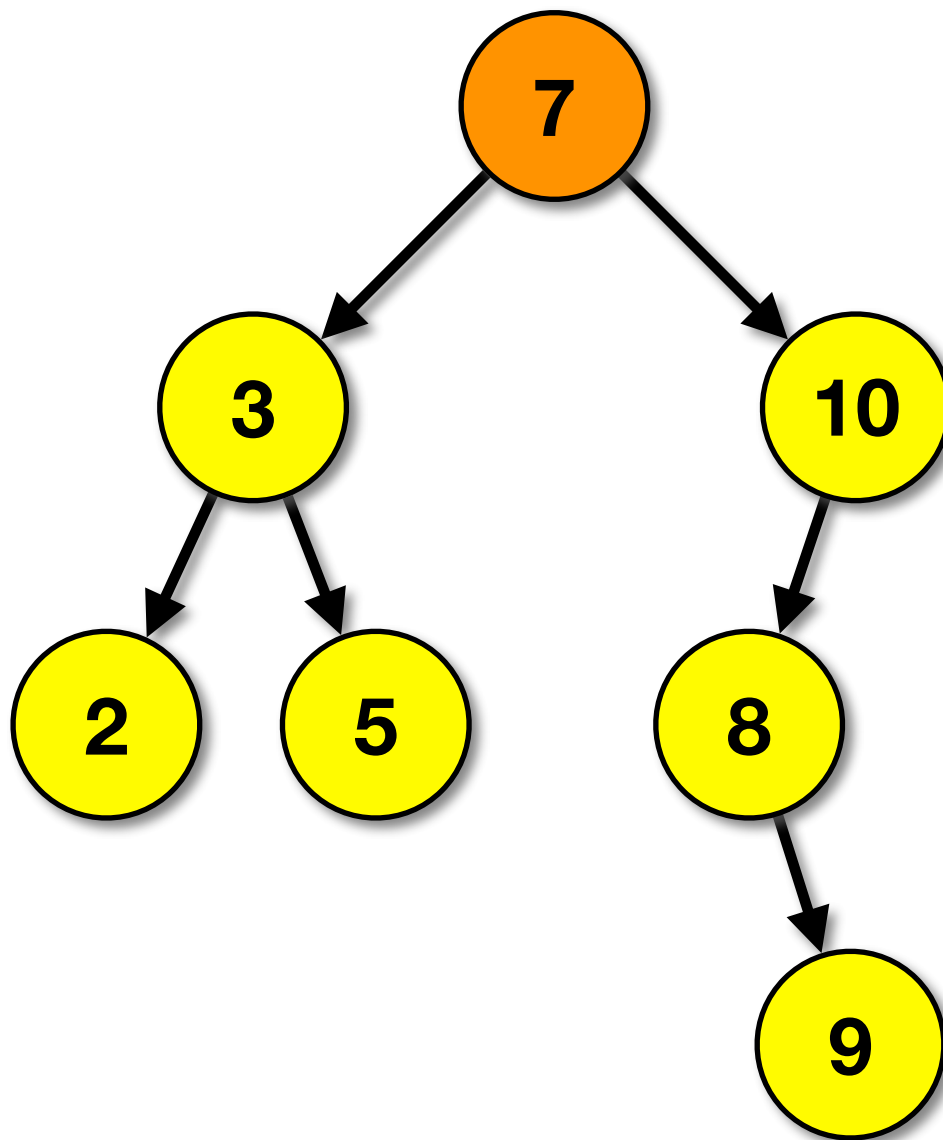
The **remove** Operation: Case #4 - Two Children

- Example of **remove** operation with a two children - **remove(3)**



The **remove** Operation: Case #4 - Two Children

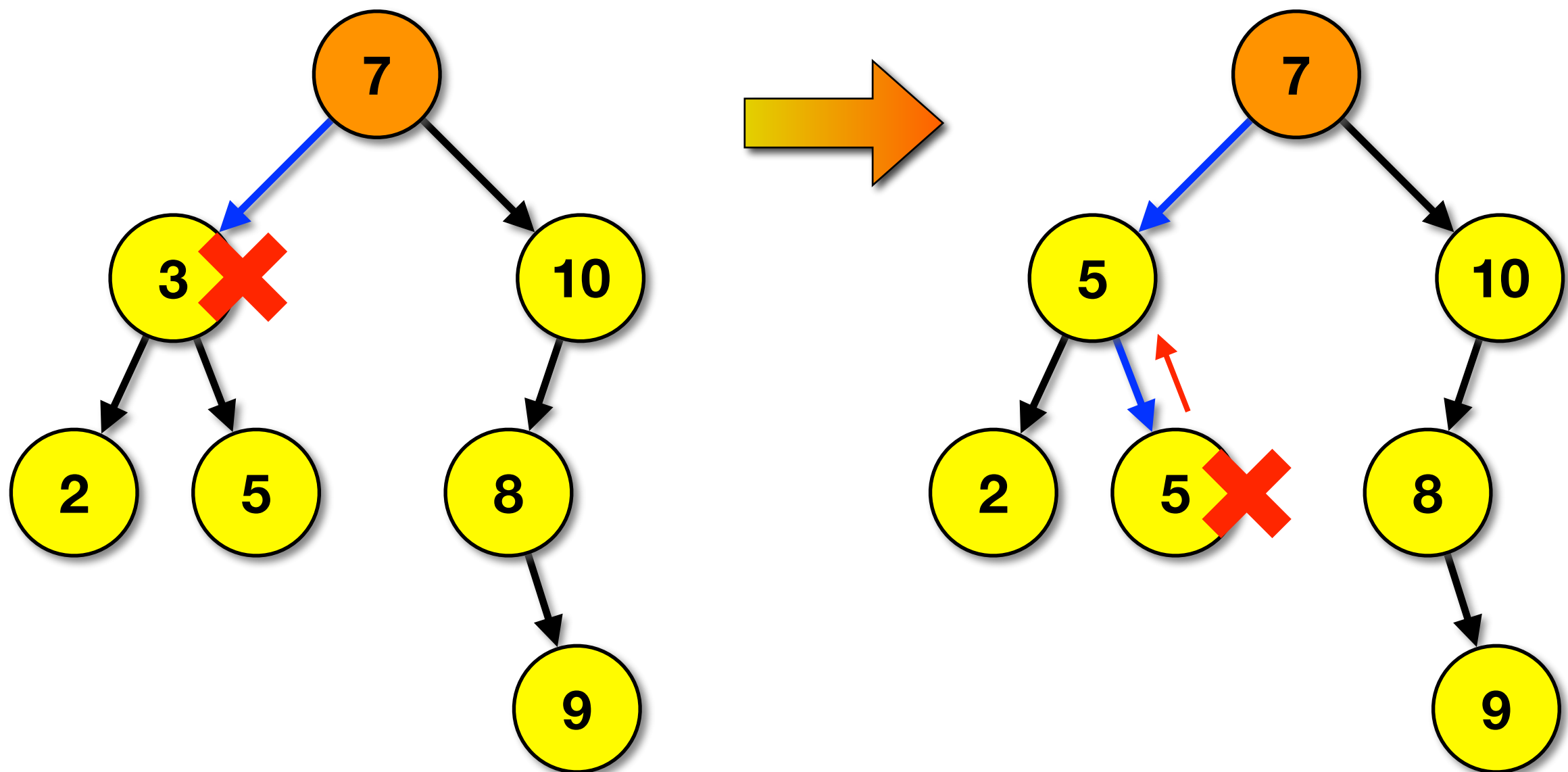
- Example of **remove** operation with a two children - **remove(3)**



- (1) start at root node (node 7)
- (2) find node 3
- (3) node 3 has two children
- (4) find the successor of node 3 (i.e. the min value from its right subtree)
- (5) replace values of node 3 with those from successor
- (6) remove the successor (may require additional modifications to the tree)

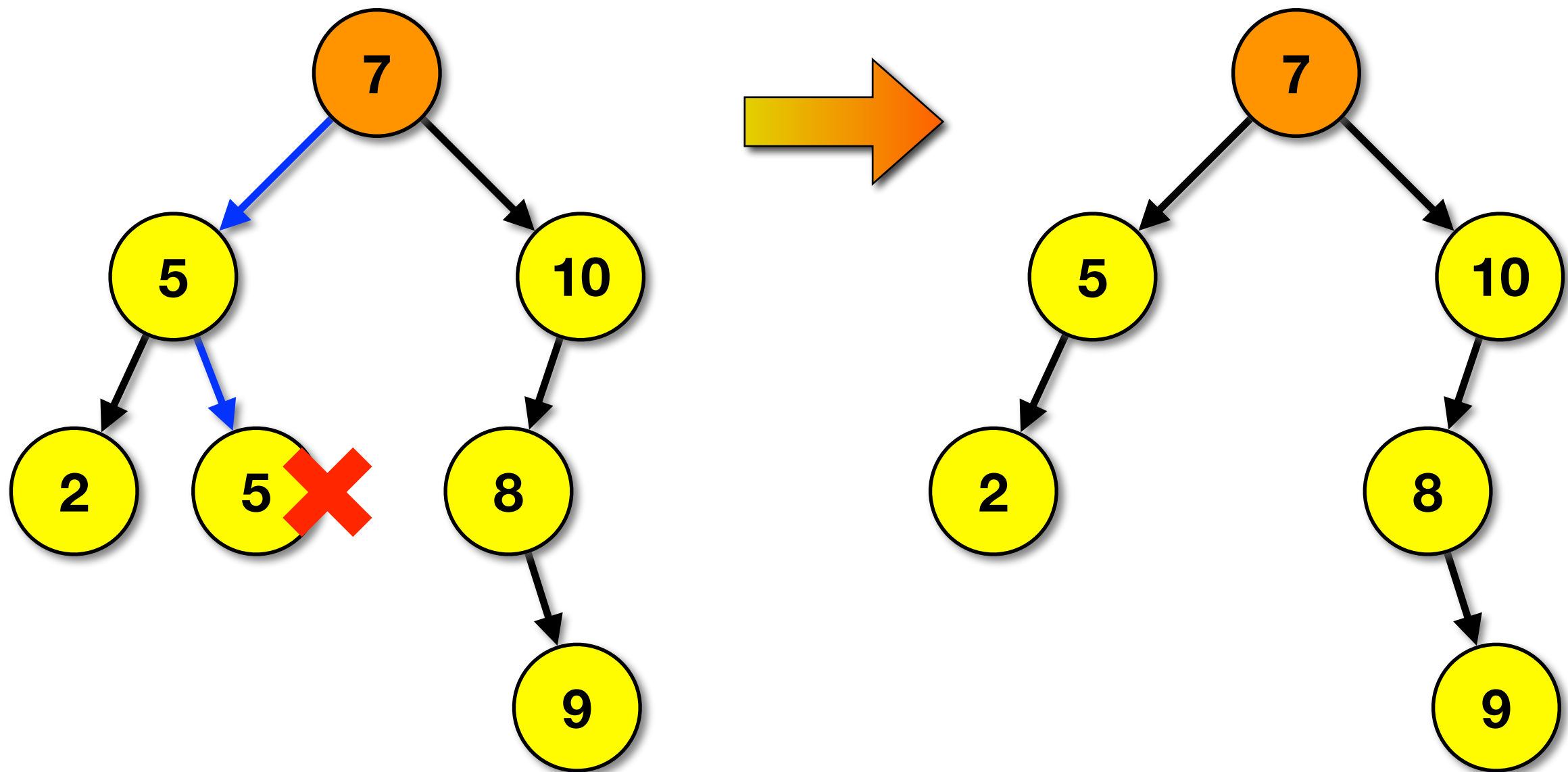
The **remove** Operation: Case #4 - Two Children

- Example of **remove** operation with a two children - **remove(3)**



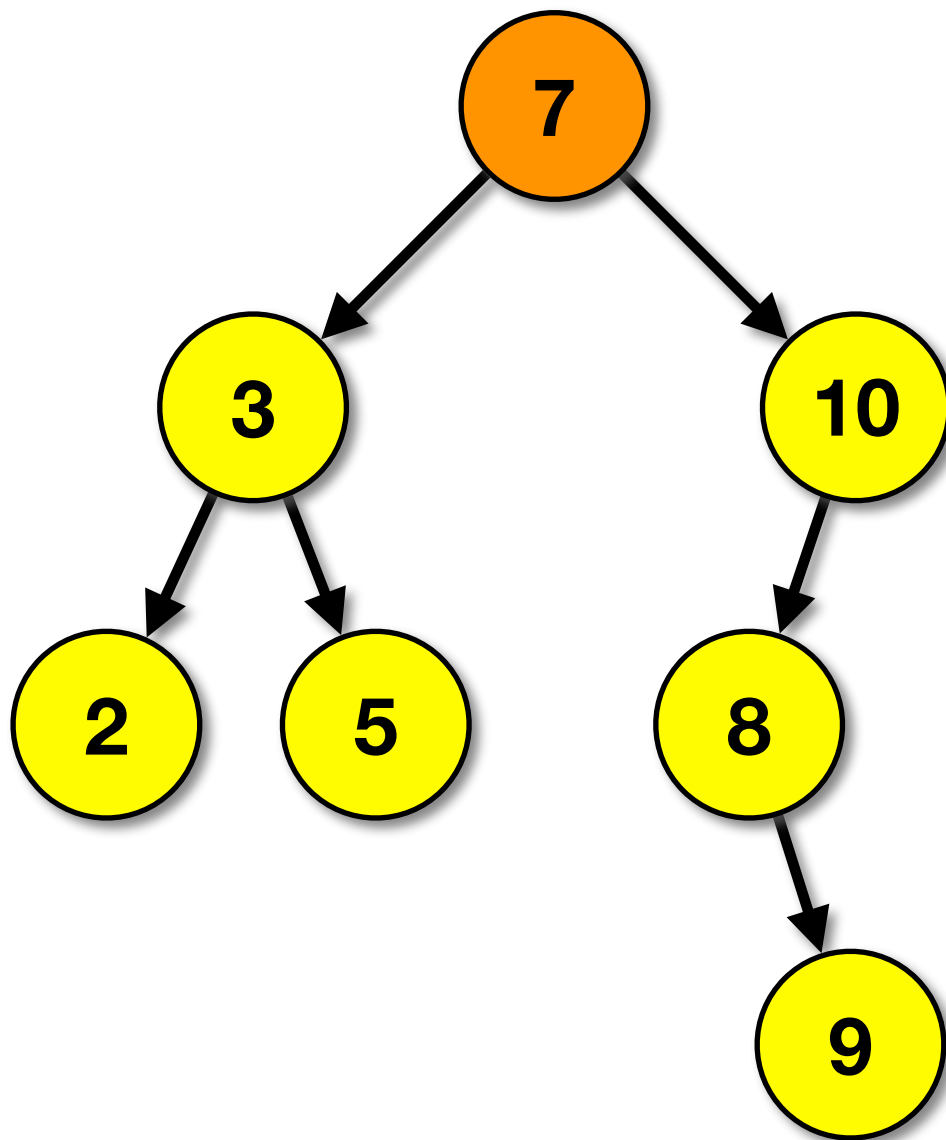
The **remove** Operation: Case #4 - Two Children

- Example of **remove** operation with a two children - **remove(3)**



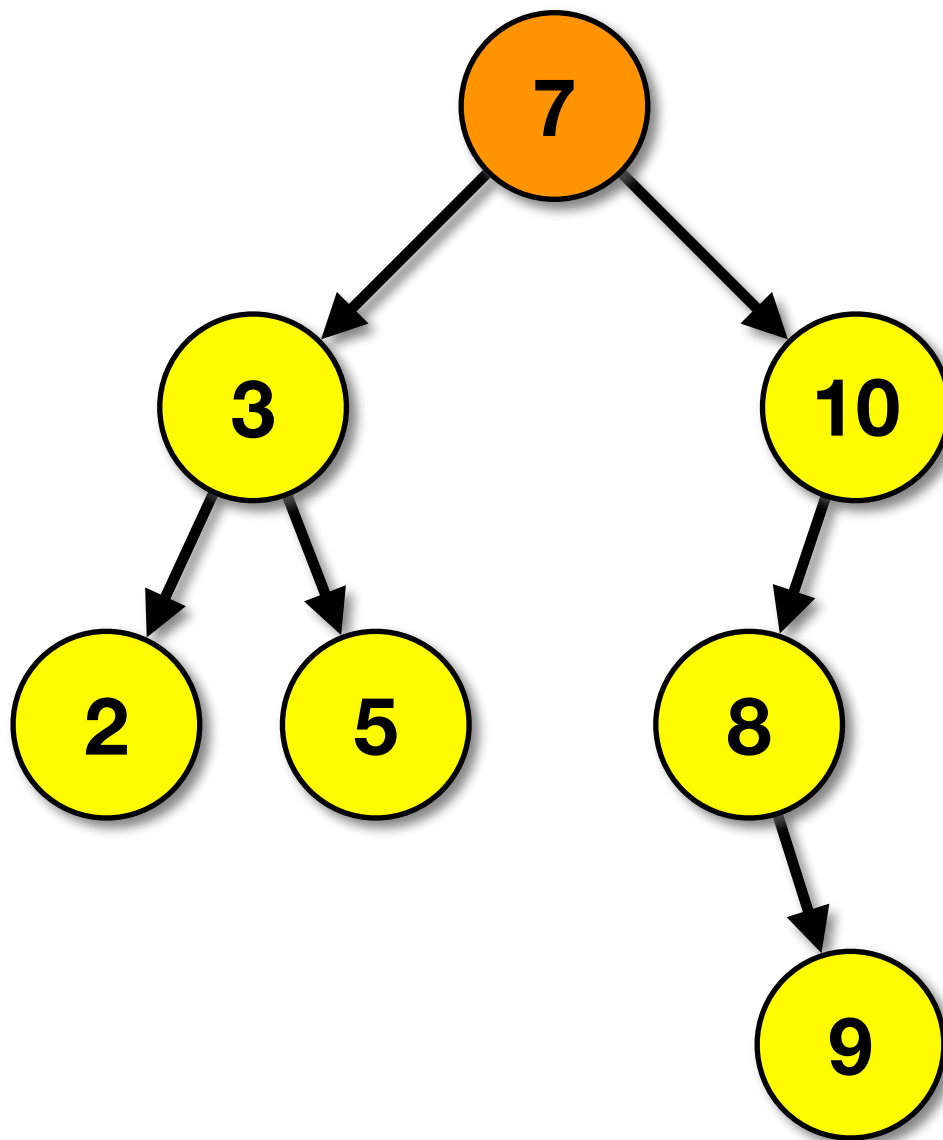
The **remove** Operation: Case #4 - Two Children

- Example of **remove** operation with a two children - **remove(7)**



The **remove** Operation: Case #4 - Two Children

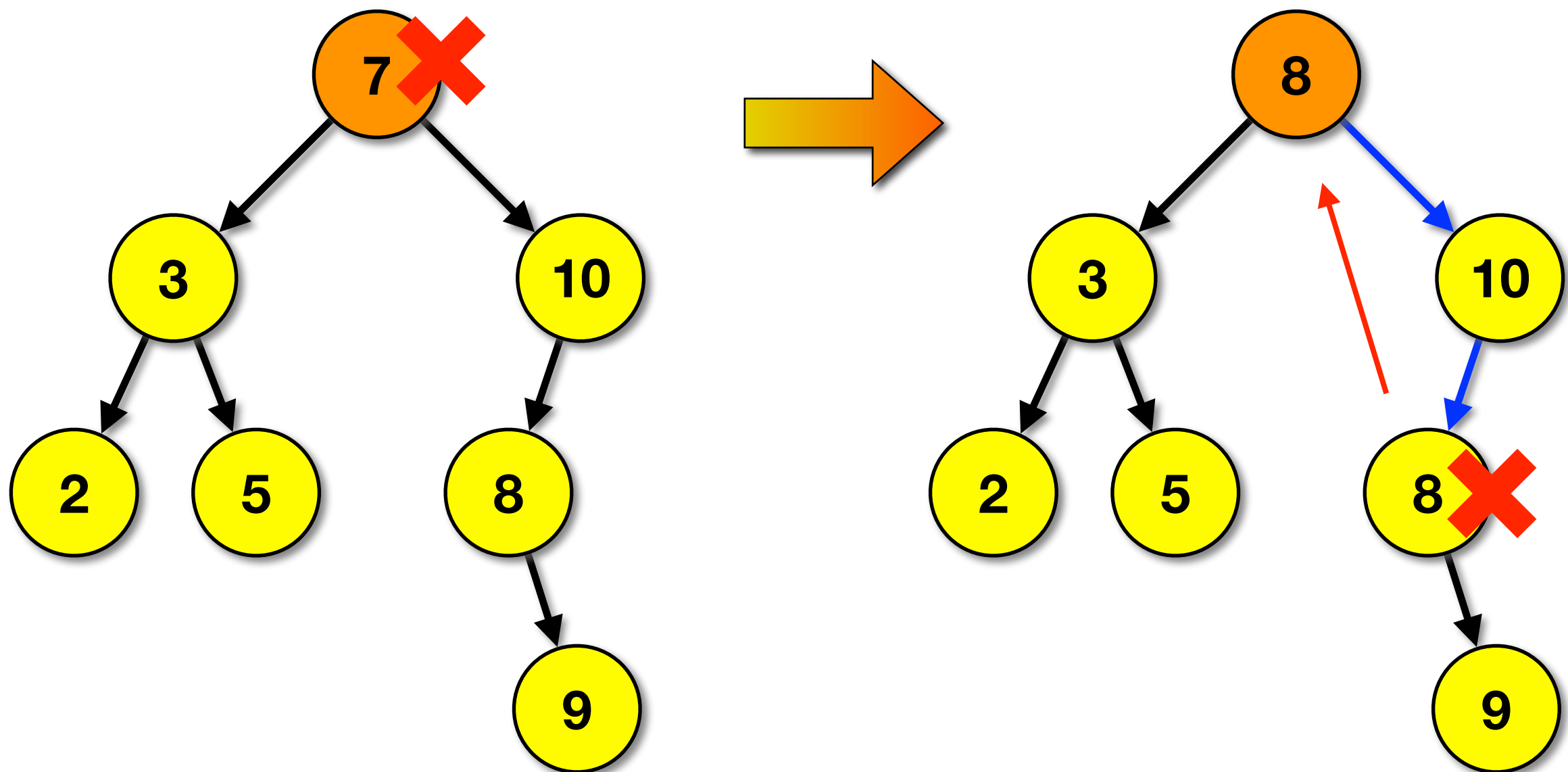
- Example of **remove** operation with a two children - **remove(7)**



- (1) start at root node (node 7)
- (2) find node 7
- (3) node 7 has two children
- (4) find the successor of node 7 (i.e. the min value from its right subtree)
- (5) replace values of node 7 with those from successor
- (6) remove the successor (may require additional modifications to the tree)

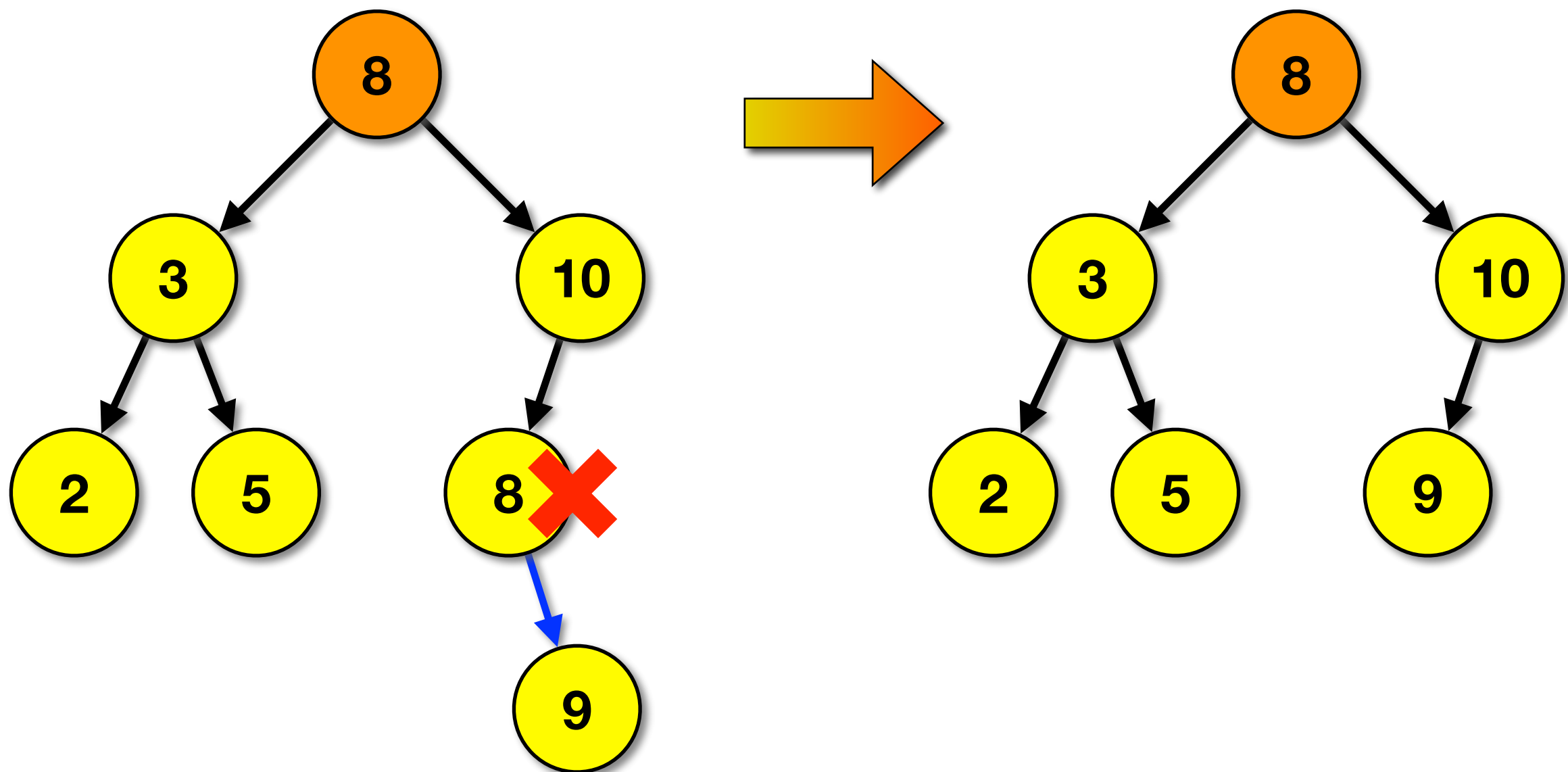
The **remove** Operation: Case #4 - Two Children

- Example of **remove** operation with a two children - **remove(7)**



The **remove** Operation: Case #4 - Two Children

- Example of **remove** operation with a two children - **remove(7)**



Analysis of BST Operations

- **Time complexity of BST operations**

	worst case	average
find	$O(N)$	$O(\log N)$
insert	$O(N)$	$O(\log N)$
remove	$O(N)$	$O(\log N)$