CSC 225 - Summer 2019

Dictionaries

Bill Bird

Department of Computer Science
University of Victoria

June 26, 2019

Dictionaries

A **Dictionary** data structure stores a collection of elements, each with a **key**. Dictionaries have three core operations:

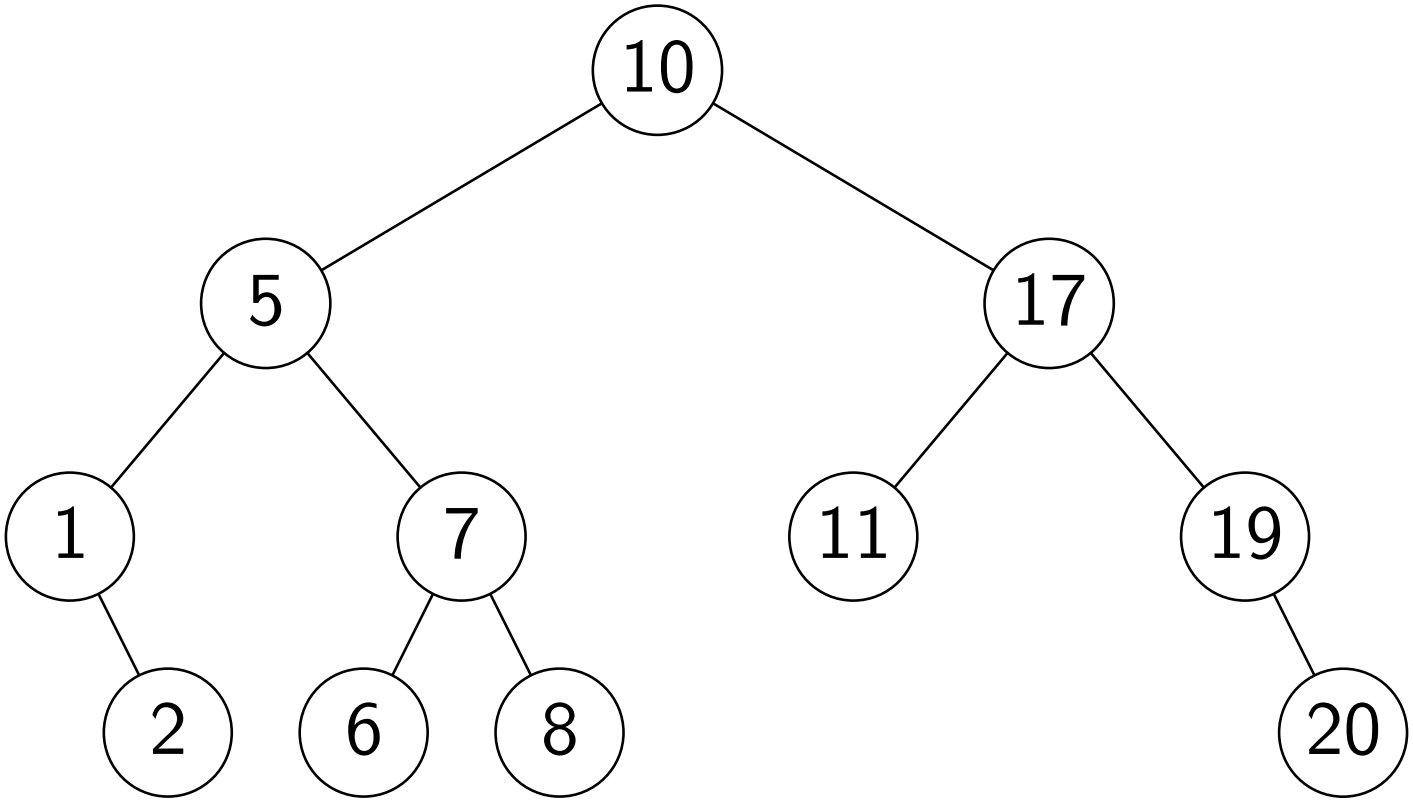
1. INSERT(e): Add the element e to the collection.
2. FIND(k): Search for an element with key k and return it if it was found.
3. REMOVE(k): Remove key k from the dictionary.

In practice, keys are often a single component of a compound data type (for example, if a set of student records were stored in a dictionary, the student number or last name might be the key). In this course, for simplicity, keys are often the only data stored in dictionaries.

Dictionary Implementations

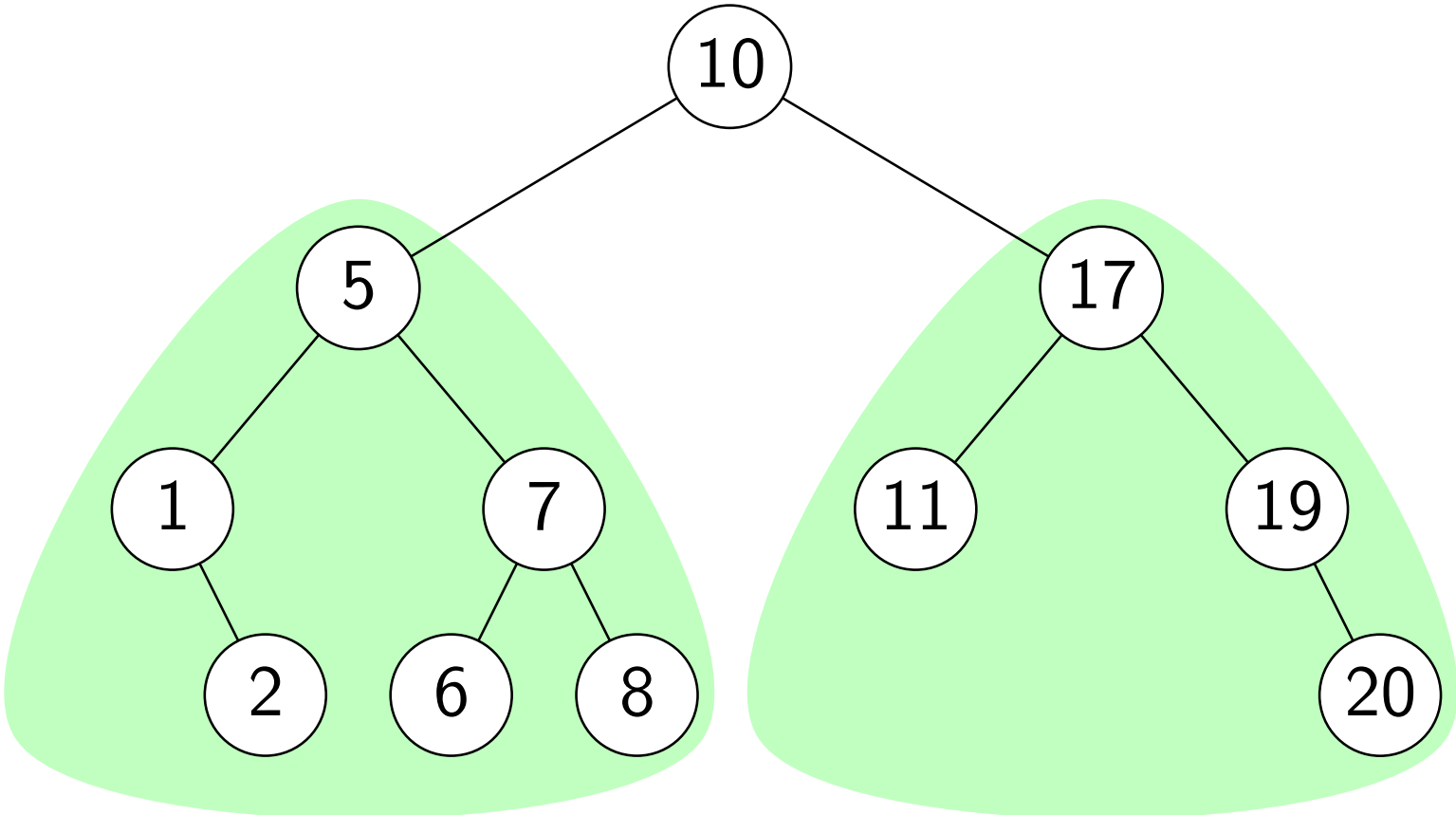
	Insert		Find	
	Expected Case	**Worst Case**	**Expected Case**	**Worst Case**
Sequence Based				
Unsorted Array	$\Theta(1)$	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$
Sorted Array	$\Theta(n)$	$\Theta(n)$	$\Theta(\log n)$	$\Theta(\log n)$
Unsorted List	$\Theta(1)$	$\Theta(1)$	$\Theta(n)$	$\Theta(n)$
Tree Based				
'Normal' Binary Search Tree	$\Theta(\log n)$	$\Theta(n)$	$\Theta(\log n)$	$\Theta(n)$
2-3 Tree	$\Theta(\log n)$	$\Theta(\log n)$	$\Theta(\log n)$	$\Theta(\log n)$
Red/Black Tree	$\Theta(\log n)$	$\Theta(\log n)$	$\Theta(\log n)$	$\Theta(\log n)$
AVL Tree	$\Theta(\log n)$	$\Theta(\log n)$	$\Theta(\log n)$	$\Theta(\log n)$
B-Tree	$\Theta(\log n)$	$\Theta(\log n)$	$\Theta(\log n)$	$\Theta(\log n)$
Table Based				
Hash Table	$\Theta(1)$	$\Theta(n)$	$\Theta(1)$	$\Theta(n)$

Binary Search Trees (1)



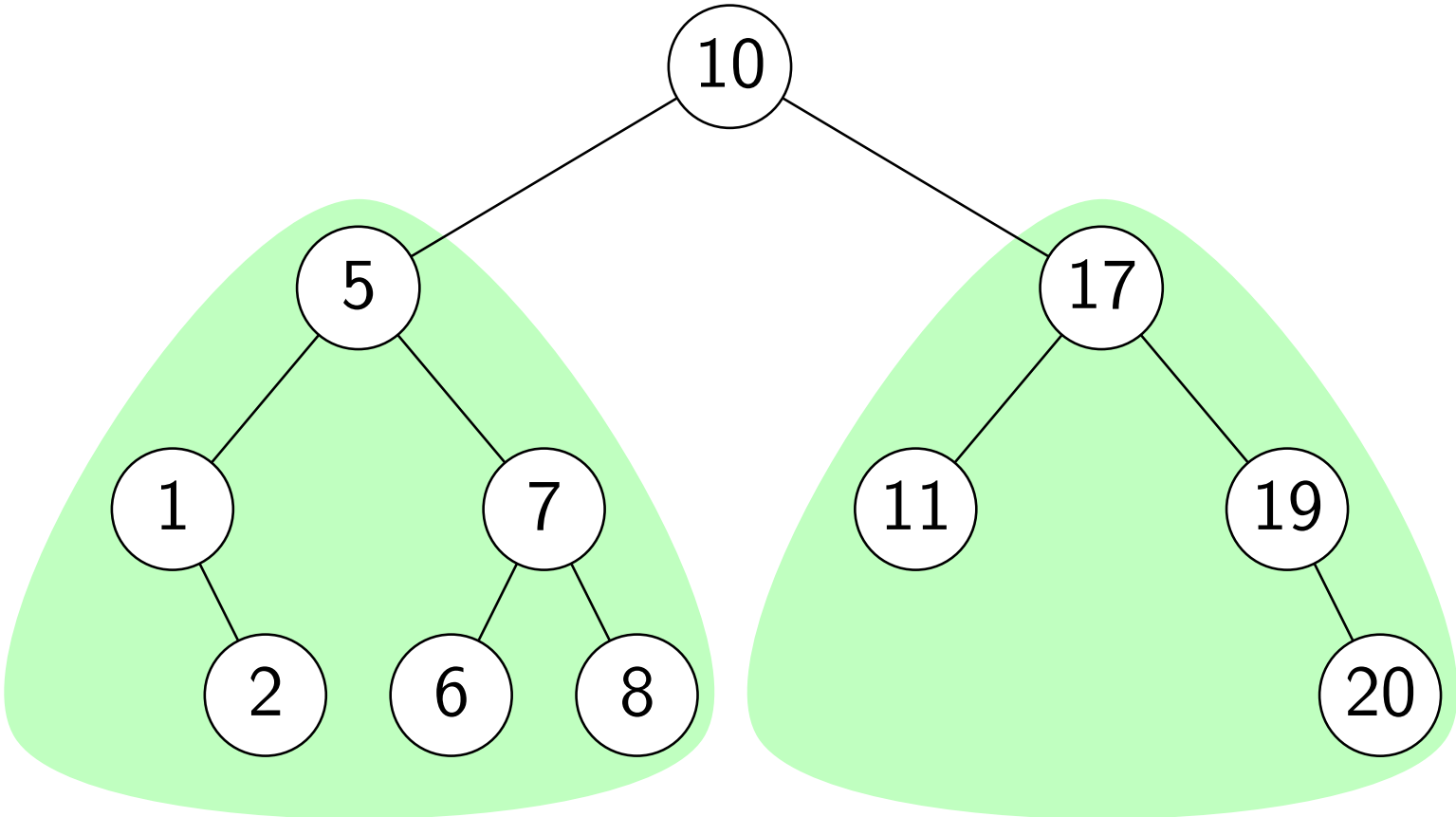
A *binary search tree* is a binary tree in which an in-order traversal produces a sorted sequence.

Binary Search Trees (2)



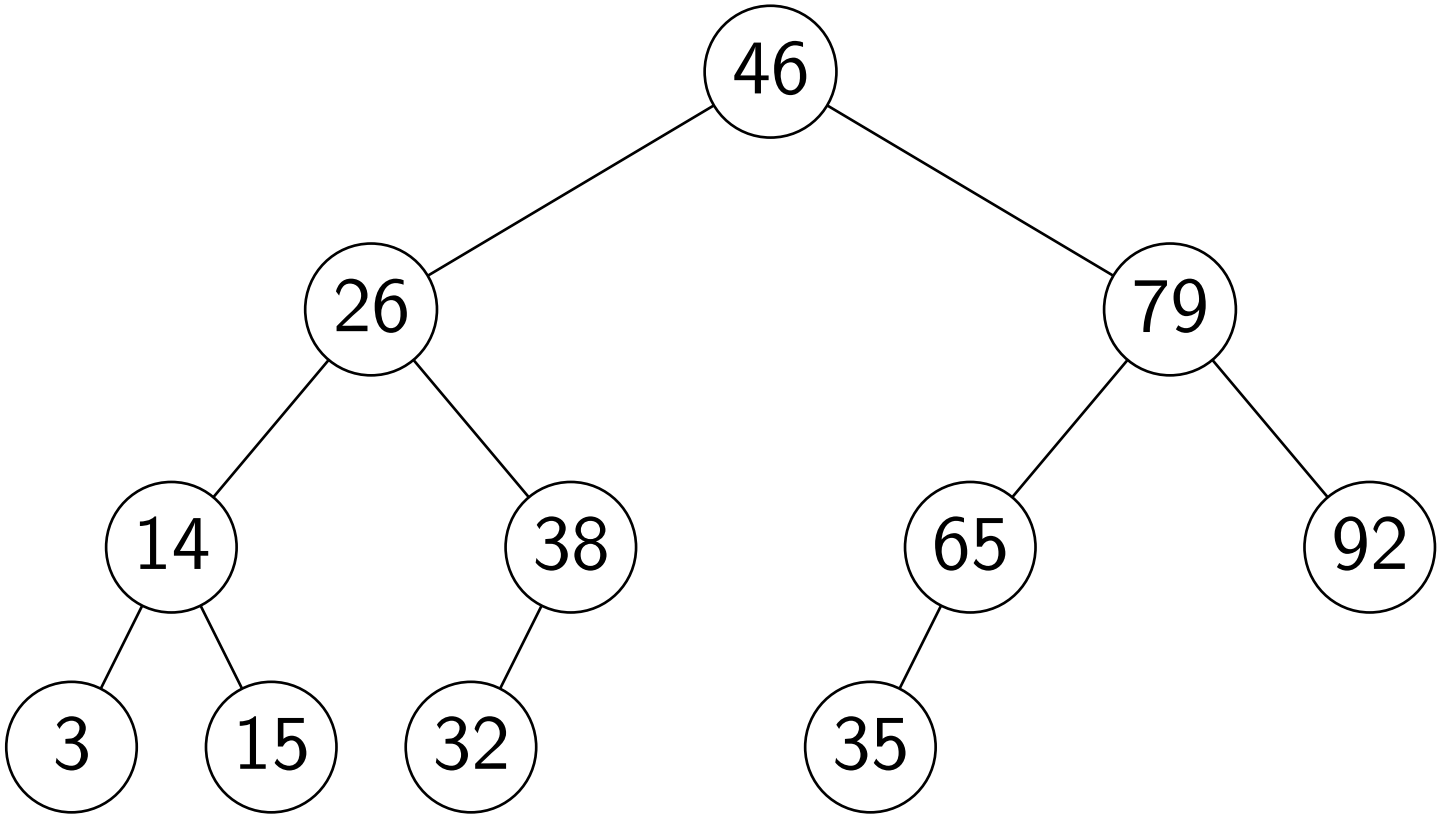
Alternatively, a binary search tree can be defined as a binary tree in which, for every node v, no nodes in the left subtree of v are greater than v and no nodes in the right subtree of v are less than v.

Binary Search Trees (3)



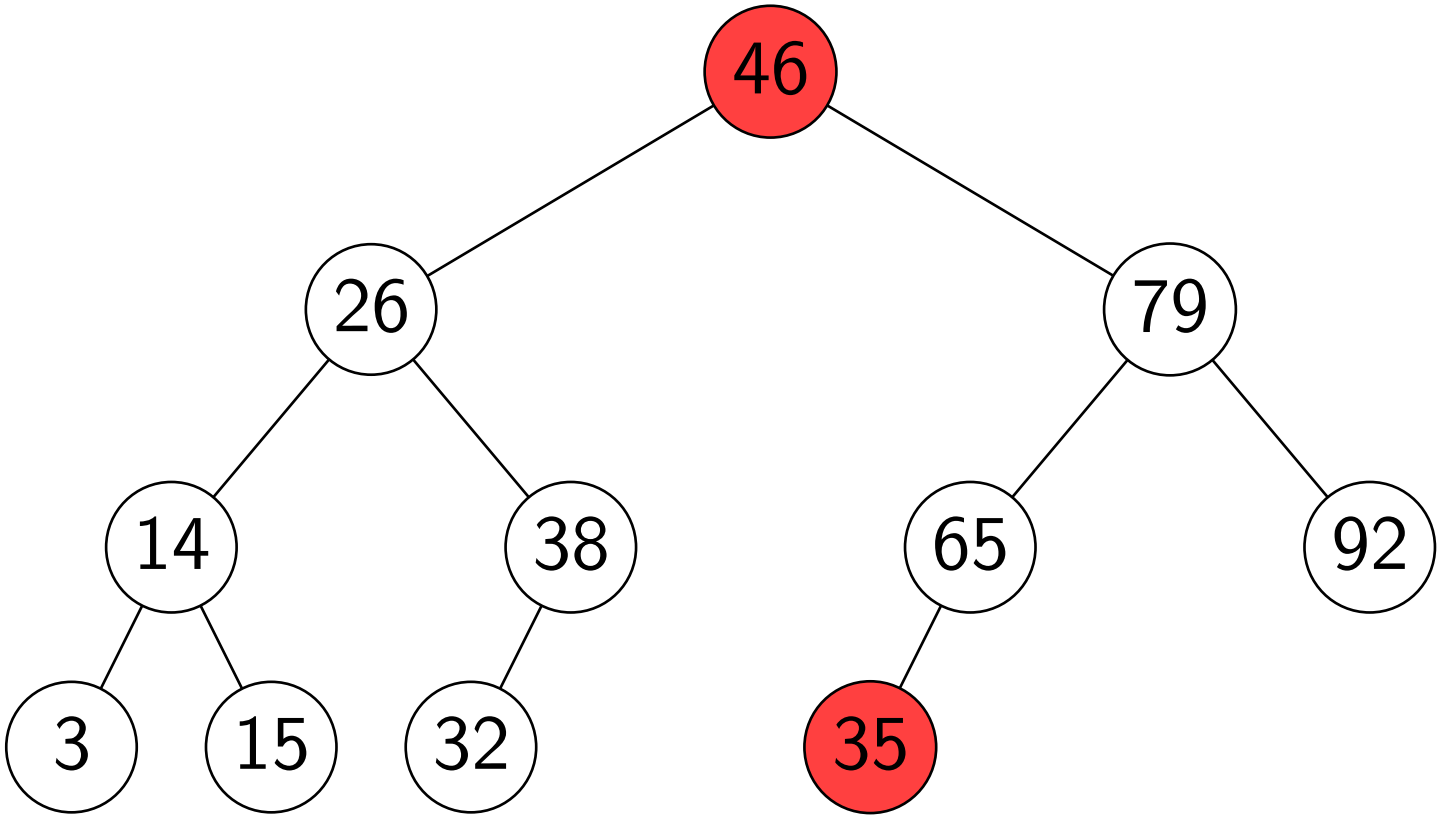
The subtrees of any node in a binary search tree are also binary search trees.

Binary Search Trees (4)



- Is this a binary search tree?

Binary Search Trees (5)



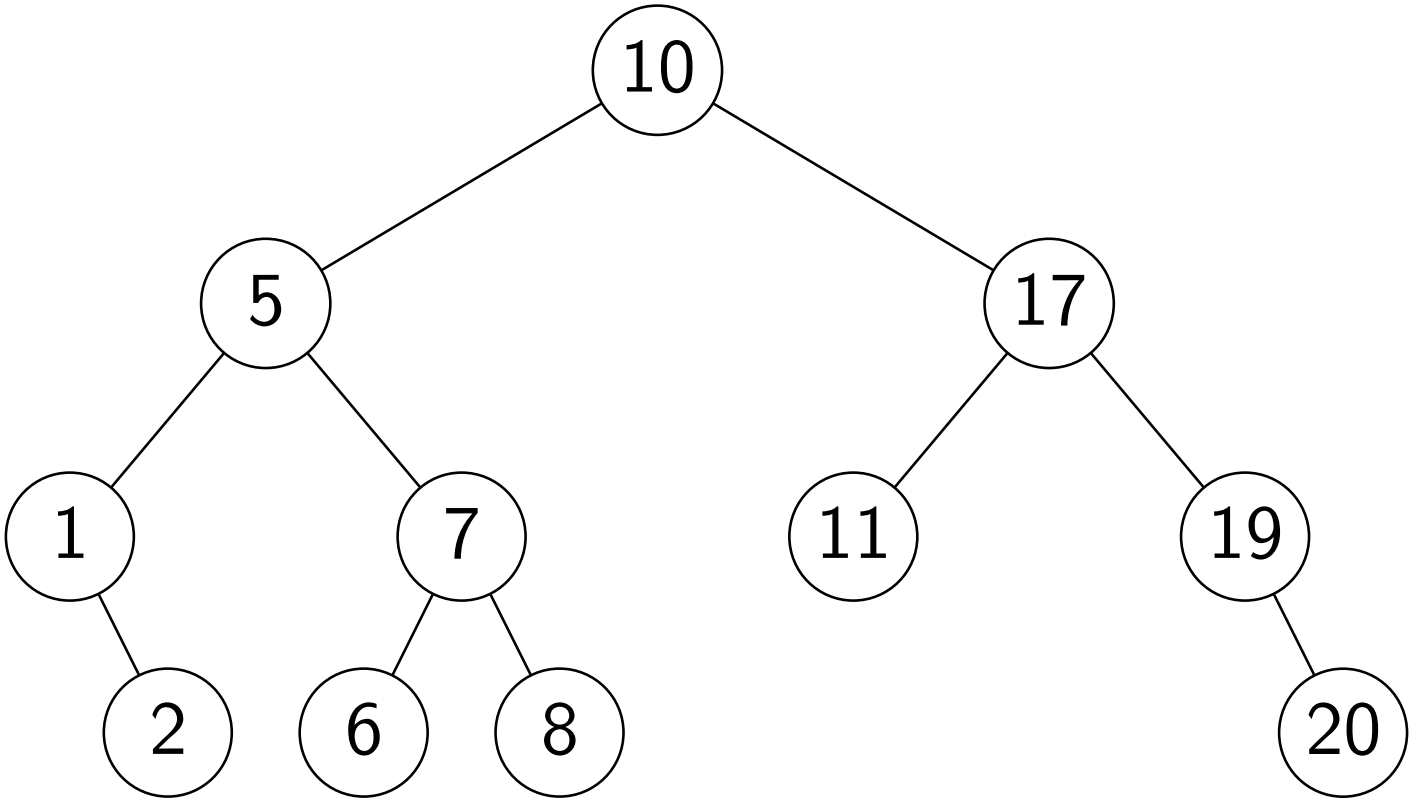
- Is this a binary search tree?
- **No**. The two highlighted nodes appear out of order in an in-order traversal.

BST Find (1)

```
1:  procedure BSTFind(k, node)
2:      if node = null then
3:          return NOT_FOUND
4:      end if
5:      if node.value > k then
6:          return BSTFind(k, node.left)
7:      else if node.value < k then
8:          return BSTFind(k, node.right)
9:      else
10:         return node
11:     end if
12: end procedure
```

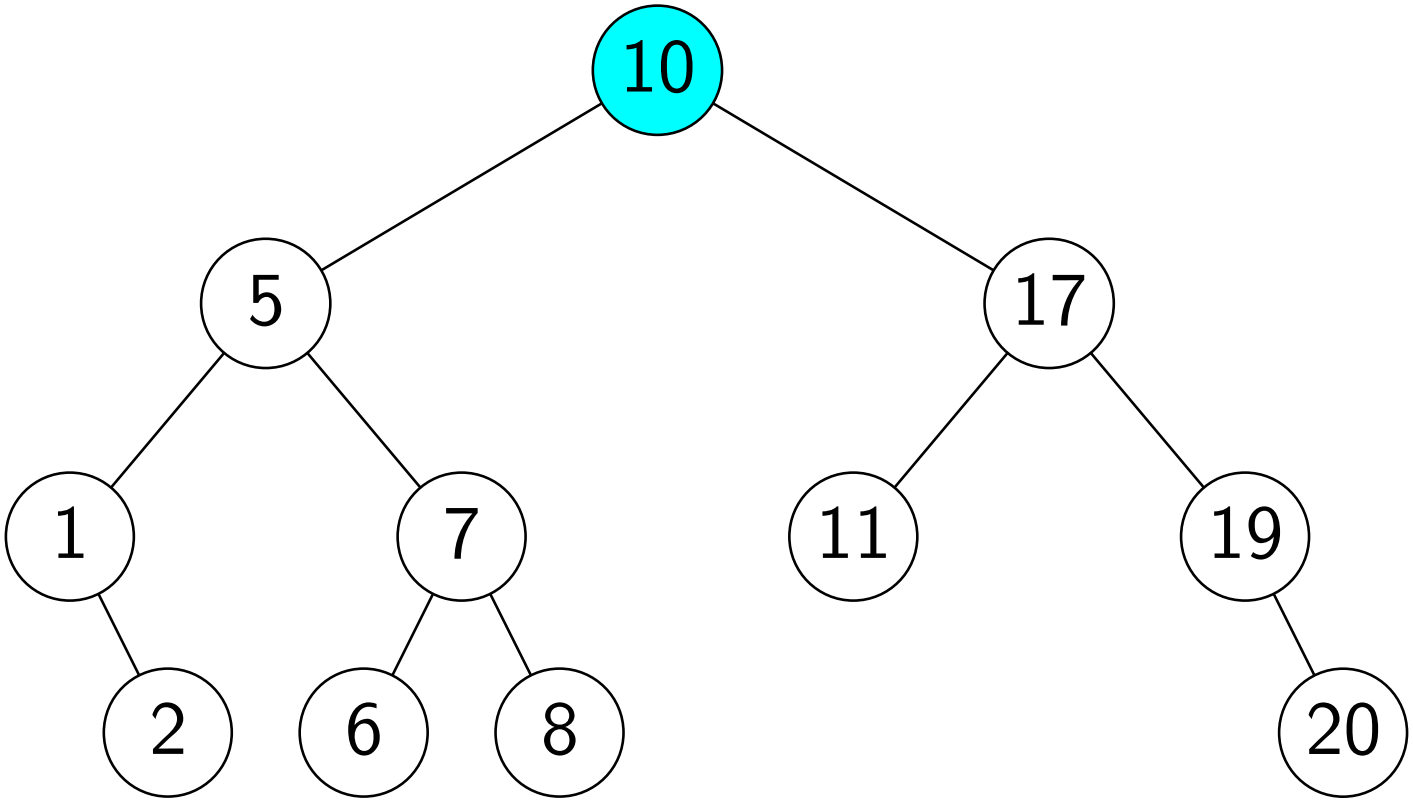
- The Find Operation in a binary search tree can be implemented with a recursive search starting at the root.

BST Find (2)



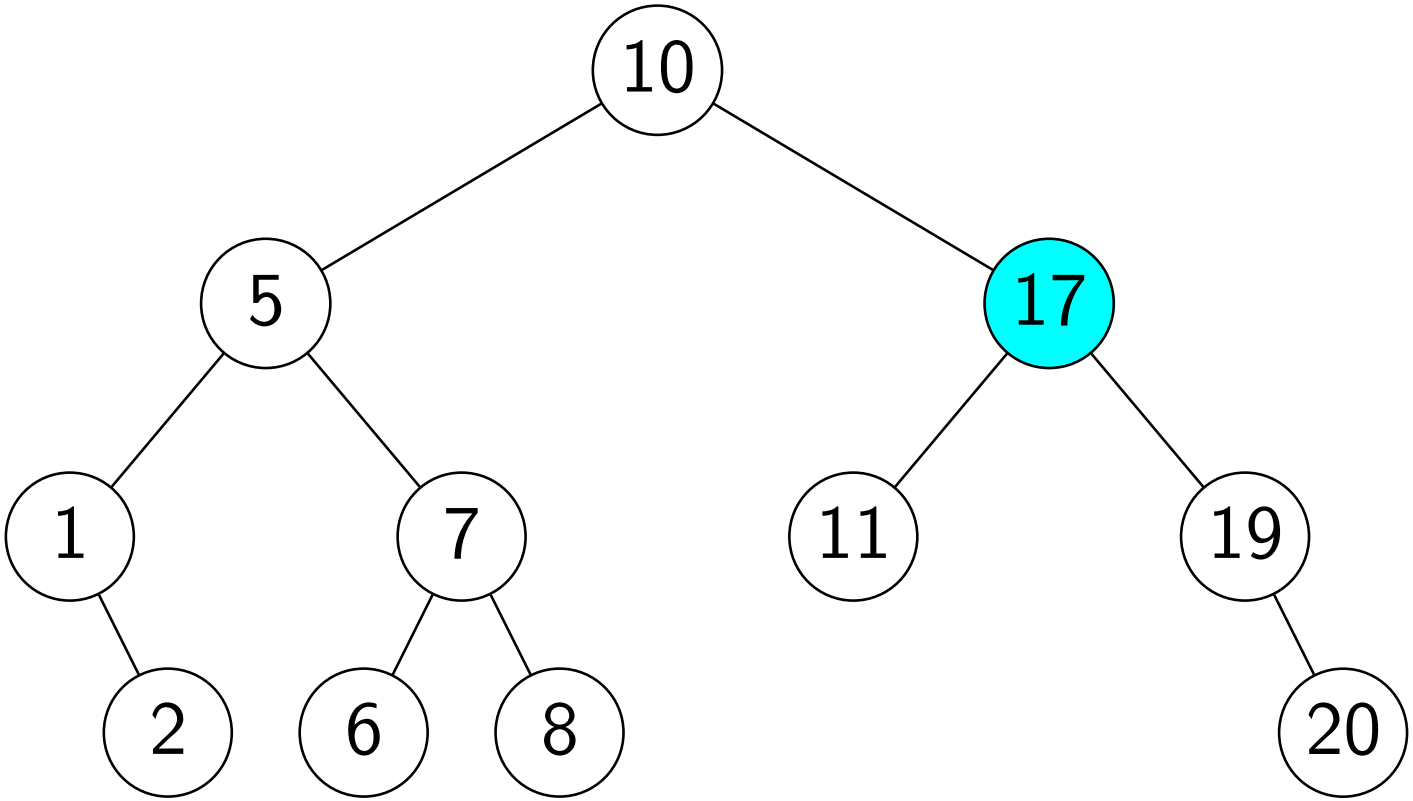
- **Exercise**: Search for the key 11 in the binary search tree above.

BST Find (3)



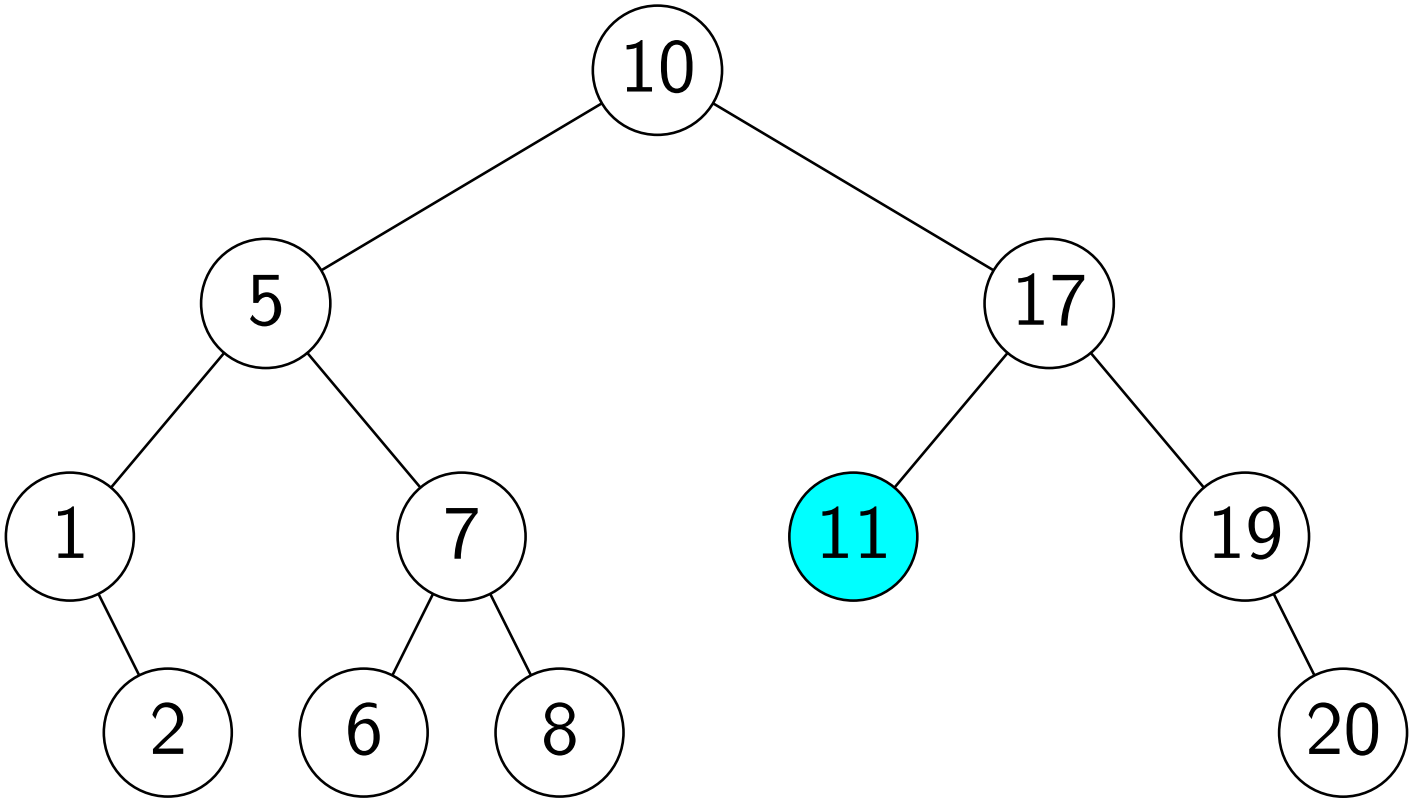
- The initial call is $\text{BSTFind}(11, \texttt{root})$

BST Find (4)



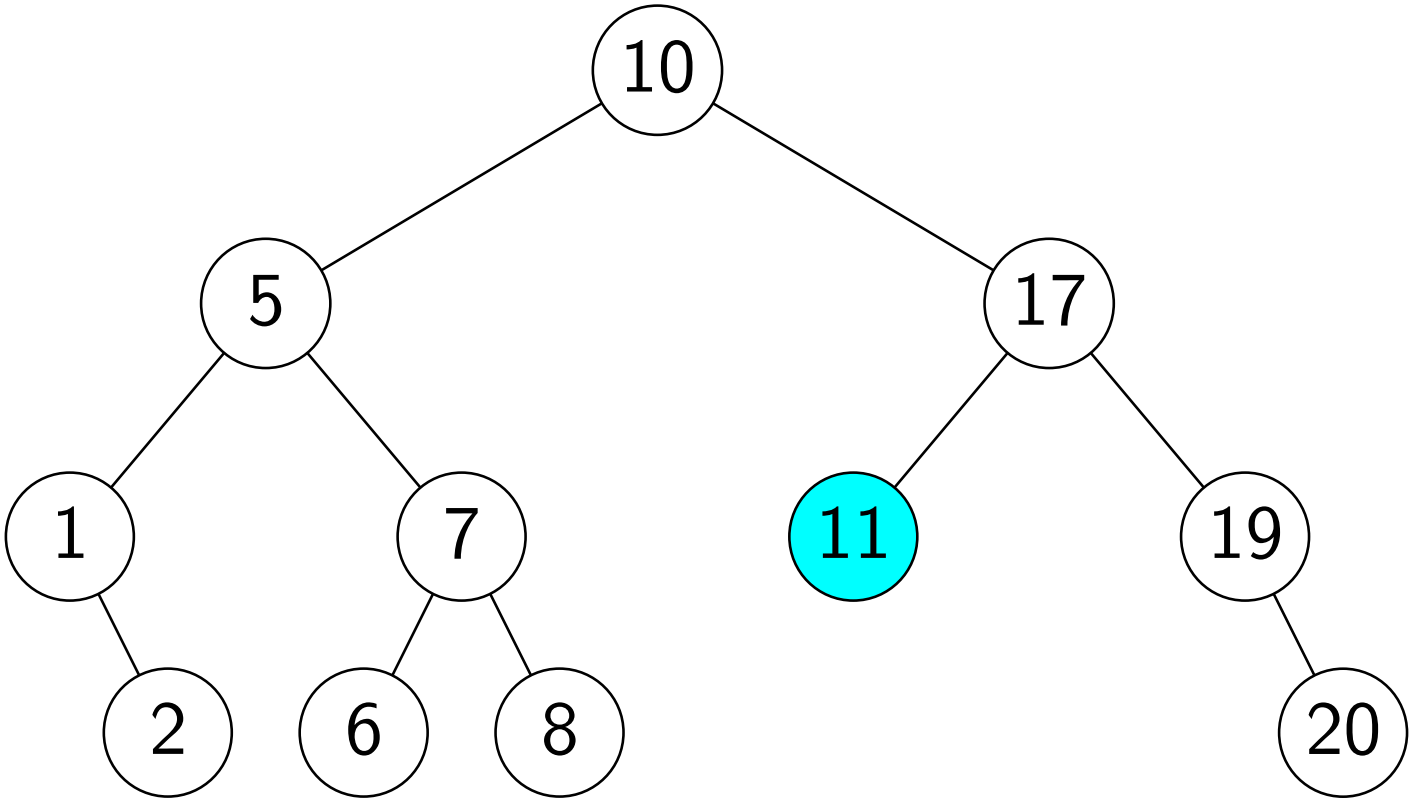
- Since $11 > 10$, the search goes to the right.

BST Find (5)



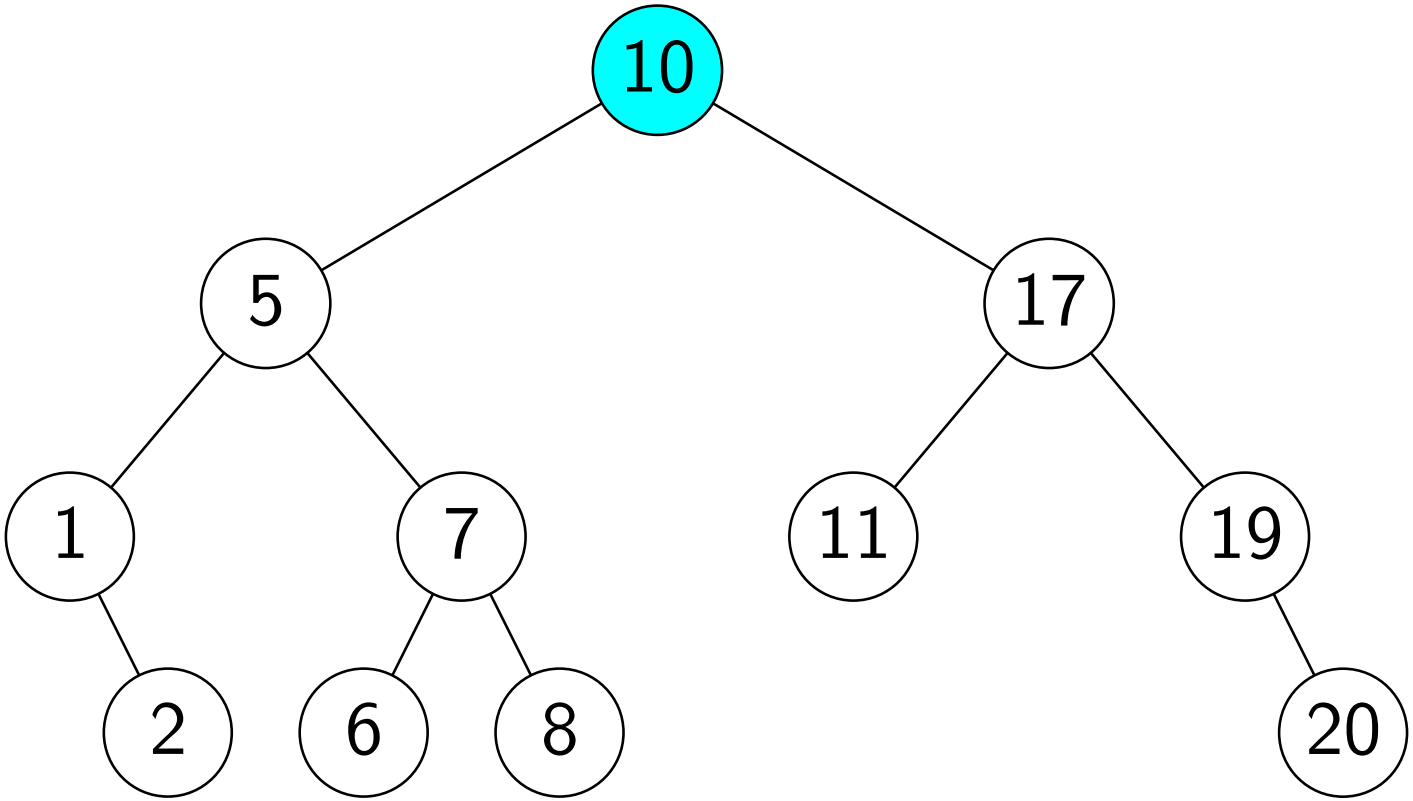
- Since $11 < 17$, the search goes to the left.

BST Find (6)



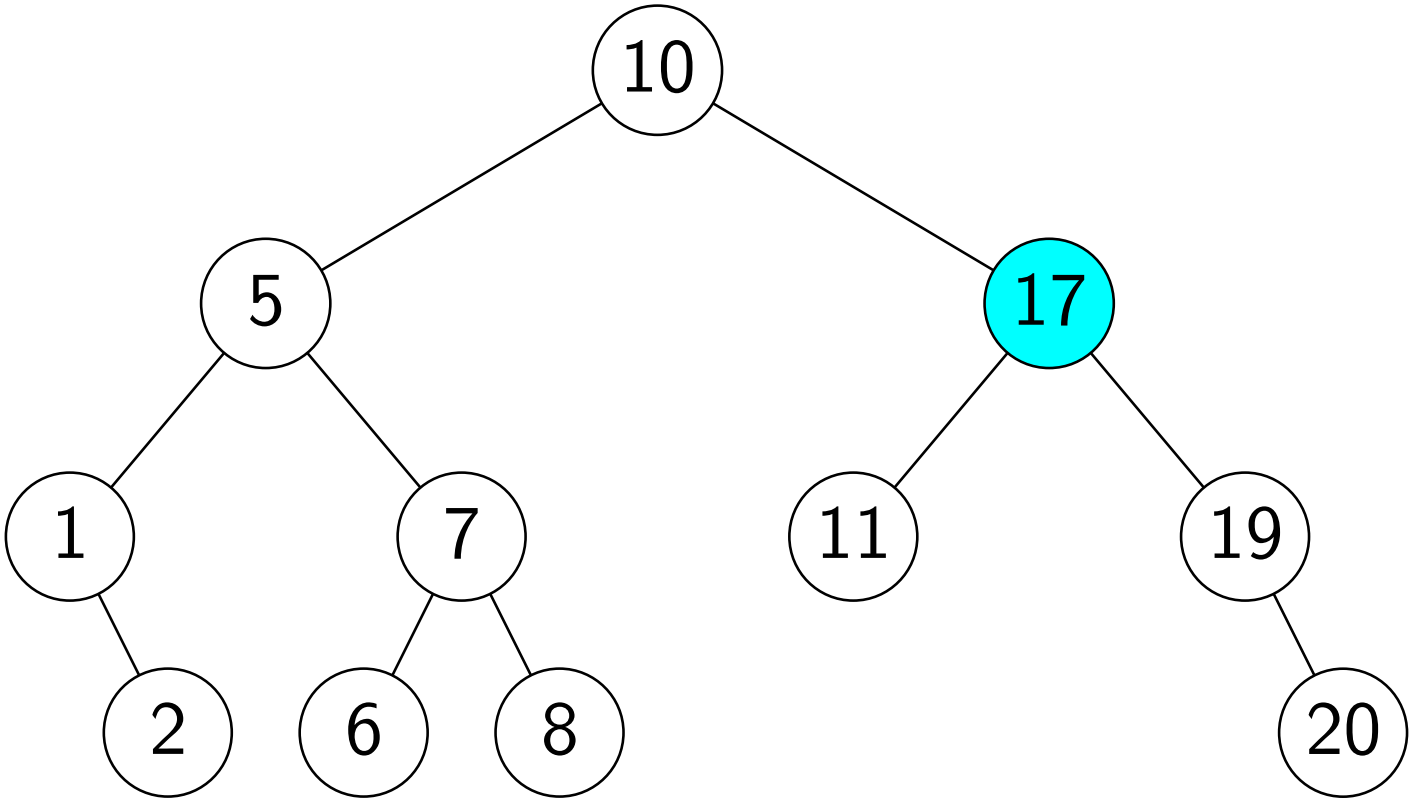
- In the worst case, the BSTFIND algorithm traces a path from the root to a leaf.
- The total running time is $O(h)$ in a tree of height h.

BST Find (7)



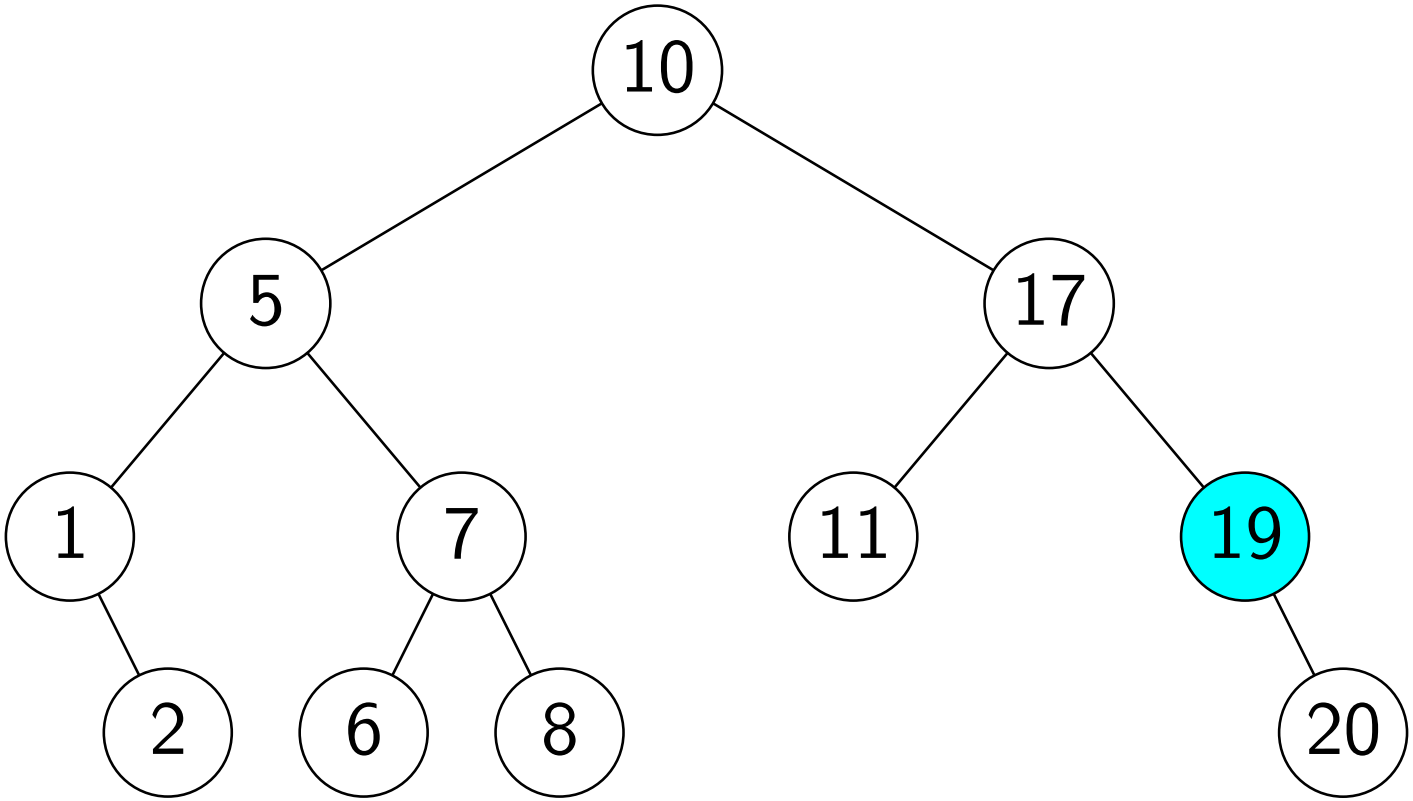
- **Exercise**: Search for the key 18 in the binary search tree above.

BST Find (8)



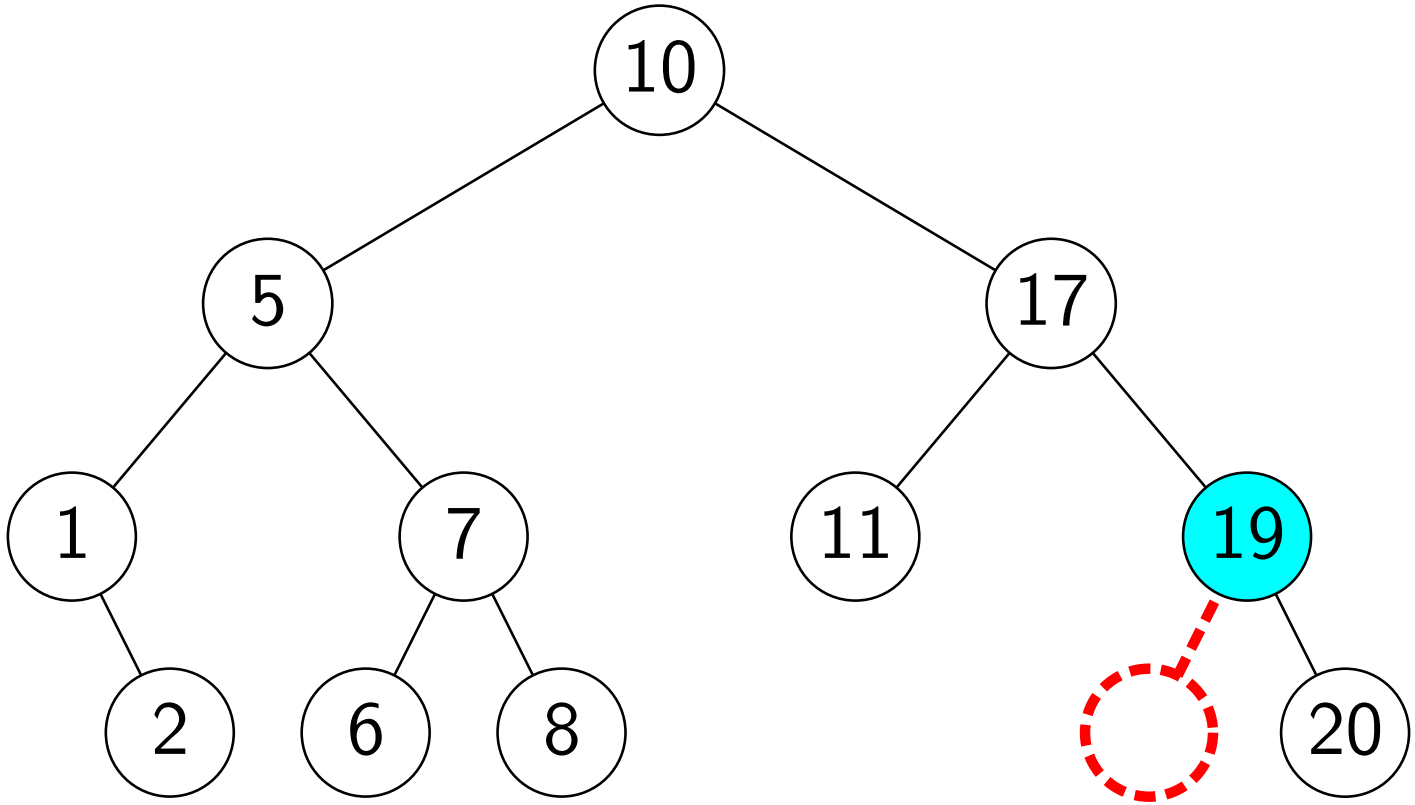
- Since $18 > 10$, the search goes to the right.

BST Find (9)



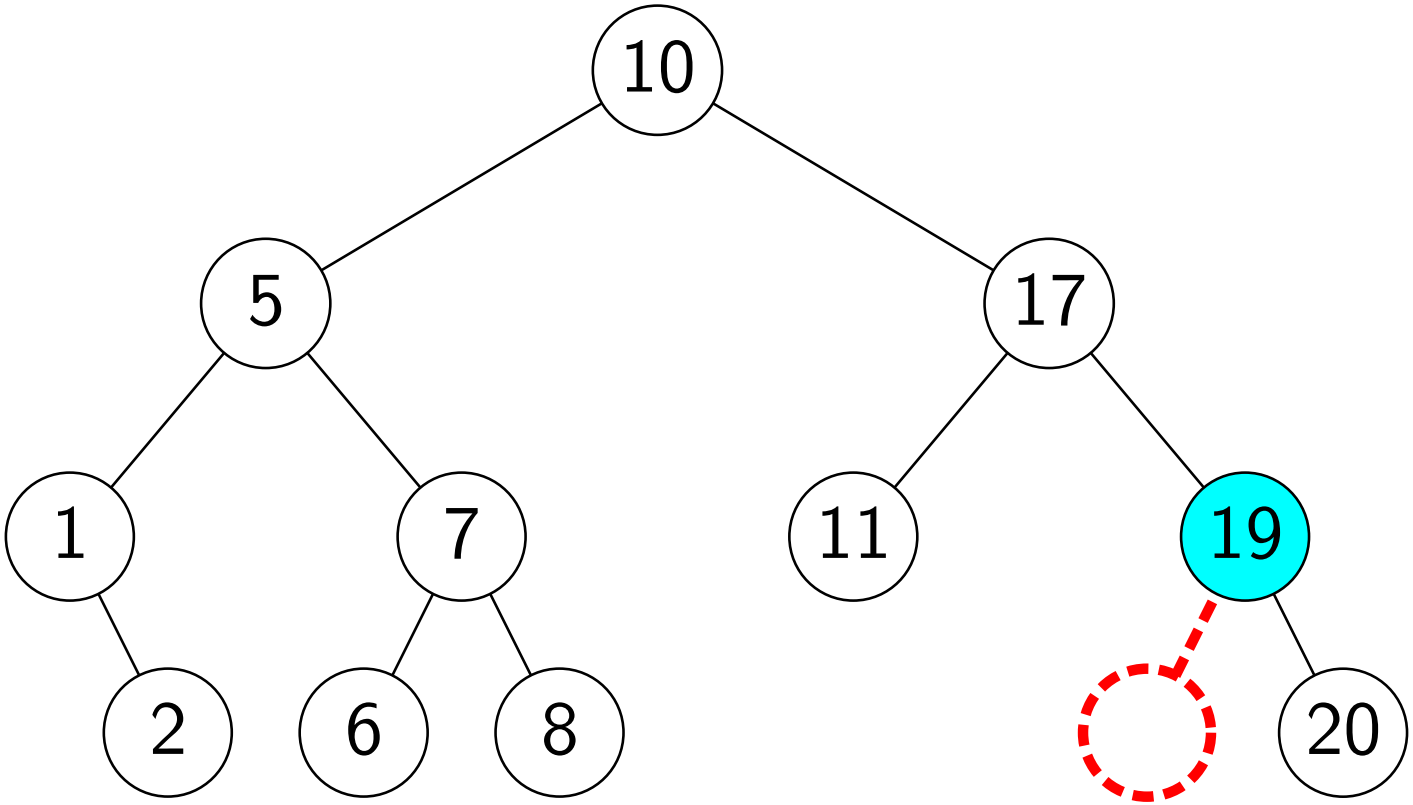
- Since $18 > 17$, the search goes to the right.

BST Find (10)



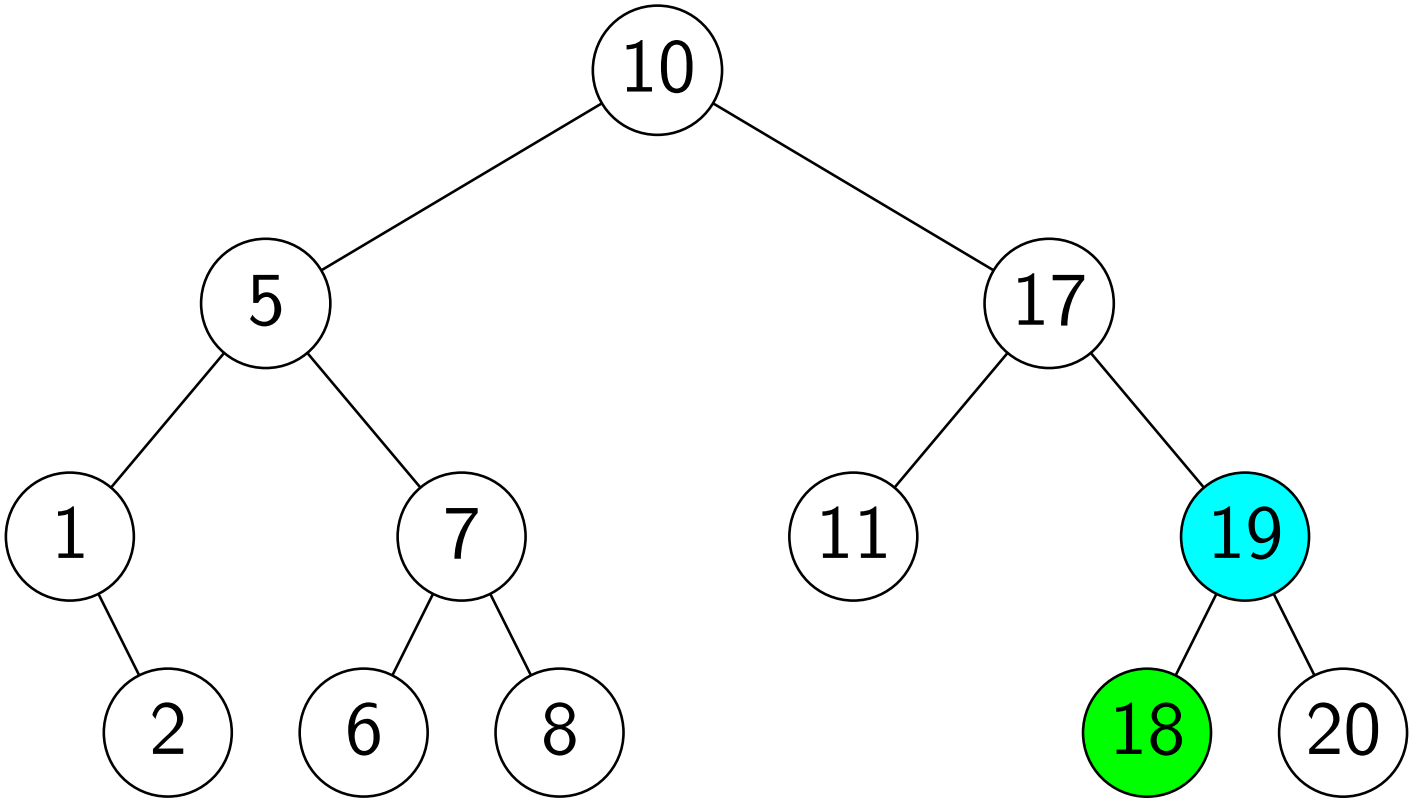
- Since $18 < 19$, the search goes left.
- Since 19 has no left child, the search terminates unsuccessfully.

BST Insert (1)



- The INSERT procedure starts by searching for the correct location to insert the provided value.
- To insert the value 18, the search reaches the state above.

BST Insert (2)



- A new node is then created and inserted into the correct position.

BST Insert (3)

```
1:  procedure BSTInsert(node, new_value)
2:      if node = null then
3:          new_node ← New node containing new_value.
4:          return new_node
5:      end if
6:      if node.value ≥ k then
7:          node.left ← BSTInsert(node.left, new_value)
8:      else if node.value < k then
9:          node.right ←BSTInsert(node.right, new_value)
10:     end if
11:     return node
12: end procedure
```

- Since the insertion itself is $\Theta(1)$, the BSTInsert algorithm requires $\Theta(h)$ time.

BST Insert (4)

```
1:  procedure BSTInsert(node, new_value)
2:      if node = null then
3:          new_node ← New node containing new_value.
4:          return new_node
5:      end if
6:      if node.value ≥ k then
7:          node.left ← BSTInsert(node.left, new_value)
8:      else if node.value < k then
9:          node.right ←BSTInsert(node.right, new_value)
10:     end if
11:     return node
12: end procedure
```

- **Question**: What is the maximum height of a binary search tree on n nodes?

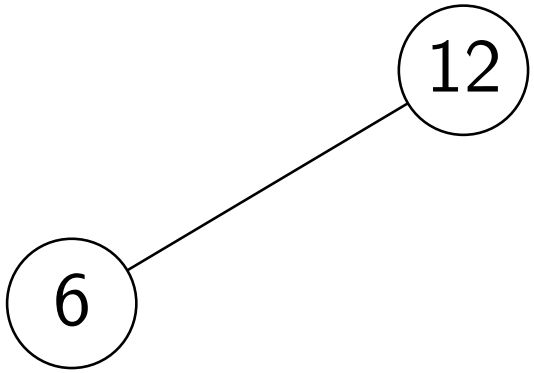
BST Insert (5)

- **Exercise**: Insert the sequence $12, 6, 10, 11$ into an initially empty binary search tree.

BST Insert (6)

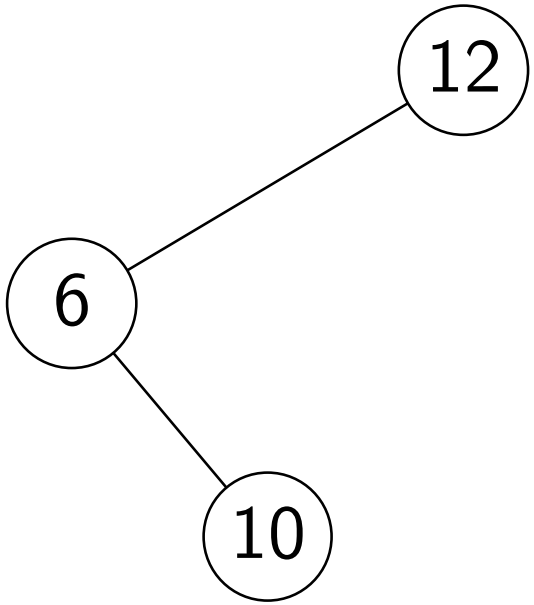
- **Exercise**: Insert the sequence $12, 6, 10, 11$ into an initially empty binary search tree.

BST Insert (7)



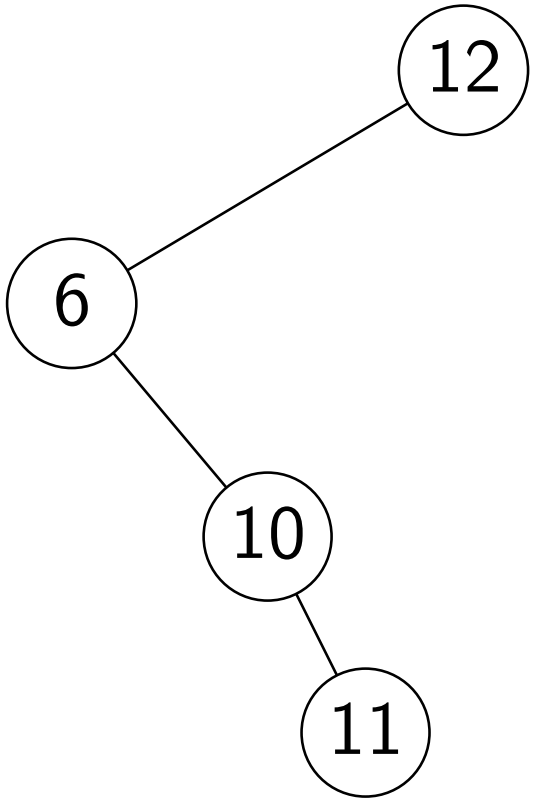
- **Exercise**: Insert the sequence $12, 6, 10, 11$ into an initially empty binary search tree.

BST Insert (8)



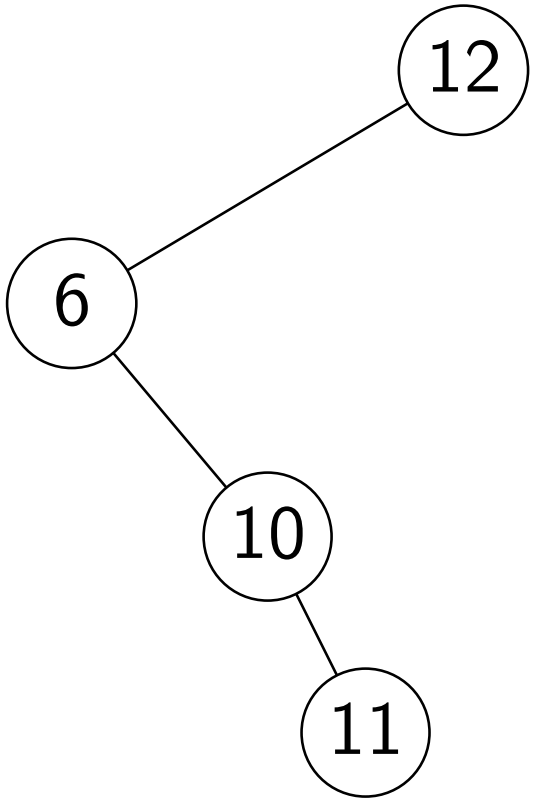
- **Exercise**: Insert the sequence $12, 6, 10, 11$ into an initially empty binary search tree.

BST Insert (9)



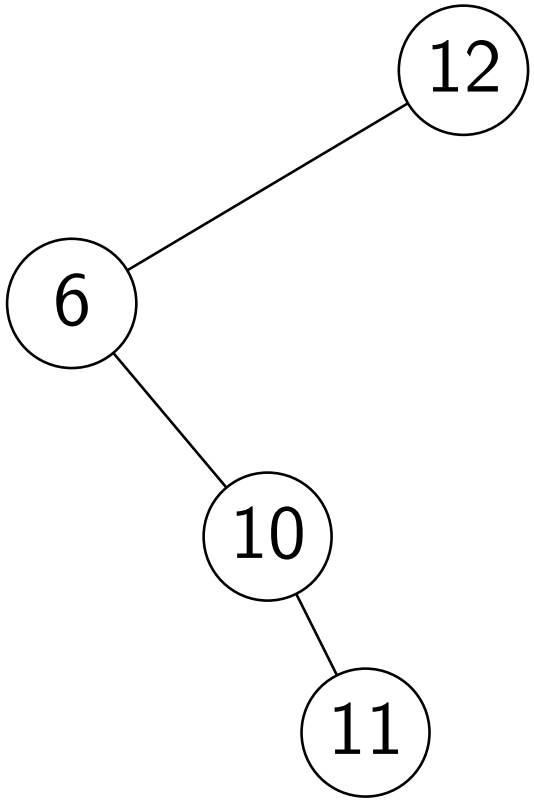
- **Exercise**: Insert the sequence $12, 6, 10, 11$ into an initially empty binary search tree.

BST Insert (10)



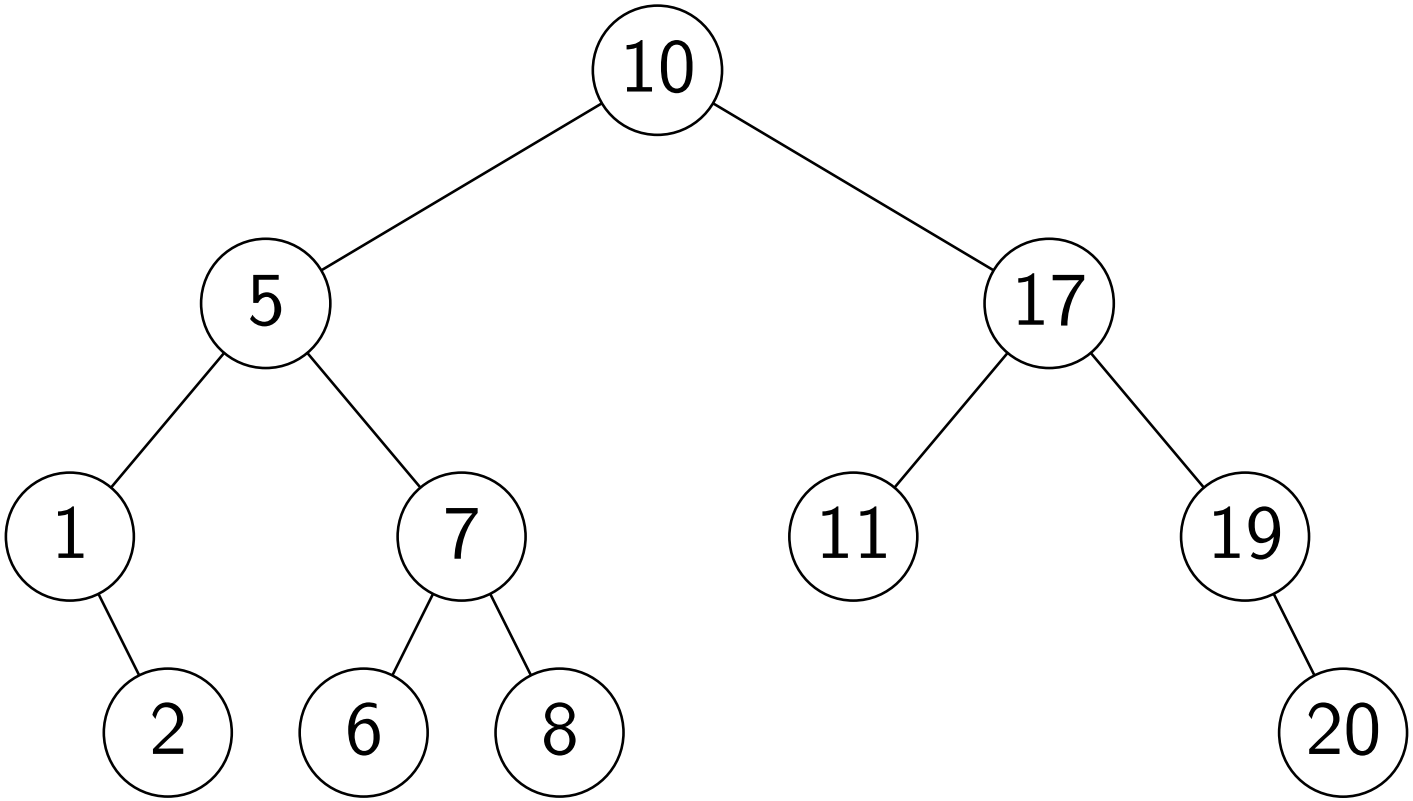
- The resulting tree contains 4 nodes and has height 3.
- In general, the height of a binary search tree is $\Theta(n)$ in the worst case.

BST Insert (11)



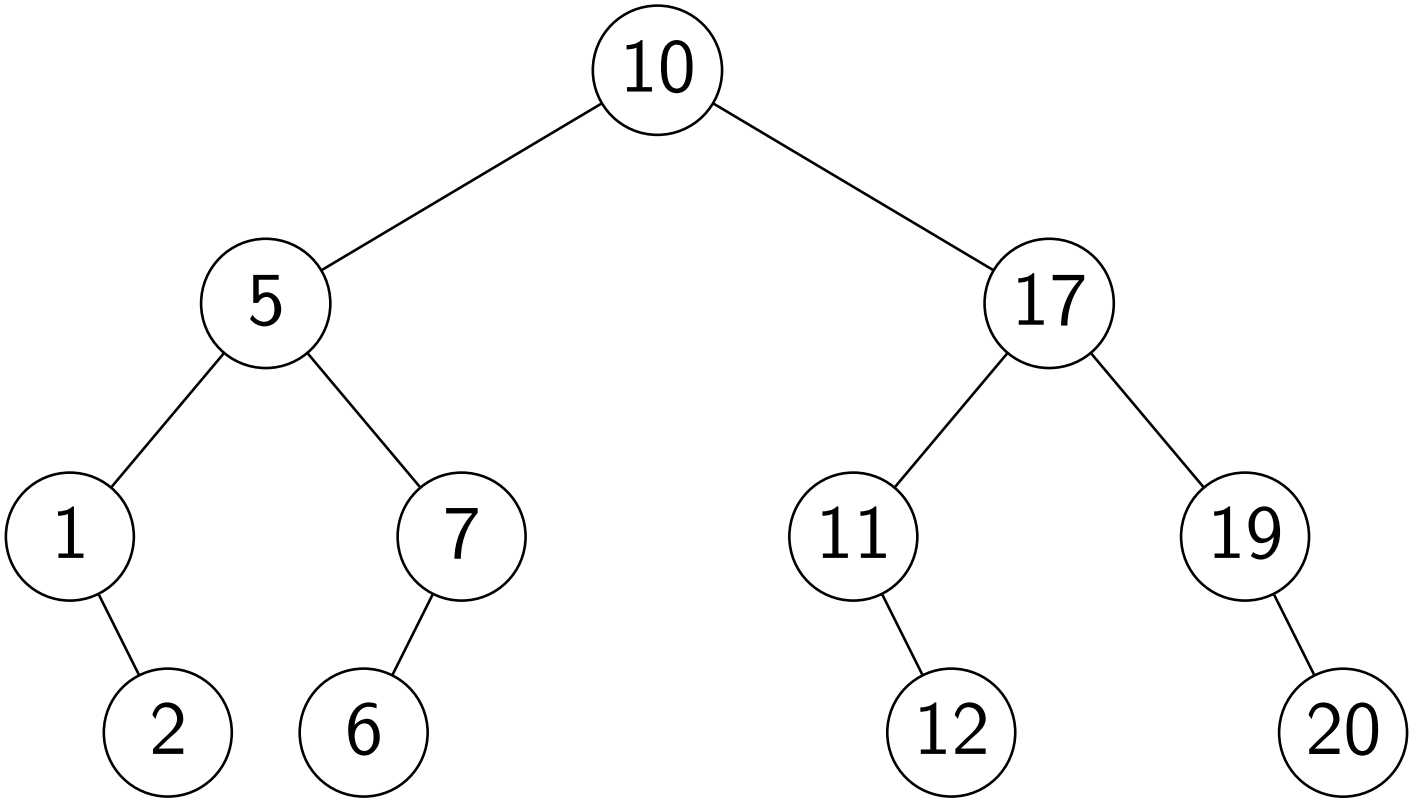
- As a result, the FIND and INSERT operations require $\Theta(n)$ time in the worst case.

BST Insert (12)



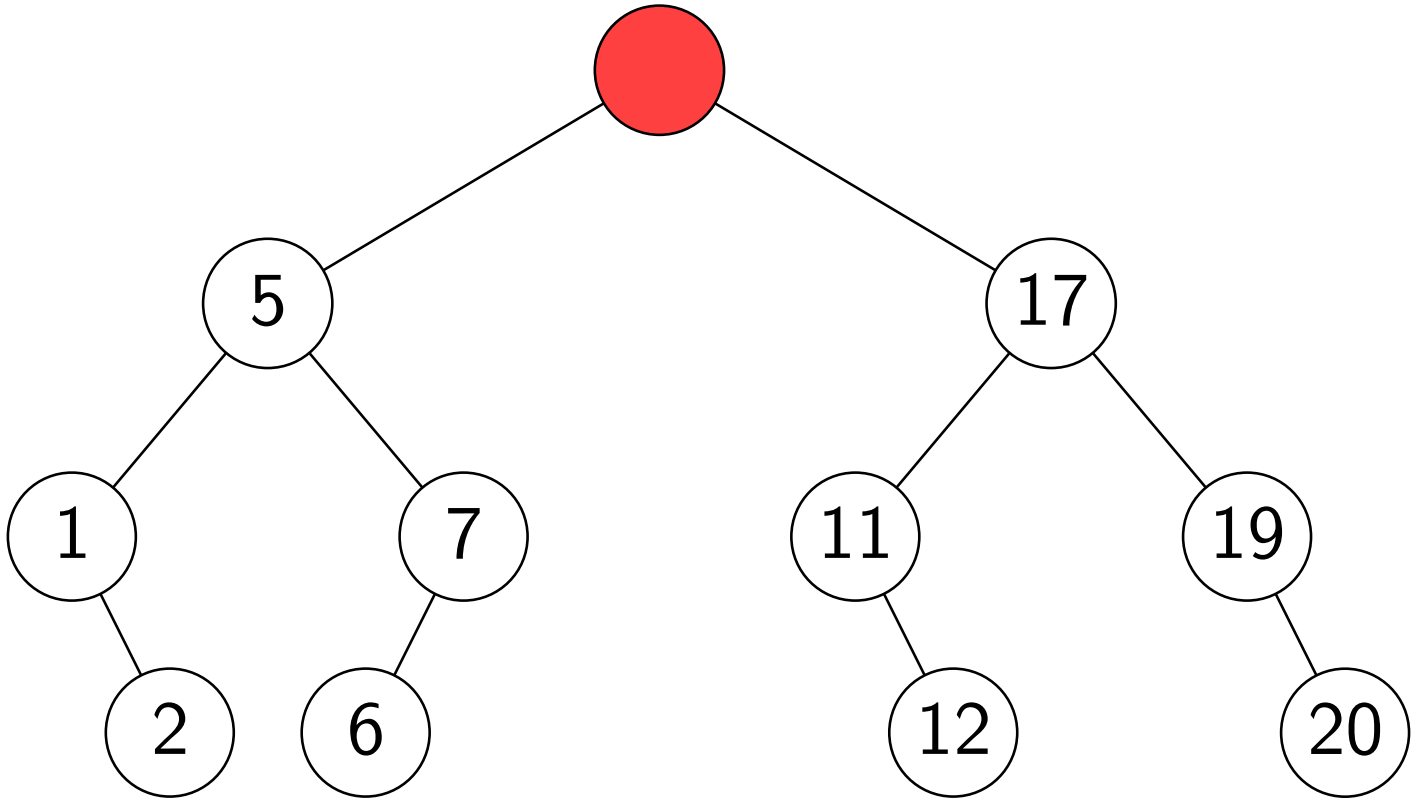
- The expected case running time of FIND and INSERT is $\Theta(\log n)$, since most sequences produce a relatively balanced tree.

BST Remove (1)



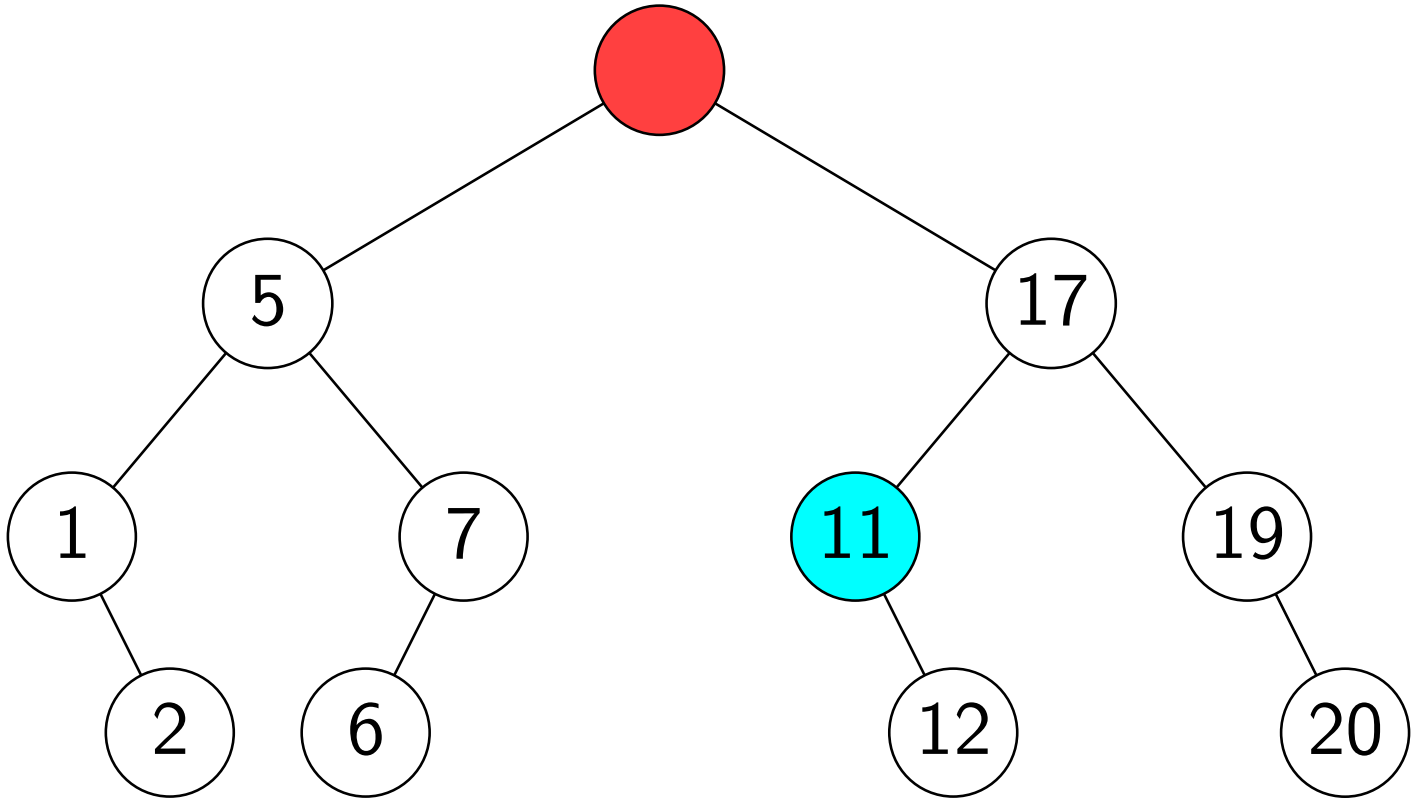
- **Exercise**: Delete the key 10 from the binary search tree above.

BST Remove (2)



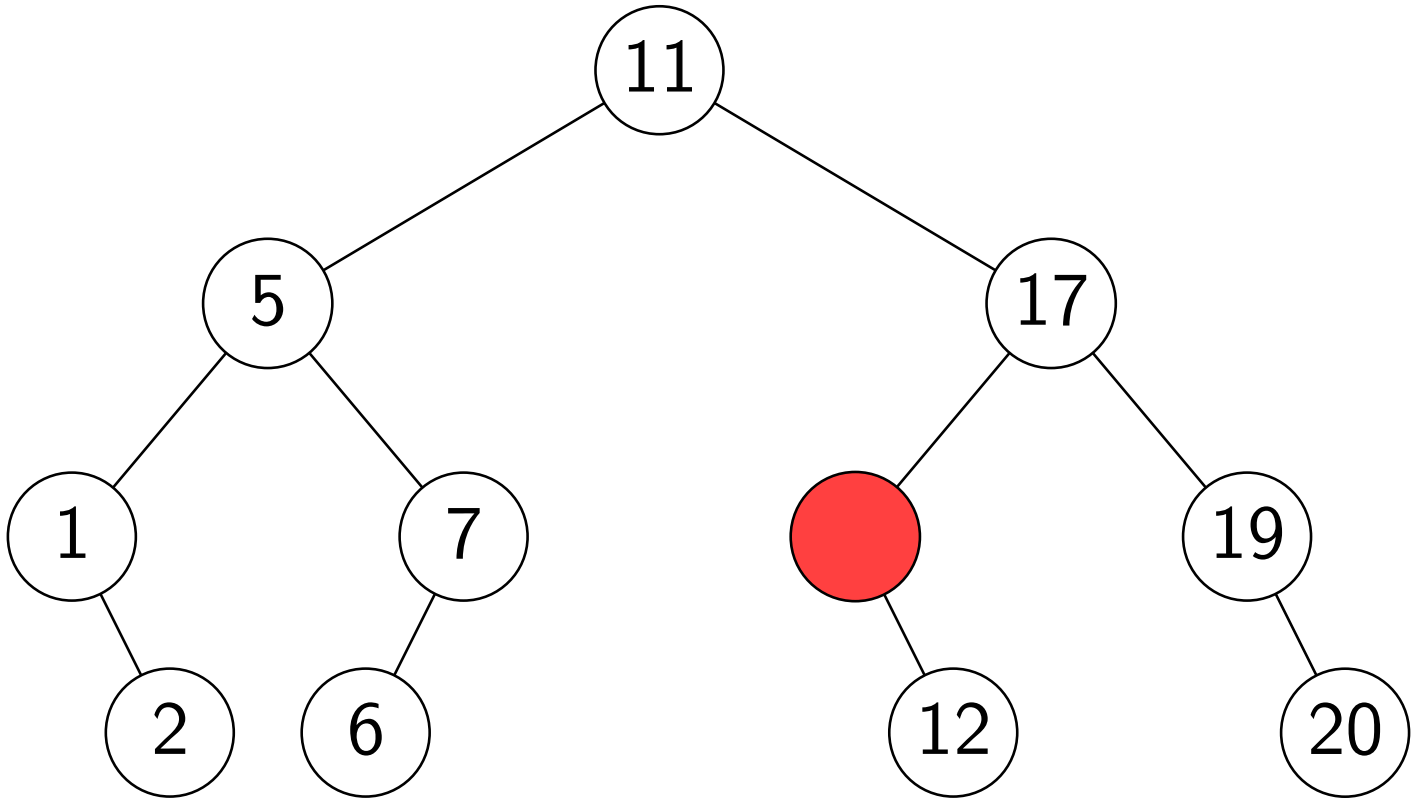
- The deletion will leave a 'hole' in the tree which must be filled by a new element.

BST Remove (3)



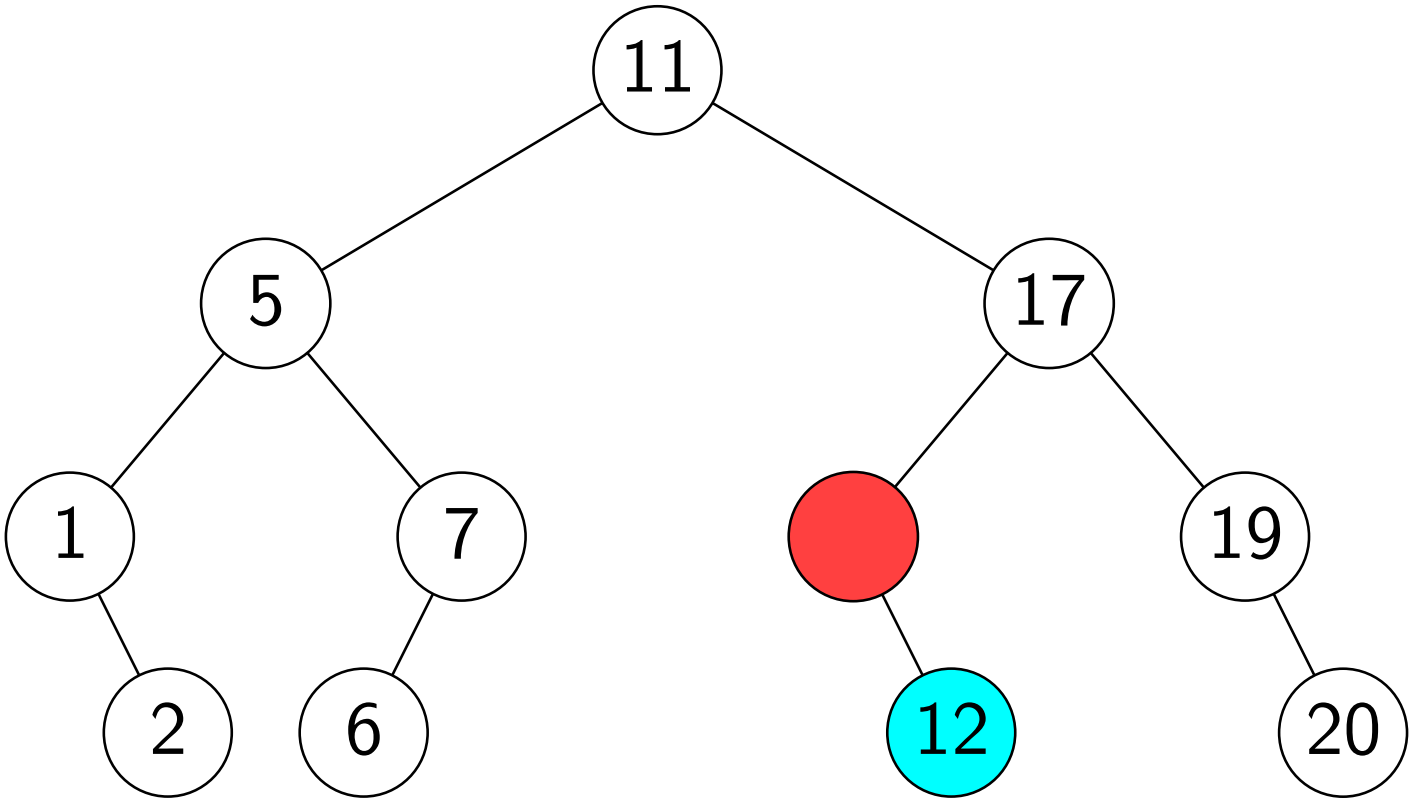
- The element which replaces the root must preserve the binary search tree.
- The only elements which qualify are the next element (or previous element) in an in-order traversal.

BST Remove (4)



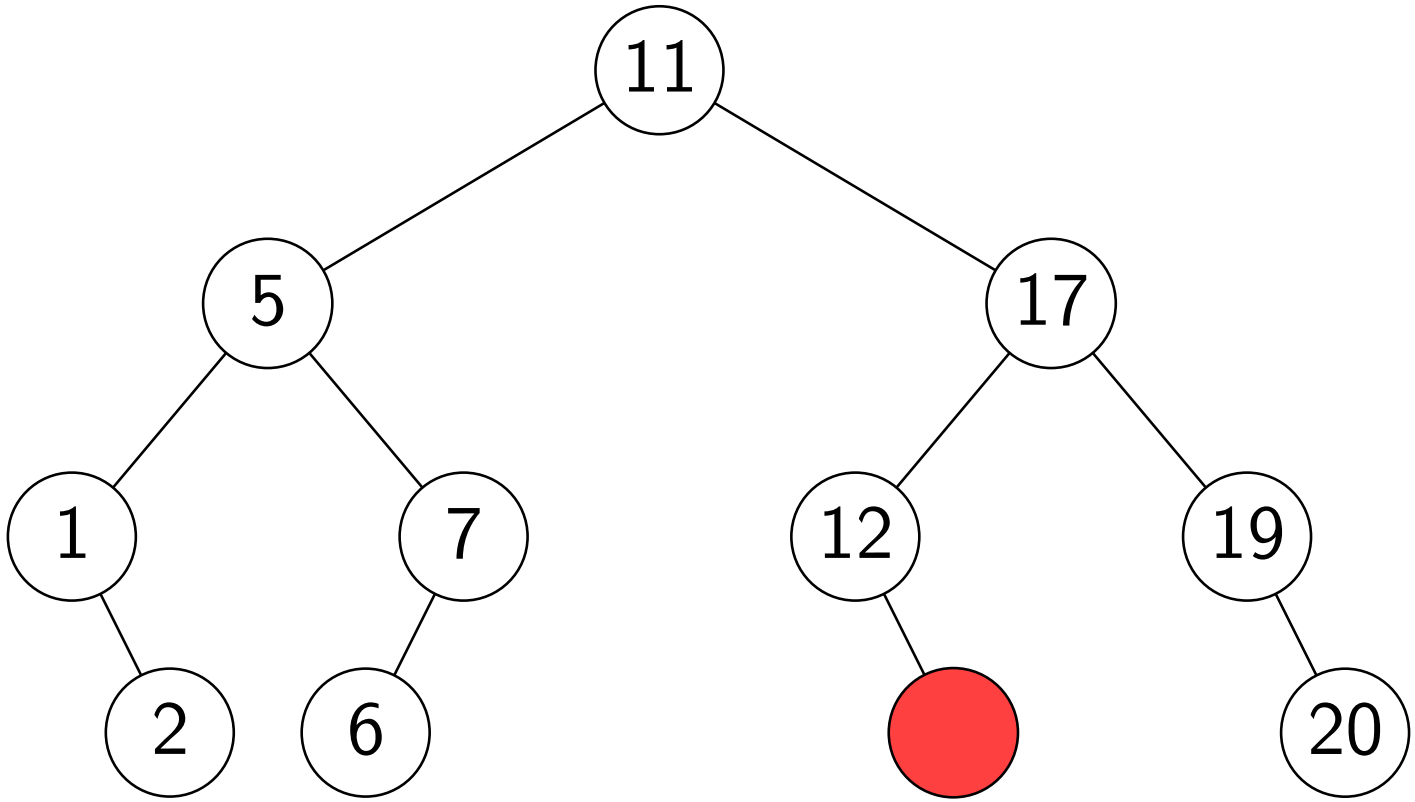
- After the replacement element is moved, a new hole is created.

BST Remove (5)



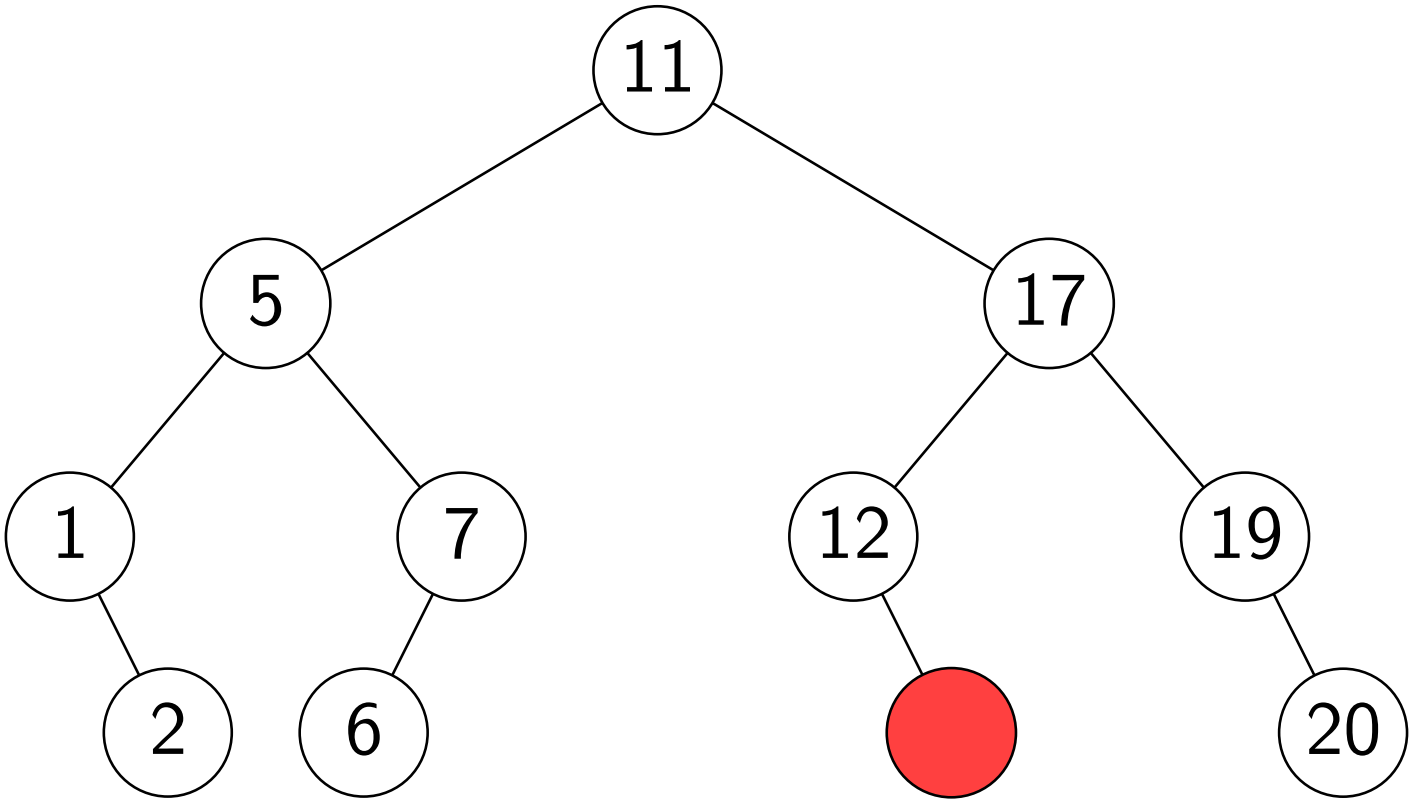
- Filling in the new hole is equivalent to recursively removing node 11.

BST Remove (6)



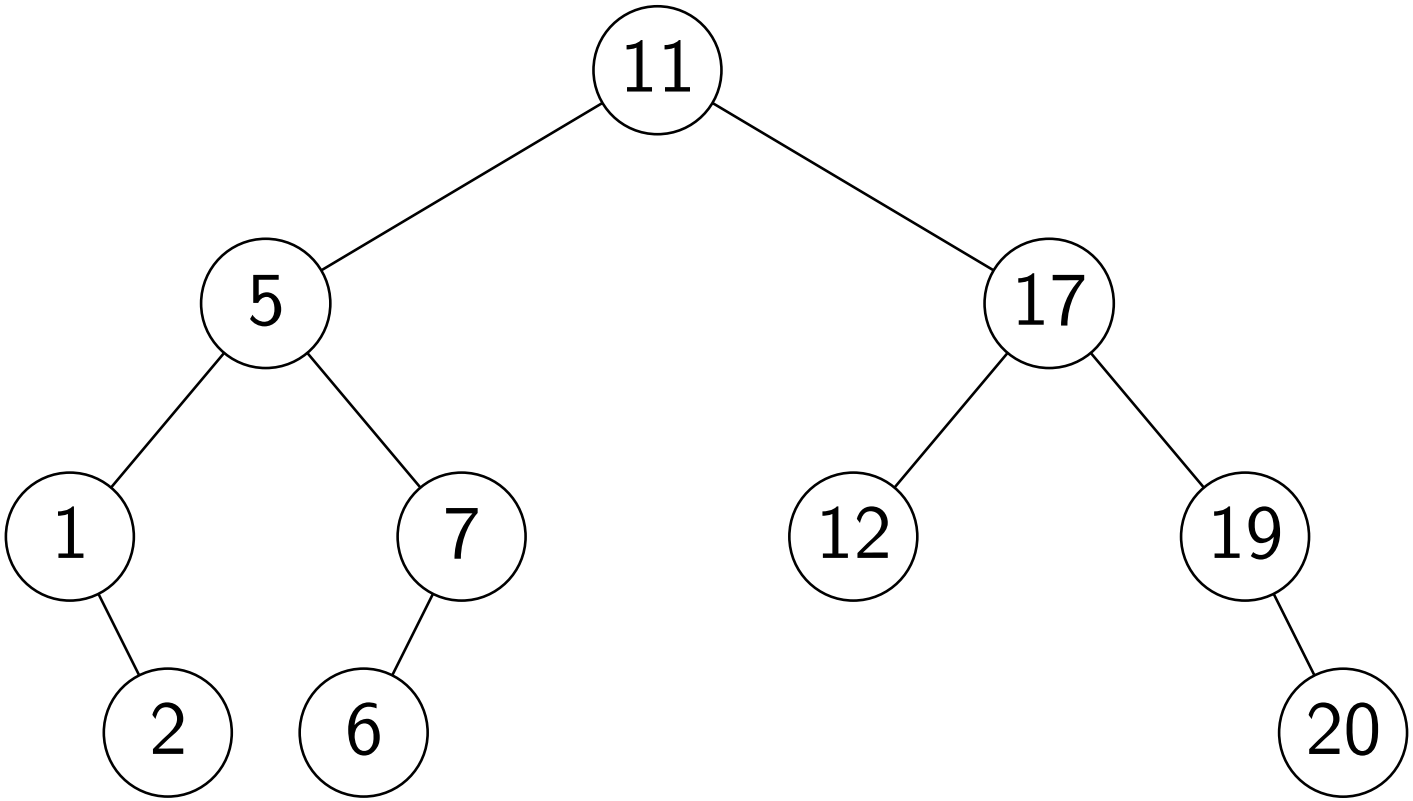
- The element 12 is moved to fill the hole created by 11 and the removal continues recursively.

BST Remove (7)



- Eventually, the removal process will end at a leaf, which can be deleted without negative consequences.

BST Remove (8)



- Eventually, the removal process will end at a leaf, which can be deleted without negative consequences.

Self-Balancing trees

To guarantee that both FIND and INSERT are $\Theta(\log n)$, it is necessary to ensure that the height of the tree is $\Theta(\log n)$. Some variants of the basic binary search tree, such as AVL trees and red/black trees, add a 'rebalancing' step to the INSERT method (after the normal binary search tree insertion) which maintains a $\Theta(\log n)$ height.

Self-balancing trees are covered in CSC 226.

2-3 Trees (1)

The 'rebalancing' approach used by AVL trees and red/black trees adjusts the tree after an insertion to compensate for any imbalances created by adding a node.

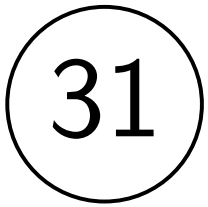
Odd Thought: What if, instead of rebalancing the tree after adding a node, we never add nodes at all? Instead, just cram new values into existing nodes, and split a node into multiple nodes if it gets too full.

Using nodes with maximum capacity 2 results in a data structure called a **2-3 tree**.

2-3 Trees (2)

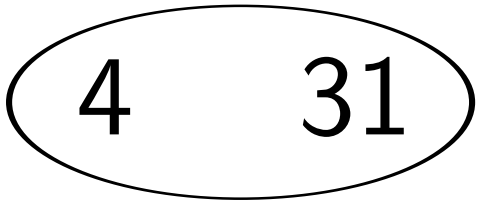
- **Example**: Insert the sequence $31, 4, 15, 9, 2, 6, 5, 3$ into an initially empty 2-3 tree.

2-3 Trees (3)



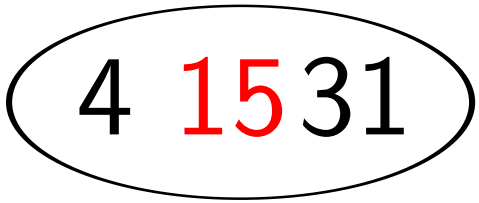
- Create a new node to insert the first element. This is the only INSERT operation that directly creates a node.

2-3 Trees (4)



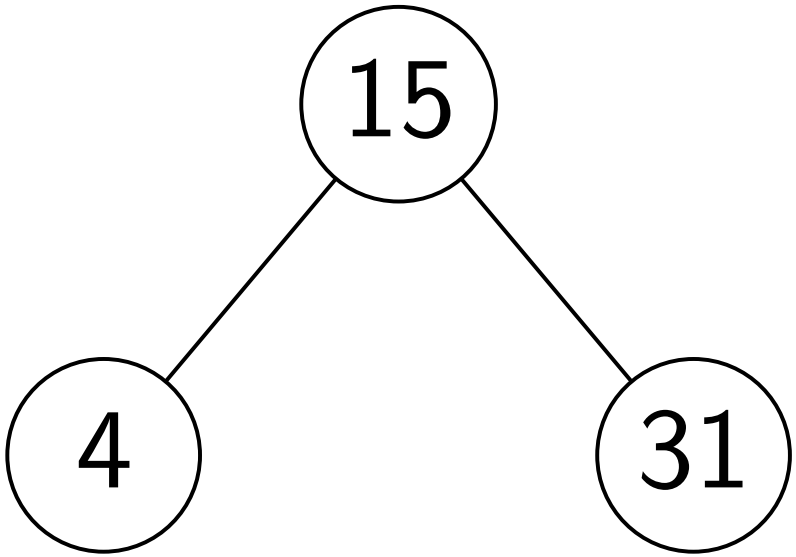
- To insert the second element, add it to the root node (which can contain up to 2 elements).

2-3 Trees (5)



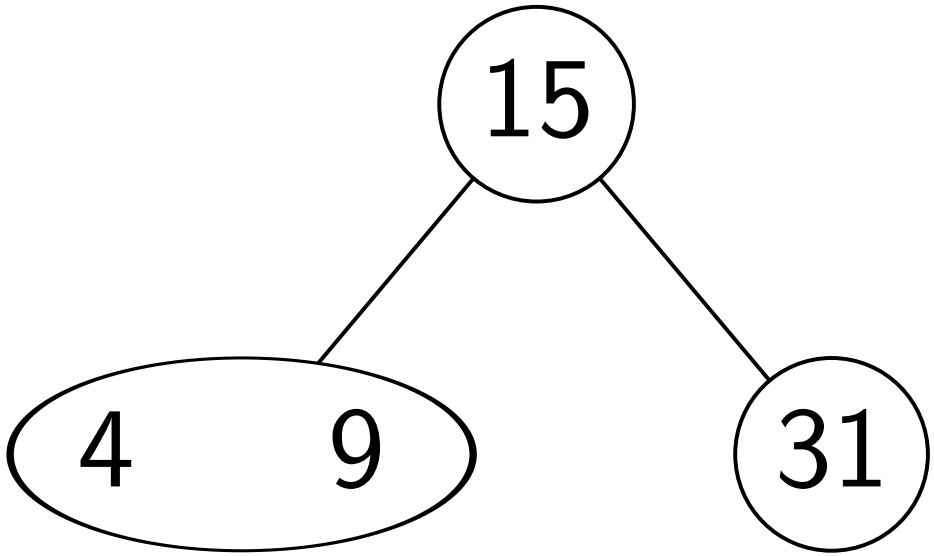
- To insert 15, we first put it in the root node.
- However, the root is now too full, so it is split.

2-3 Trees (6)



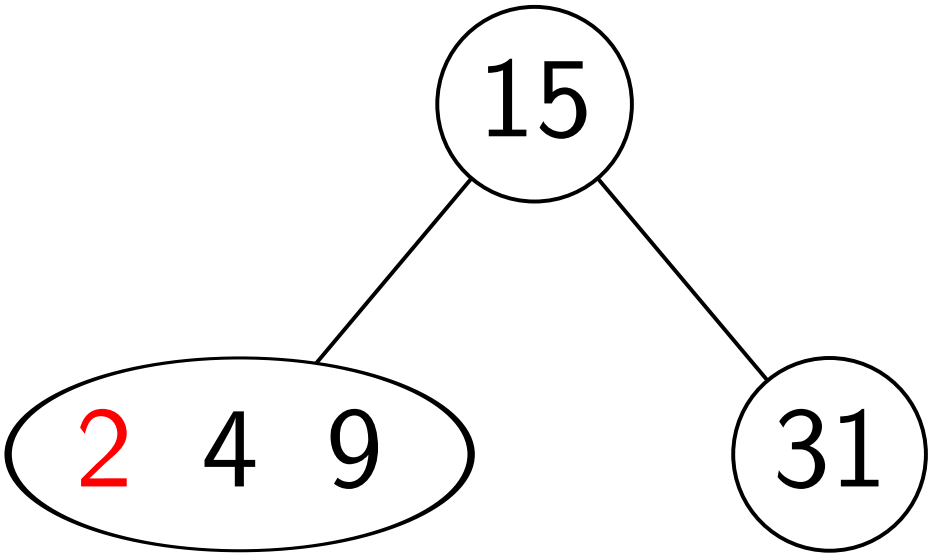
- Splitting an over-full node produces two single nodes (for the smallest and largest elements in the over-full node) and elevates the middle element to the level above (which creates a new root).

2-3 Trees (7)



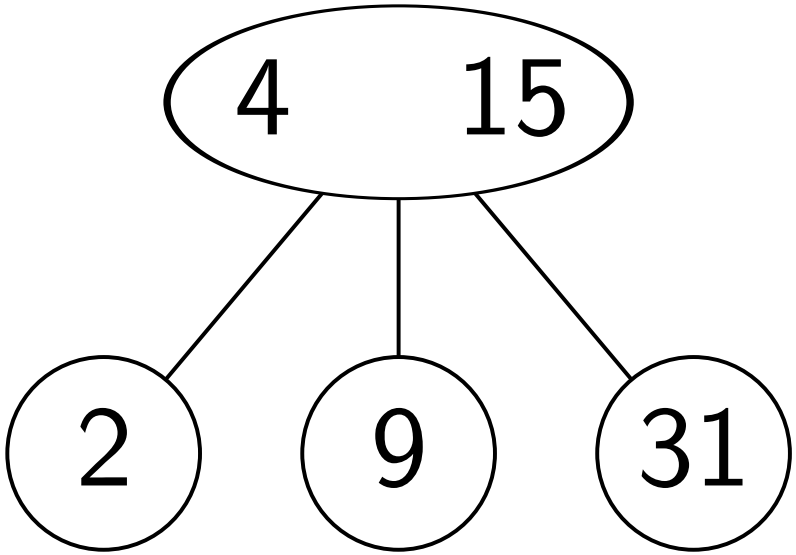
- To insert 9, use the FIND algorithm until the search terminates at a leaf.
- The element 9 is added to the node containing the 4.

2-3 Trees (8)



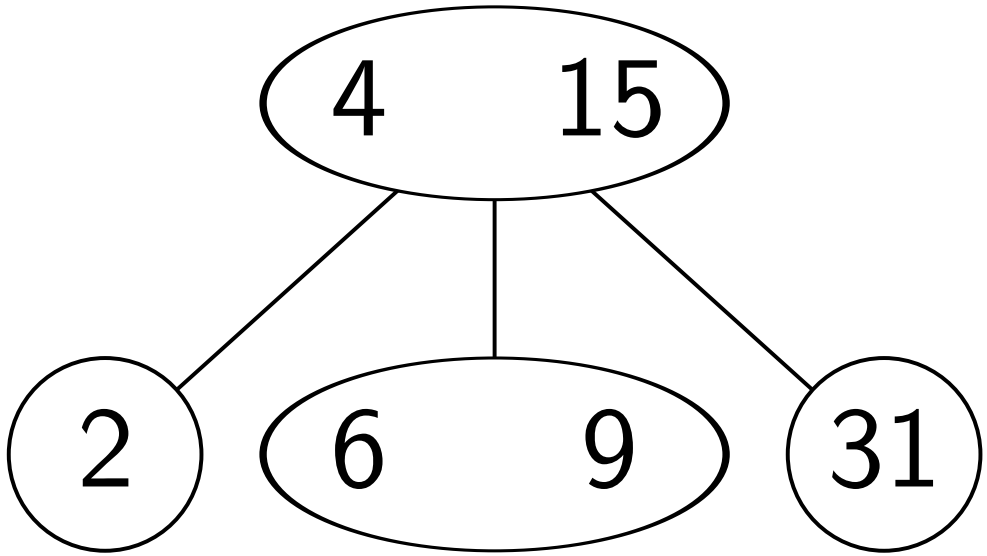
- Inserting 2 into the node with 4 and 9 makes the node too full, so it is split into three pieces: The 2 and 9 become single nodes, and the 4 is pushed upwards into the parent node.

2-3 Trees (9)



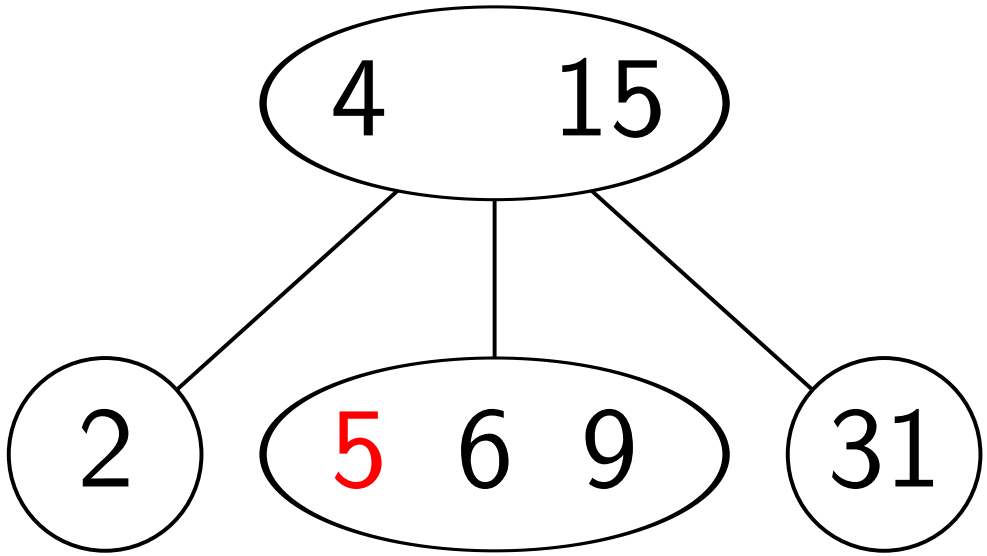
- A node with 2 keys a and b can have 3 children, corresponding to the elements less than a and b, elements between a and b, and elements greater than a and b.

2-3 Trees (10)



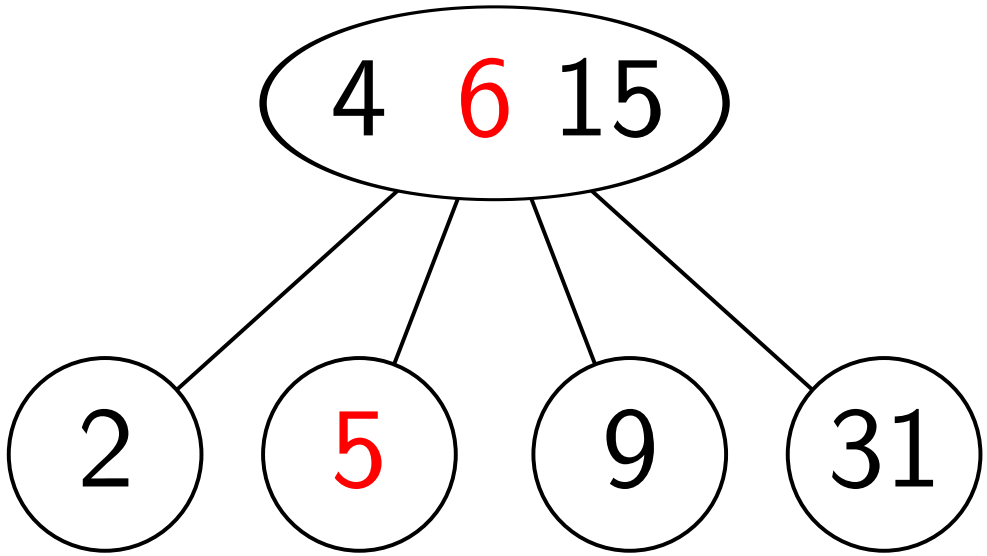
- The element 6 is inserted into the node with 9.

2-3 Trees (11)



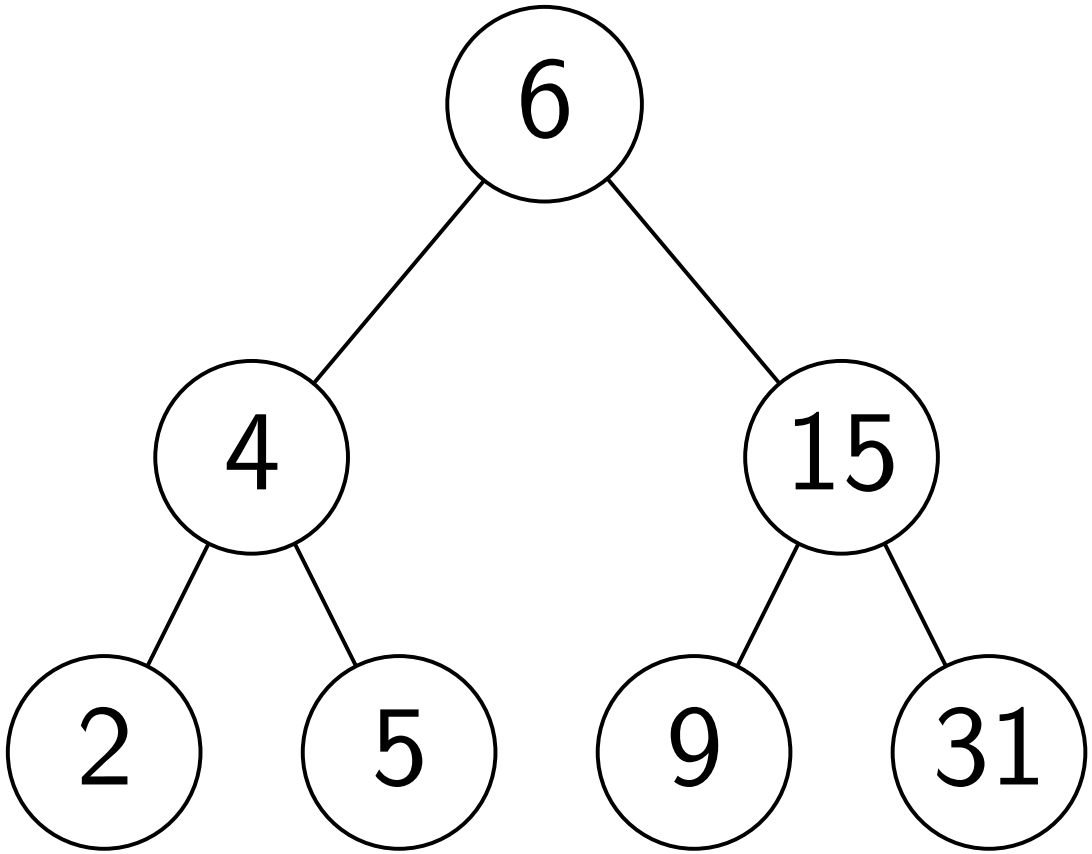
- The element 5 is inserted into the node with 6 and 9.
- The resulting node is too full, so it is split, producing single nodes for 5 and 9, and pushing the 6 upwards.

2-3 Trees (12)



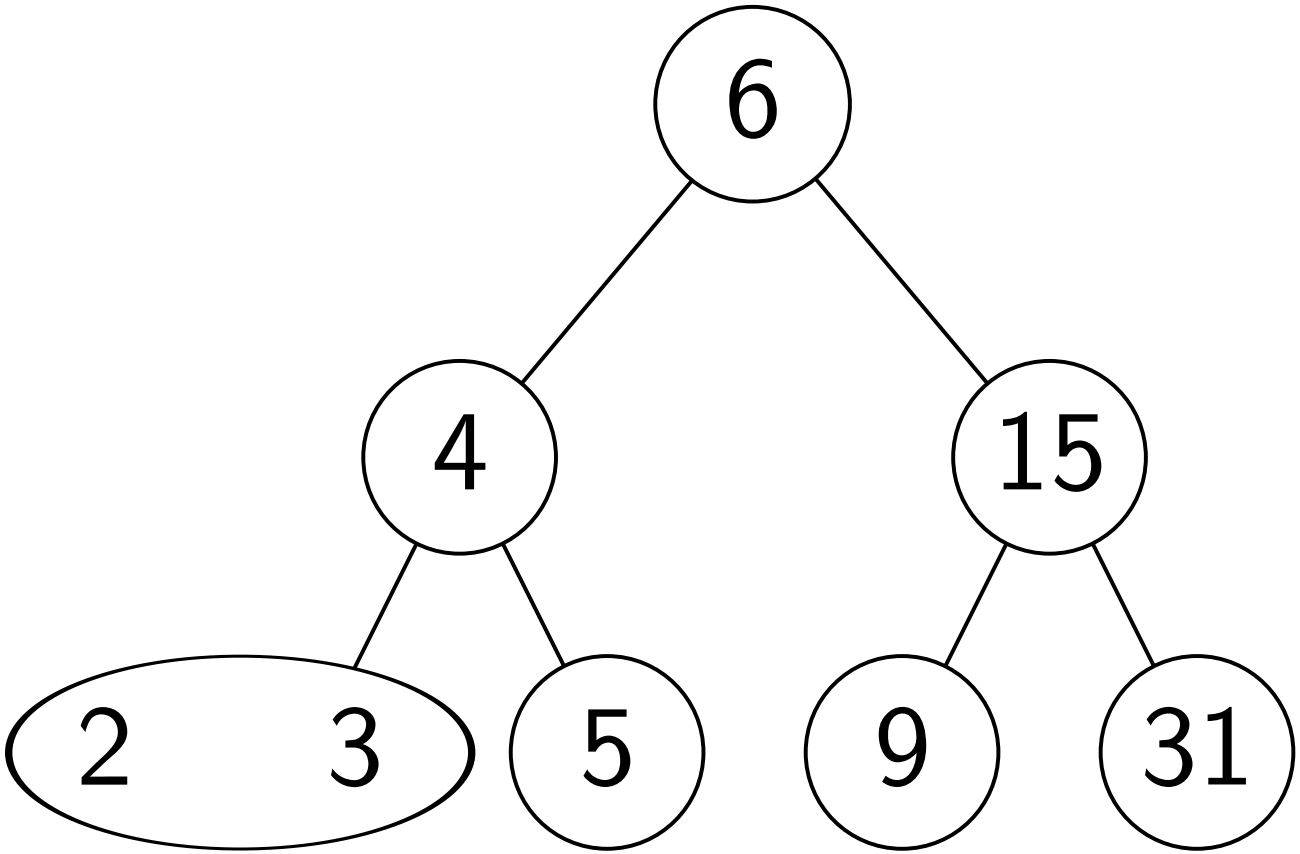
- Now the root is too full, so it is split, producing single nodes for 4 and 15 (which each take two of the child nodes), and pushing the 6 further upwards.

2-3 Trees (13)



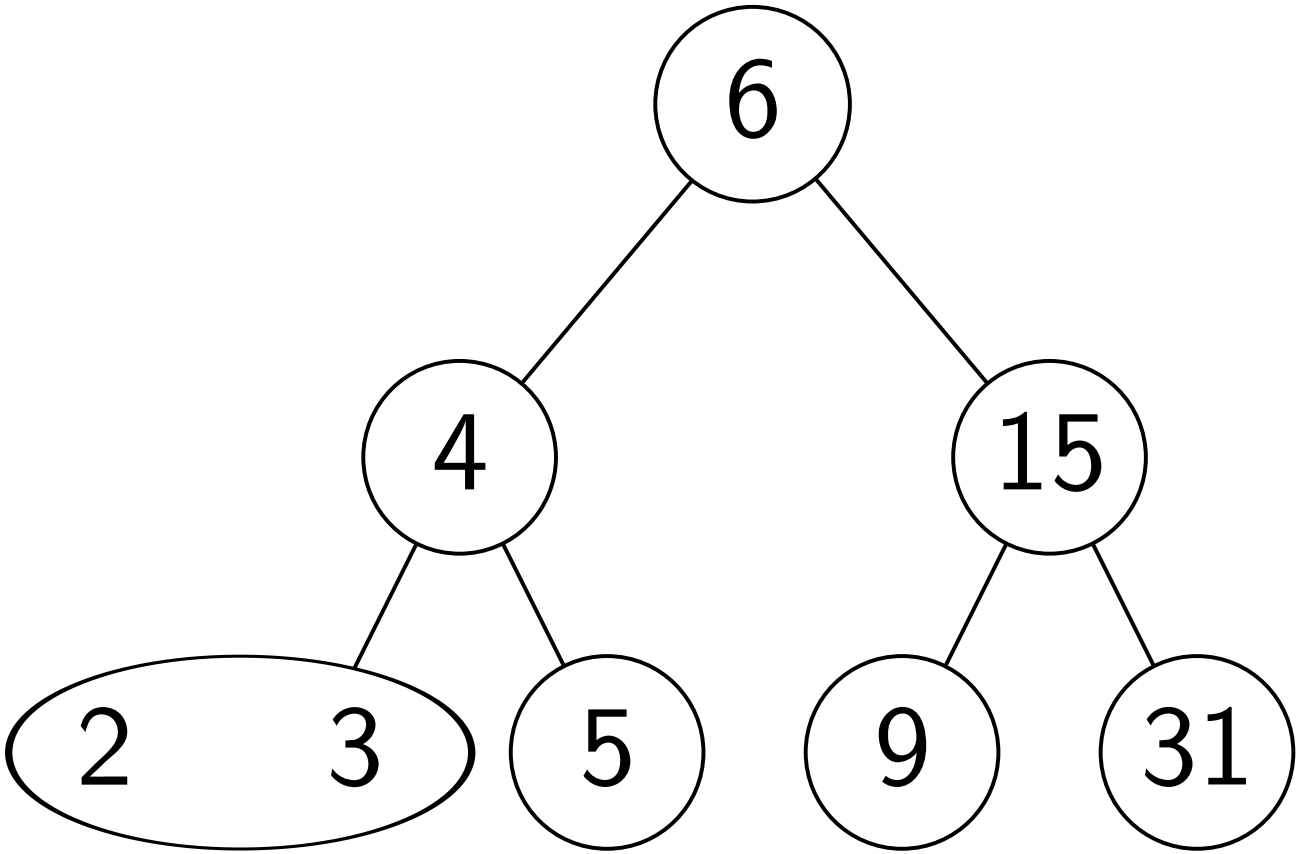
- The resulting tree contains 6 at the root.
- Every level of a 2-3 tree is always full, and there are never any gaps.

2-3 Trees (14)



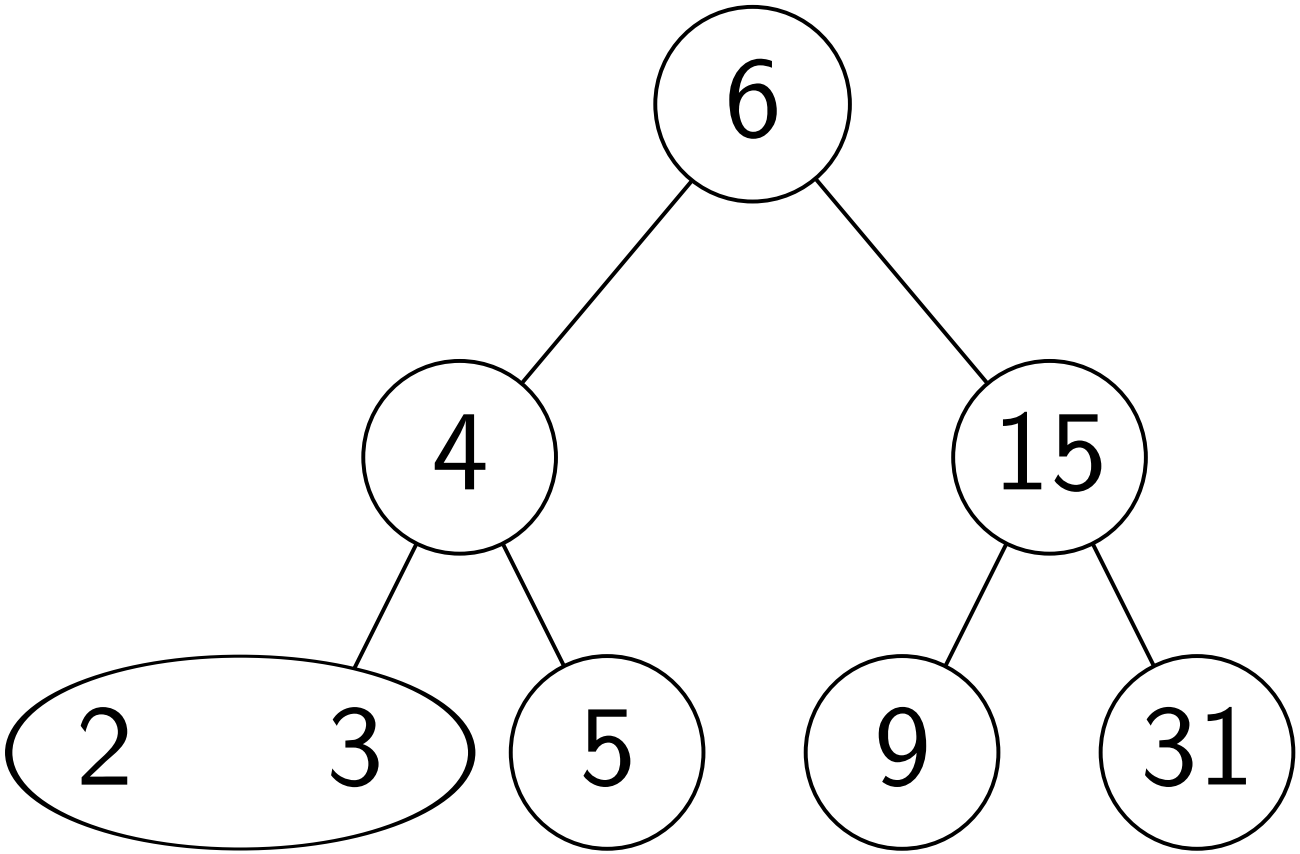
- The element 3 is inserted into the node containing 2.

2-3 Trees (15)



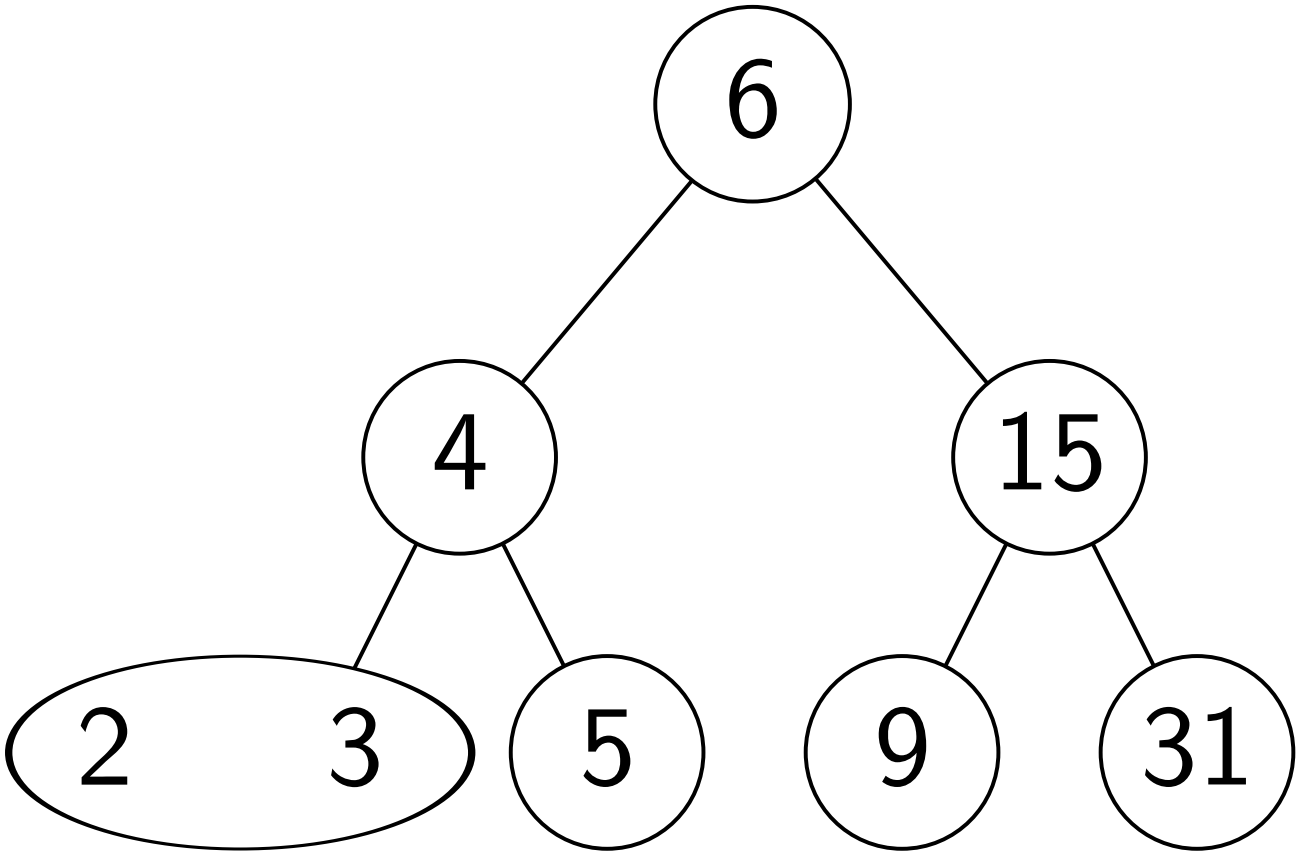
- The height of a 2-3 tree is always $\Theta(\log n)$, so FIND requires $\Theta(\log n)$ time.

2-3 Trees (16)



- The insertion process may require splitting one or more nodes, but the sequence of splits will progress through a single path from a leaf to the root, requiring $\Theta(\log n)$ time.

2-3 Trees (17)



- 2-3 trees are related to red/black trees, and are covered in greater detail in CSC 226.
- 2-3 trees are also a special case of B-trees, which are very important to database systems (covered in CSC 370).