Lecture 12
Binary search trees

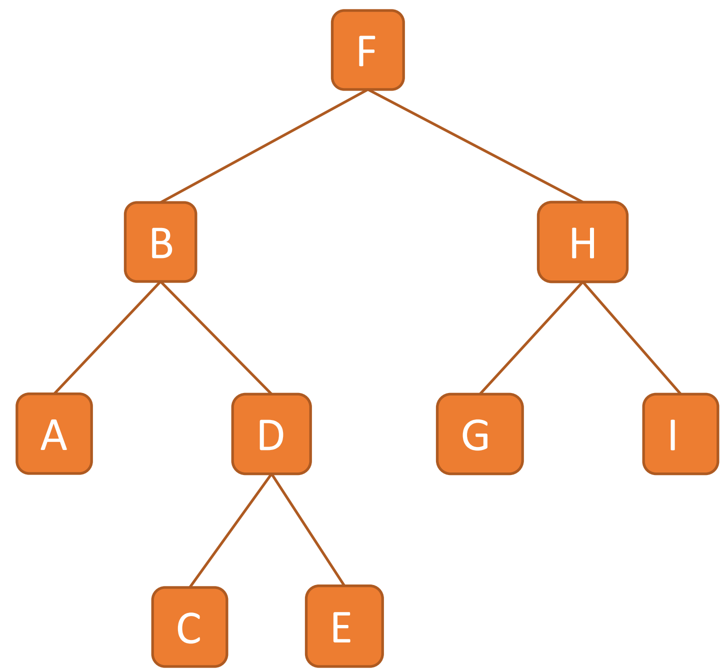
Sultan ALPAR
associate professor, IITU
s.alpar@iitu.edu.kz

Outline

1) Binary Search Trees
2) Searching BSTs
3) Adding to BSTs
4) Removing from BSTs
5) BST Analysis
6) Balancing BSTs

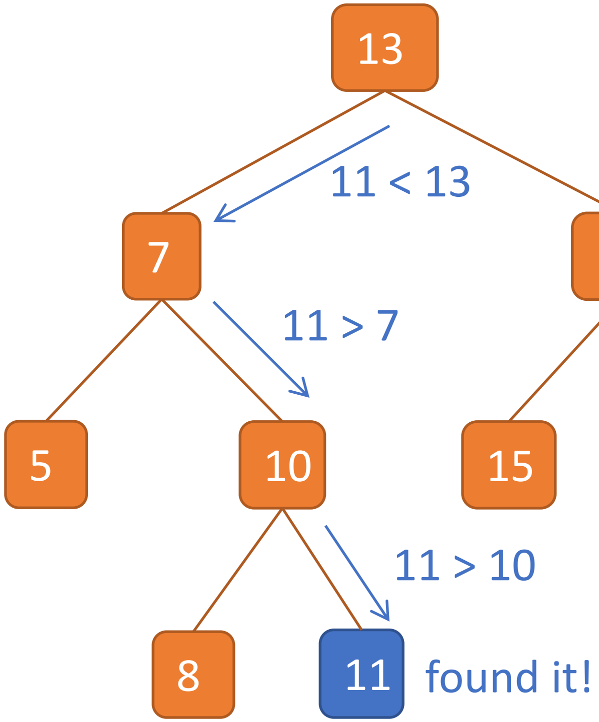
Binary Search Tree

- Binary search trees (BSTs) are binary trees with a special property
- For each node
  - All descendants in its left subtree have a lower value
  - All descendants in its right subtree have a higher value
- An in-order traversal will output the nodes in increasing order, hence the name “in-order”



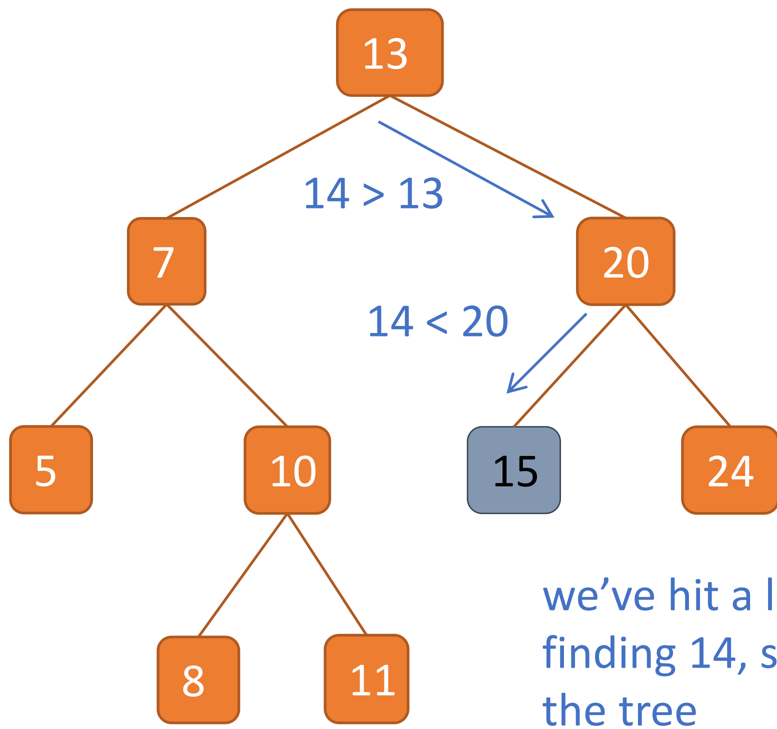
Searching a BST

- How do we implement `contains()` using a BST?
- Suppose we're looking for 11 in the tree below
- Starting at the root, each comparison tells us which subtree to look in



Searching a BST (2)

- What if an element isn't in the tree?
- Suppose we are looking for 14 in the same tree



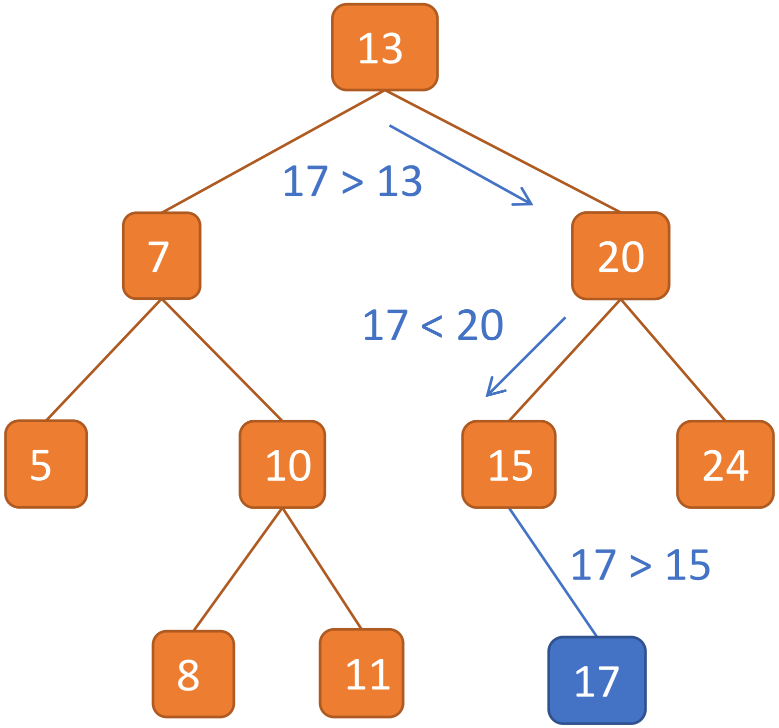
we've hit a leaf without finding 14, so it's not in the tree

BST contains()

```
function contains(node, toFind):
   // Input: node - root node of tree
   //        toFind - data of the node you’re trying to find
   // Output: the node with data toFind or null if toFind is not
   //         in BST

   if node.data == toFind:
      return node

   else if toFind < node.data and node.left != null:
      return contains(node.left, toFind)

   else if toFind > node.data and node.right != null:
      return contains(node.right, toFind)

   return null
```

Inserting into a BST

- To add an item to a BST, perform the same search to find where it should go
- An item is always added as a new leaf node
- Example: add 17

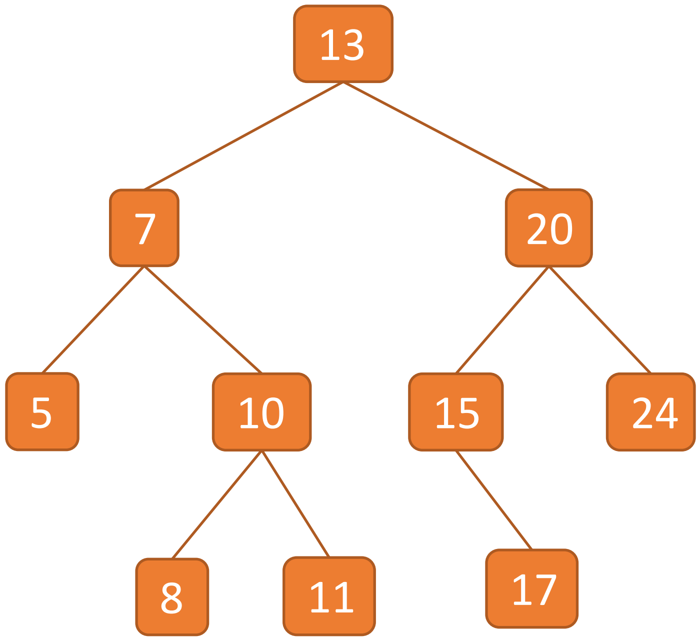


BST insert()

```
function insert(node, toInsert):
   // Input: node - root node of tree
   //        toInsert - data you are trying to insert

   if node.data == toInsert: // data already in tree
      return

   if toInsert < node.data:
      if node.left == null:
         node.addLeft(toInsert)
      else:
         insert(node.left, toInsert)
   else:
      if node.right == null:
         node.addRight(toInsert)
      else:
         insert(node.right, toInsert)
```

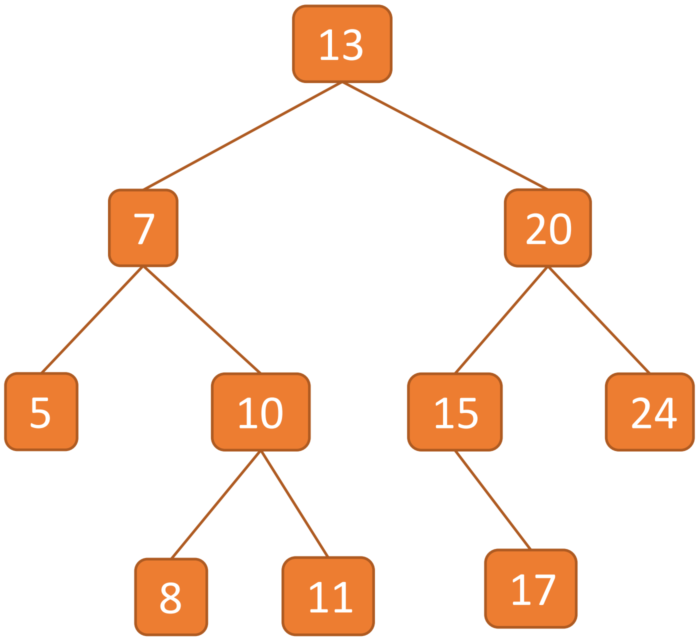
Removing from a BST

- Removing an item from a BST is tricky (sometimes).
- We have three cases to consider:
  - **1)** Removing a leaf (easy, just remove it)
  - **2)** Removing an internal node with one child
  - **3)** Removing an internal node with two children



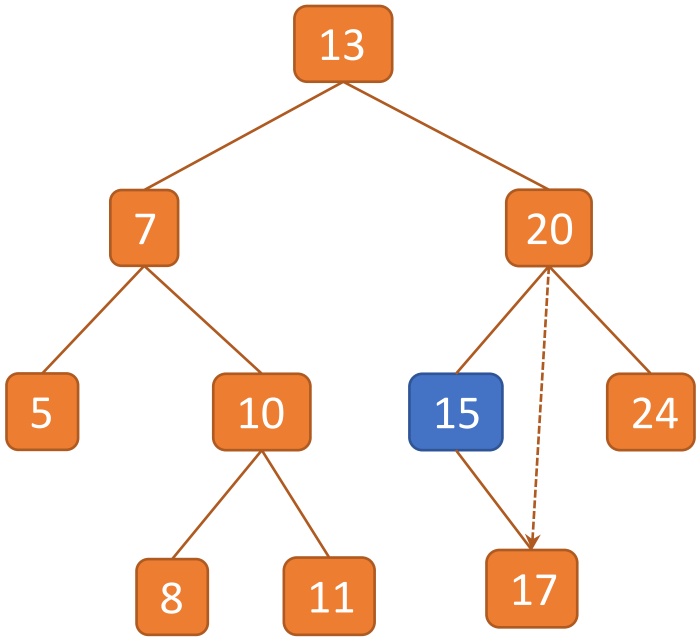
Removing from a BST – Case 2

- **Case 2:** Removing an internal node with one child
- General strategy:
  - “splice” out the node to remove by connecting the node’s parent to the node’s child.
- Example: **remove(15)**



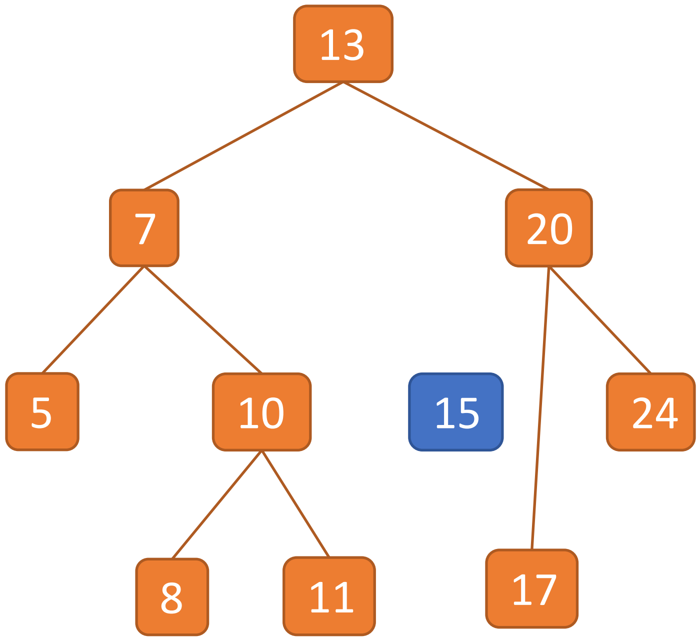
Removing from a BST – Case 2

- **Case 2:** Removing an internal node with one child
- General strategy:
  - "splice" out the node to remove by connecting the node's parent to the node's child.
- Example: **remove(15)**
  - We set the parent node's `left` reference to the given node's only child



Removing from a BST – Case 2

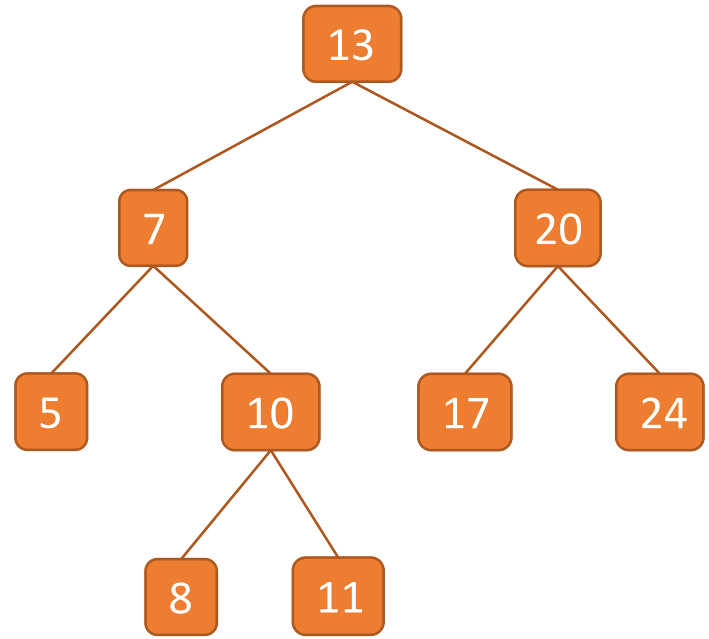
- **Case 2:** Removing an internal node with one child
- General strategy:
  - "splice" out the node to remove by connecting the node's parent to the node's child.



- Example: **remove(15)**
  - We set the parent node's `left` reference to the given node's only child
  - There are no more references to the given node, so it is deleted (garbage collected)

Removing from a BST – Case 2

- **Case 2:** Removing an internal node with one child
- General strategy:
  - “splice” out the node to remove by connecting the node’s parent to the node’s child.
- Example: **remove(15)**
  - We set the parent node’s `left` reference to the given node’s only child
  - There are no more references to the given node, so it is deleted (garbage collected)
  - BST order is maintained

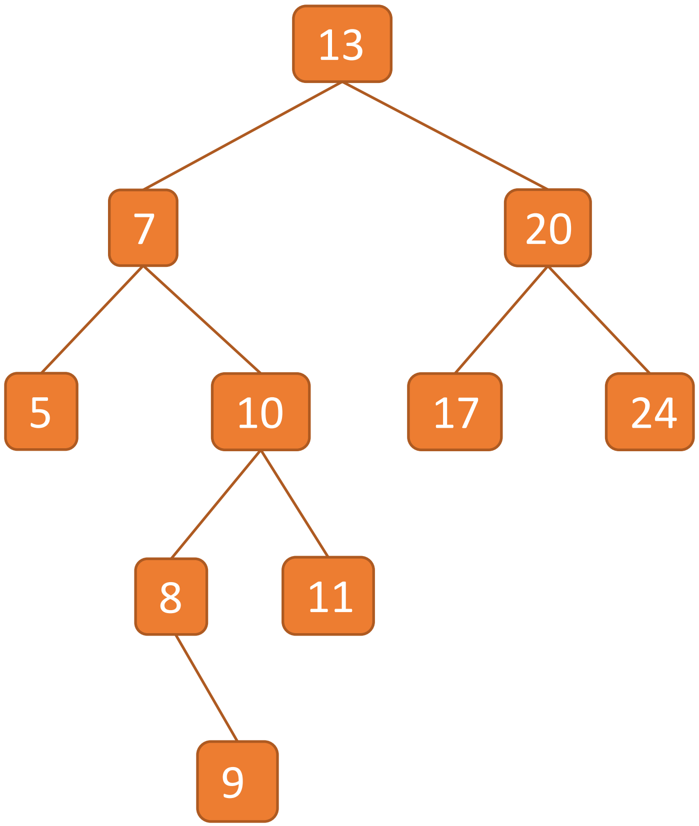


Removing from a BST – Case 3

- **Case 3:** Removing an internal node with two children

- General strategy:
  - Replace the data of the node to remove with the data of the node's successor.
  - Delete the successor.

- The successor is also called the **in-order successor**, because it comes next in the in-order tree traversal.

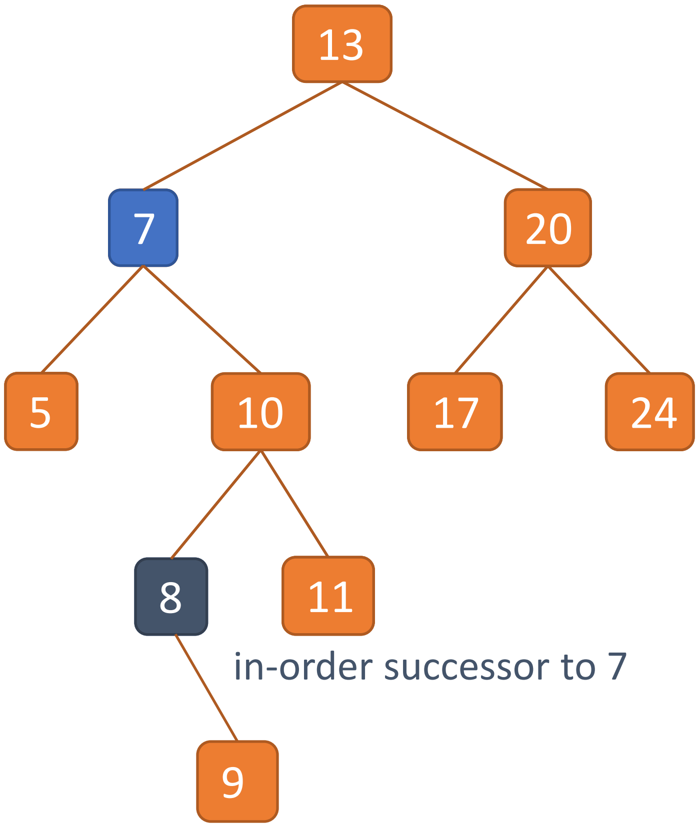


(first, to help with our example, we made a small addition to this tree)

Removing from a BST – Case 3

- **Case 3:** Removing an internal node with two children

- Example: **remove(7)**
  - First, let's find the **in-order successor** to the given node.
  - Since we know the given node has **two** children – which guarantees the node has a right subtree – we know that its successor will always be the left-most node in its right subtree.

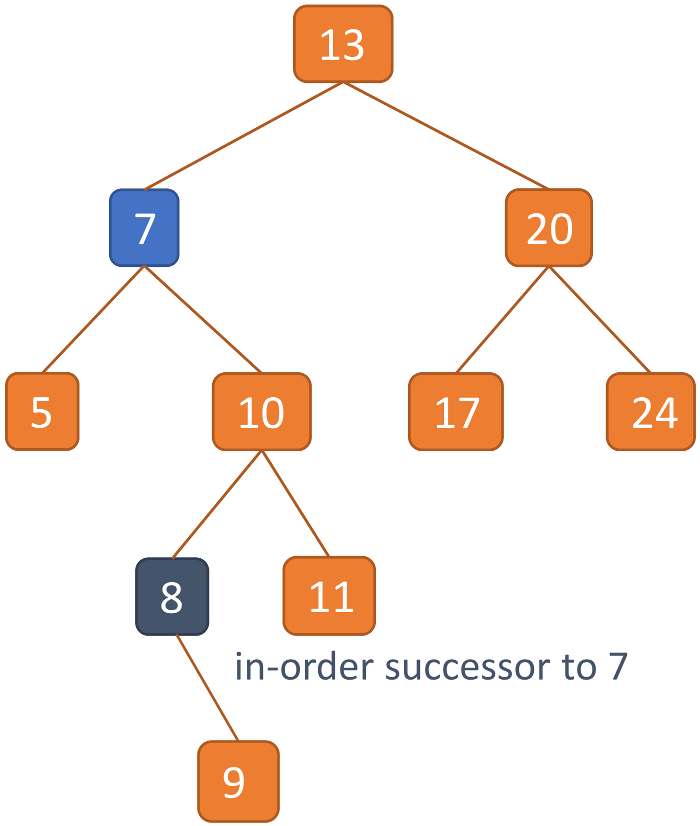


Removing from a BST – Case 3

- **Case 3:** Removing an internal node with two children

- Example: **remove(7)**
  - Code to find the in-order successor:

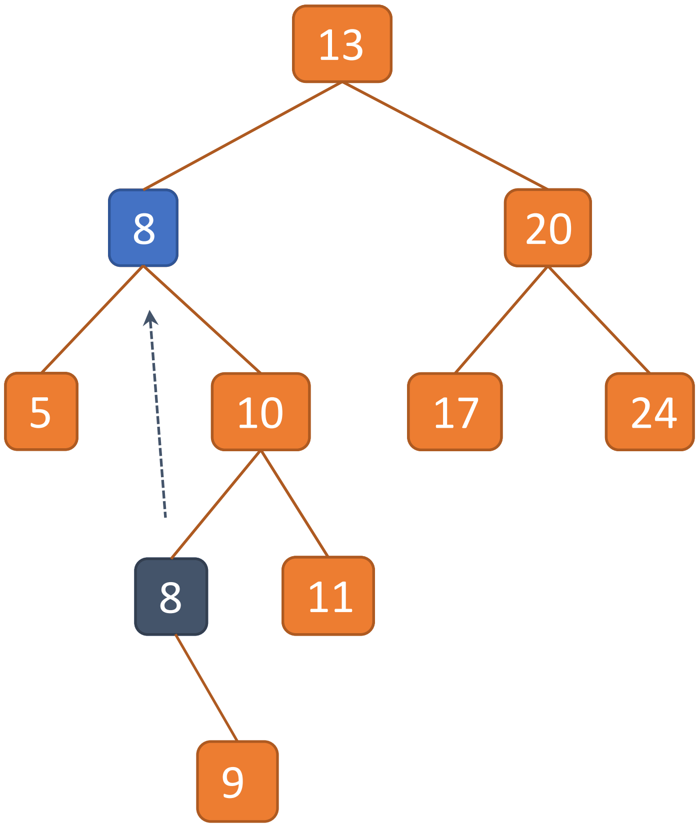
```
successor(node):
    // Input: node – the node for
    // which to find the successor
    curr = node.right
    while (curr.left != null):
        curr = curr.left
    return curr
```



Removing from a BST – Case 3

- **Case 3:** Removing an internal node with two children

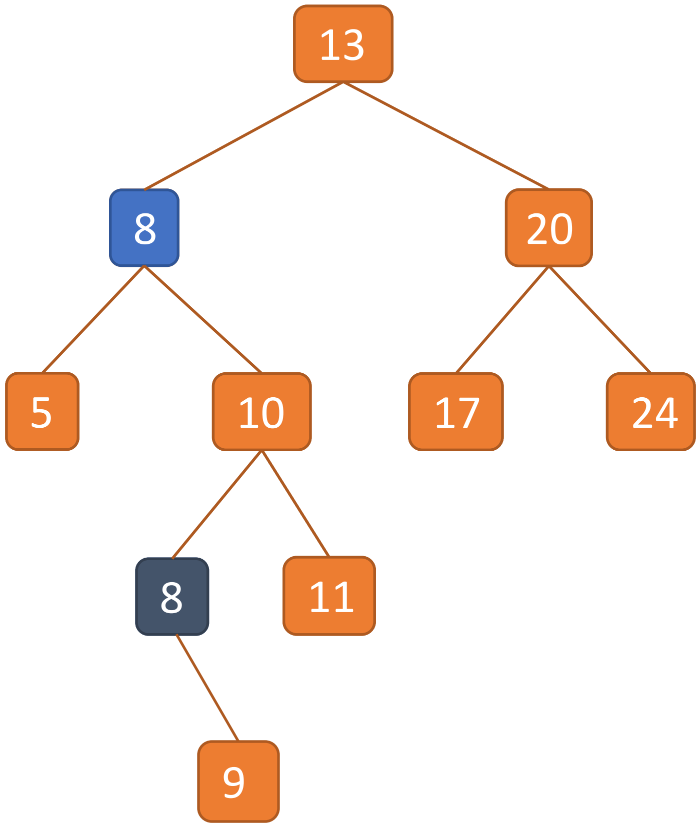
- Example: **remove(7)**
  - Second, let's replace the data of the node to remove with that of its successor.



Removing from a BST – Case 3

- **Case 3:** Removing an internal node with two children

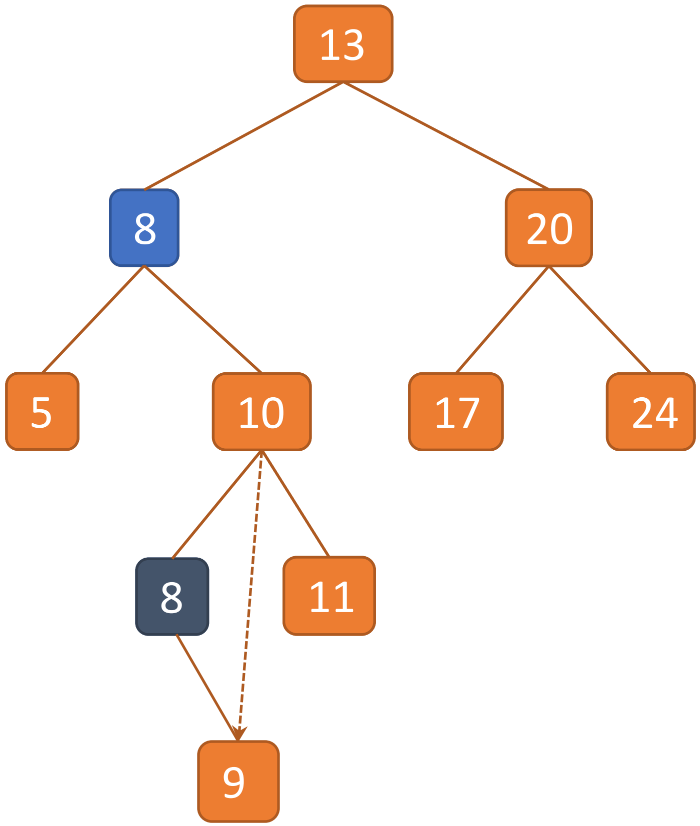
- Example: **remove(7)**
  - Lastly, remove the successor.
  - Notice that we can make one very important guarantee: the successor cannot have a left child, otherwise that child would have been the in-order successor to 7. Thus, **the successor can have at most one (right) child**.
  - Therefore, we can delete the successor according to the strategies we defined for Cases 1 and 2.



Removing from a BST – Case 3

- **Case 3:** Removing an internal node with two children

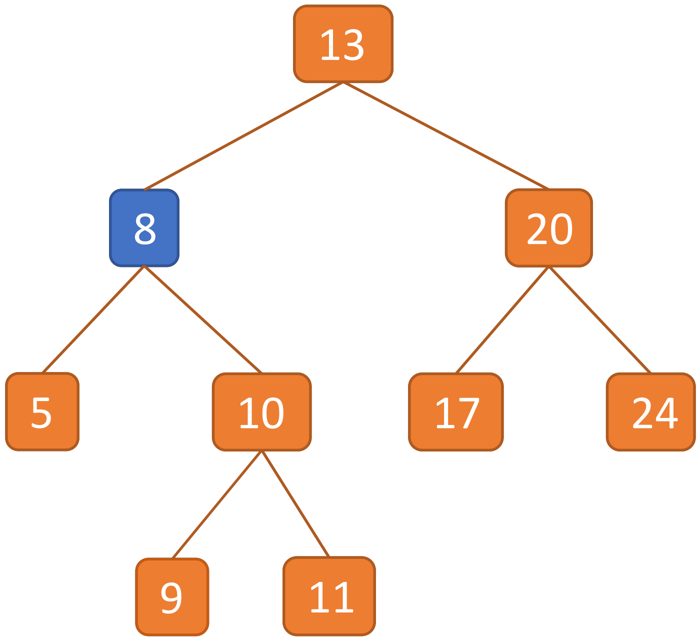
- Example: **remove(7)**
  - In this case, we remove the successor according to Case 2: internal node with one child.



Removing from a BST – Case 3

- **Case 3:** Removing an internal node with two children

- Example: **remove(7)**
  - Successor is removed.
  - BST order is maintained.



BST remove()

```
function remove(node):
   // Input: node – the node we are trying to remove. We can find this node
   //                    by calling contains()

   if node has no children: // case 1 – node is a leaf
      node.parent.removeChild(node)
   else if node only has left child: // case 2a – only left child
      if node.parent.left == node: // if node is a left child
         node.parent.left = node.left
      else:
         node.parent.right = node.left
   else if node only has right child: // case 2b – only right child
      if node.parent.left == node:
         node.parent.left = node.right
      else:
         node.parent.right = node.right
   else:  // case 3 – node has two children
      nextNode = successor(node)
      node.data = nextNode.data // replace node's data with that of nextNode
      remove(nextNode) // nextNode guaranteed to have at most one child
```

Successor vs. Predecessor

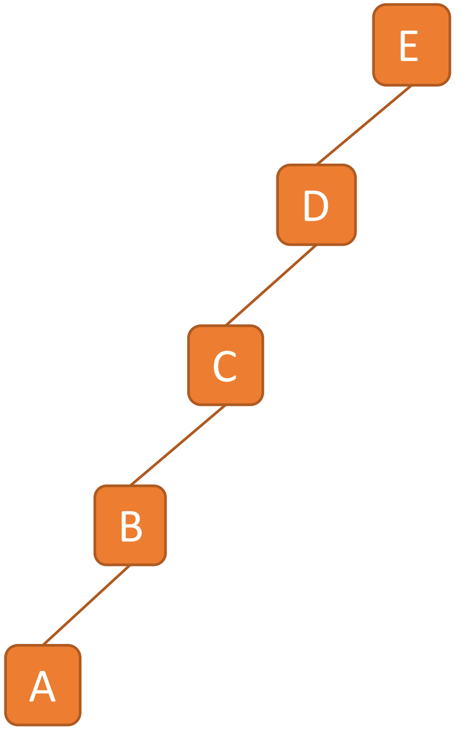
- It should be noted that it is perfectly valid to use a node's in-order predecessor in place of its successor in our BST remove() algorithm.

- It doesn't matter if you use one over the other, but randomly picking between the two helps keep the tree more balanced

- In case 3, the predecessor would be the right-most node of the given node's left subtree.

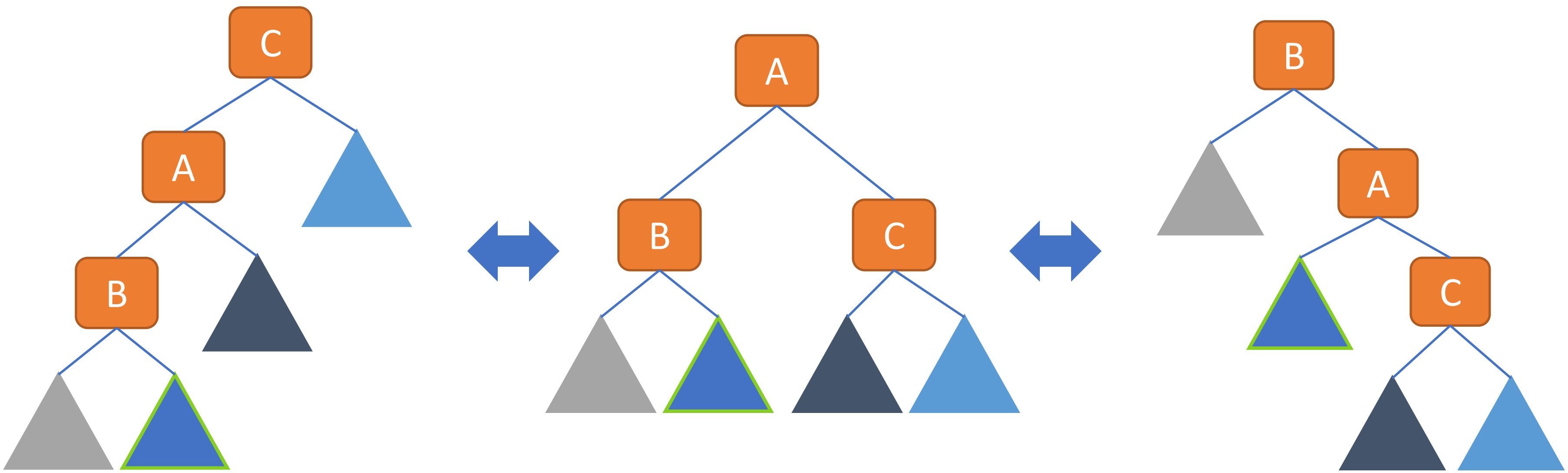
BST Analysis

- How fast are the BST functions?
- Depends on the height of the tree! The worst case requires traversing all the way to the leaf with the greatest depth
- If the tree is perfectly balanced, then its height is about $\log_2 n$, which would let the BST functions run in $O(\log n)$ time
- But in the extremely unbalanced case, a binary search tree just becomes a sorted linked list, and its functions run in $O(n)$ time



Balancing a BST

- If a binary tree becomes unbalanced, we can fix it by performing a series of tree rotations



Observe that the in-order ordering of each of these trees remains

which means that the BST order is preserved between rotations