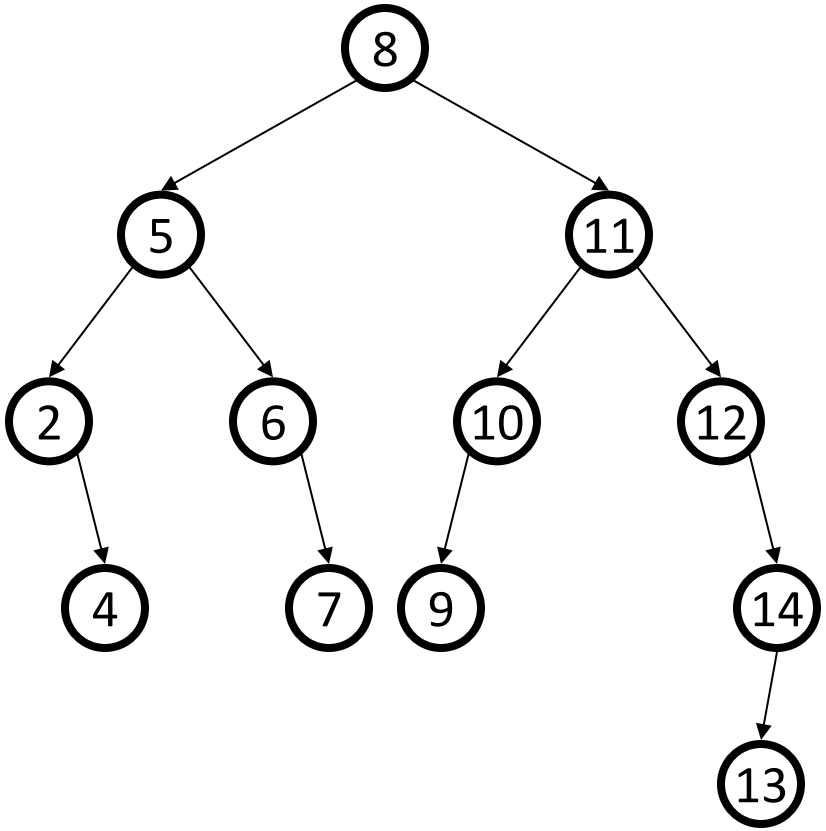
Lecture 4

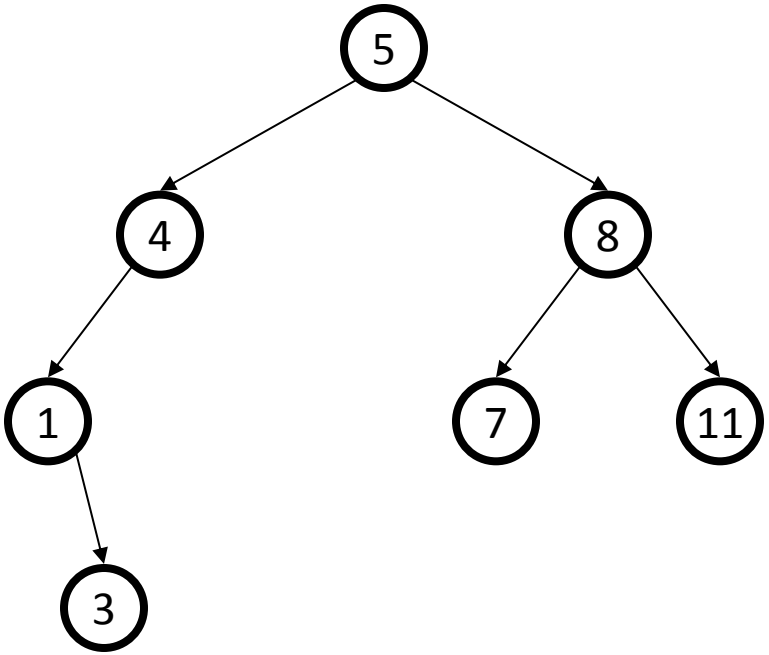
Binary Search Trees

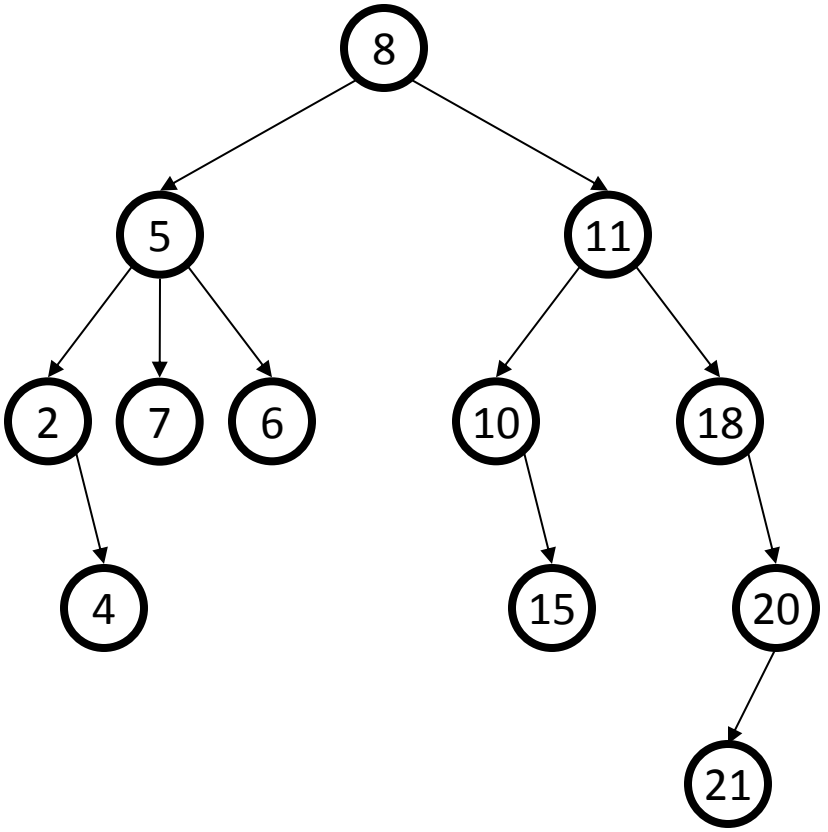
Binary Search Tree

- Structure property ("binary")
  - Each node has ≤ 2 children
  - Result: keeps operations simple
- Order property
  - All keys in left subtree smaller than node's key
  - All keys in right subtree larger than node's key
  - Result: easy to find any given key

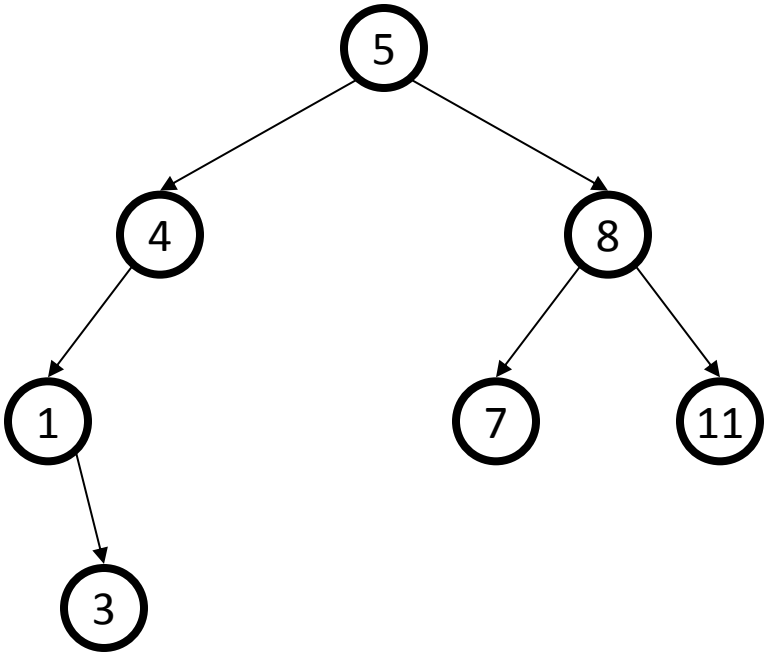


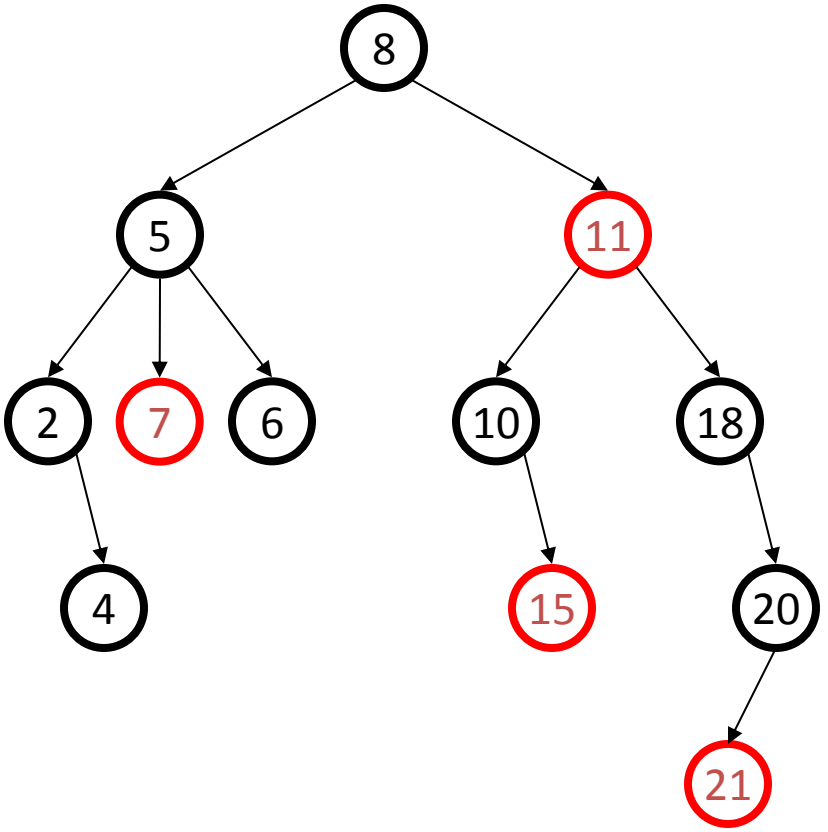
Are these BSTs?



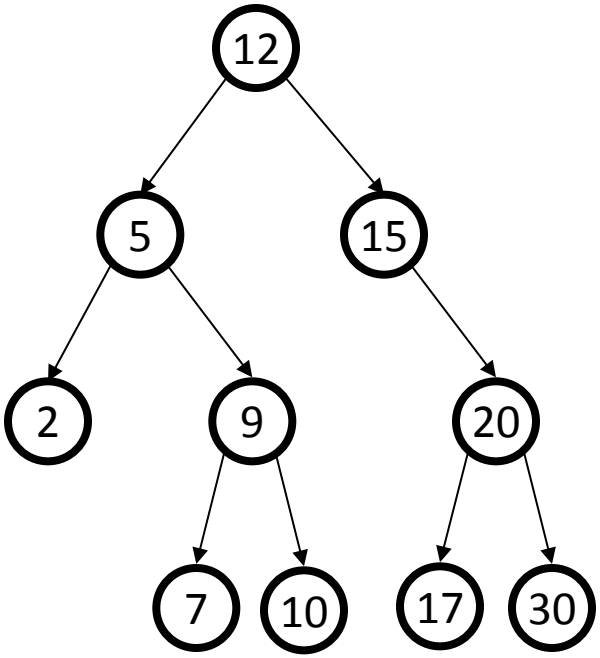


Are these BSTs?



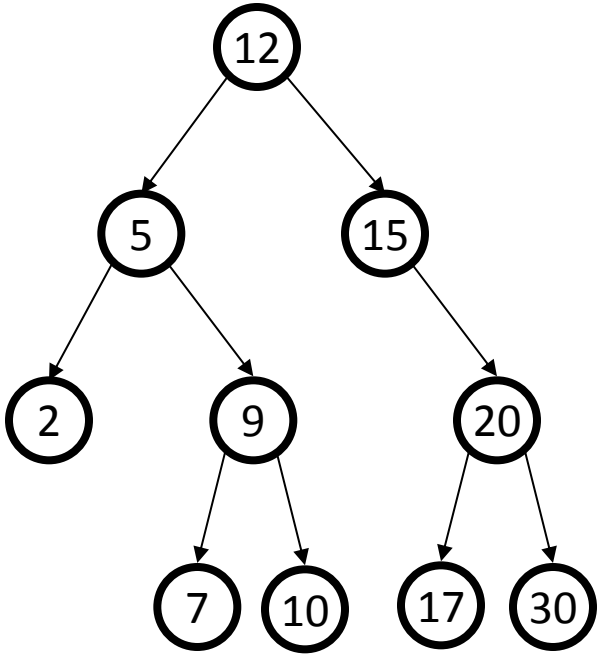


Find in BST, Recursive




```
int find(Key key, Node root){
 if(root == null)
   return null;
 if(key < root.key)
   return find(key,root.left);
 if(key > root.key)
   return find(key,root.right);
 return root.data;
}
```

Find in BST, Iterative




```
int find(Key key, Node root){
 while(root != null && root.key != key){
  if(key < root.key)
     root = root.left;
  else(key > root.key)
     root = root.right;
 }
 if(root == null)
     return null;
 return root.data;
}
```

Find in BST, complexity

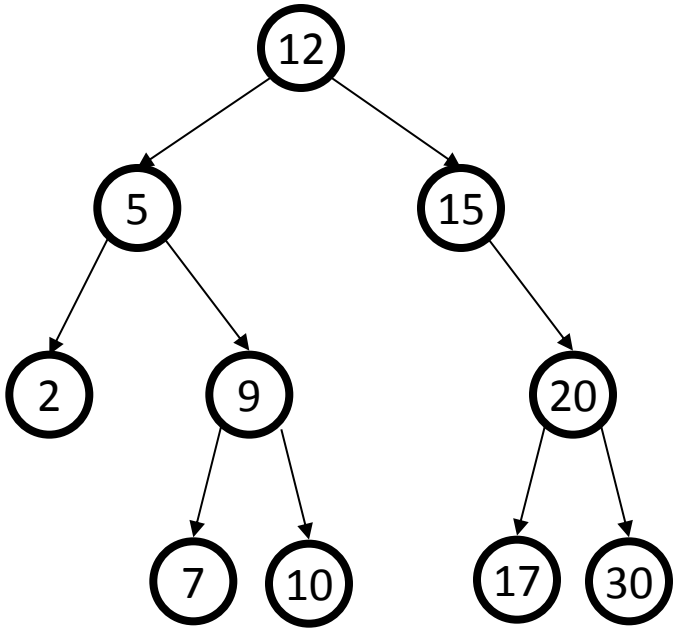
- O(h) where h is the height of the tree
  - N is the size (number of nodes) and h is the height
  - Worst case: h = n - 1
  - Best case: h = log(n)

- Average case
  - Consider all BST obtained by inserting n distinct values in the empty BST
  - Compute the average of height of all those BST
  - Result is $\Theta(\log(N))$

Other “Finding” Operations

- Find *minimum* node
- Find *maximum* node

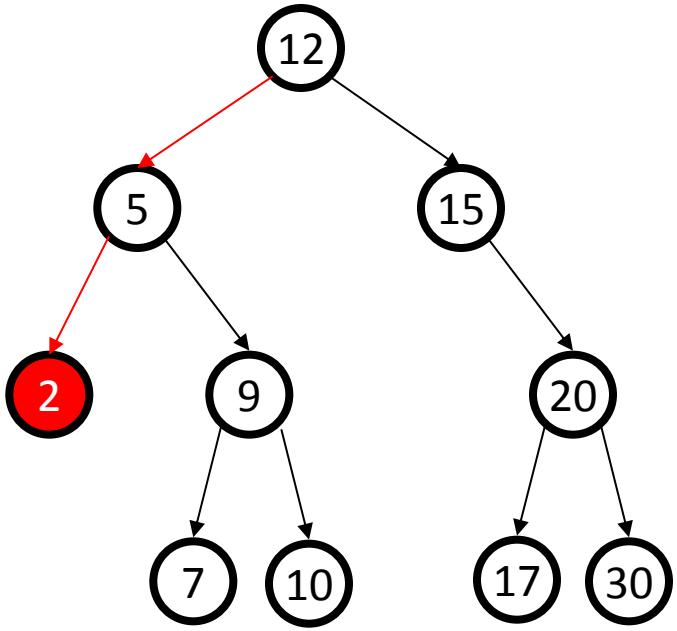
- Find *predecessor* of a non-leaf
- Find *successor* of a non-leaf
- Find *predecessor* of a leaf
- Find *successor* of a leaf



Other “Finding” Operations

- Find *minimum* node
- Find *maximum* node

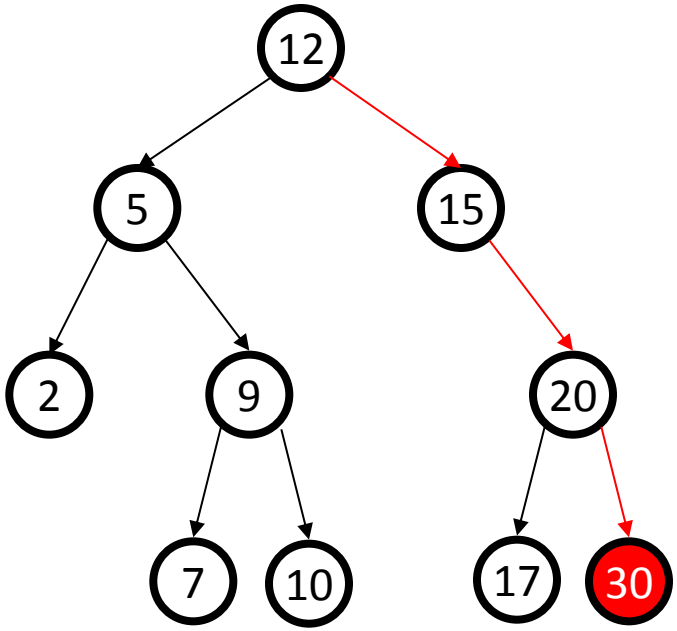
- Find *predecessor* of a non-leaf
- Find *successor* of a non-leaf
- Find *predecessor* of a leaf
- Find *successor* of a leaf



Other “Finding” Operations

- Find *minimum* node
- Find *maximum* node

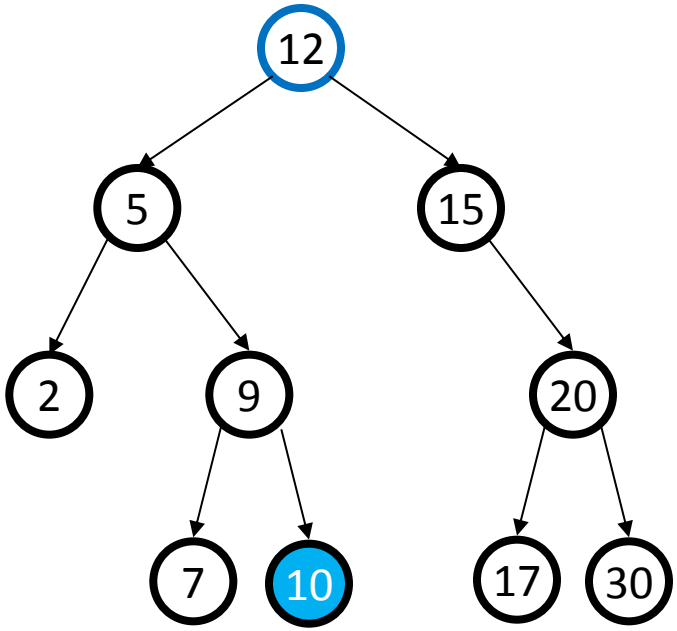
- Find *predecessor* of a non-leaf
- Find *successor* of a non-leaf
- Find *predecessor* of a leaf
- Find *successor* of a leaf



Other “Finding” Operations

- Find *minimum* node
- Find *maximum* node

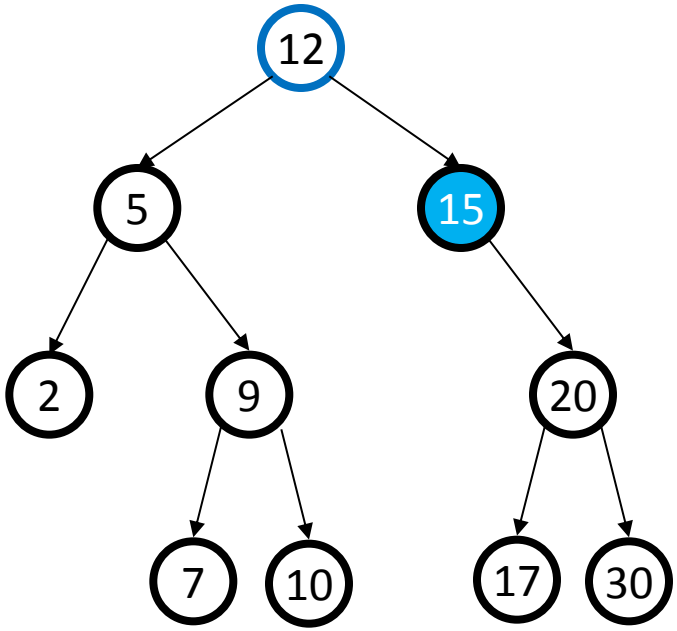
- Find *predecessor* of a non-leaf
- Find *successor* of a non-leaf
- Find *predecessor* of a leaf
- Find *successor* of a leaf



Other “Finding” Operations

- Find *minimum* node
- Find *maximum* node

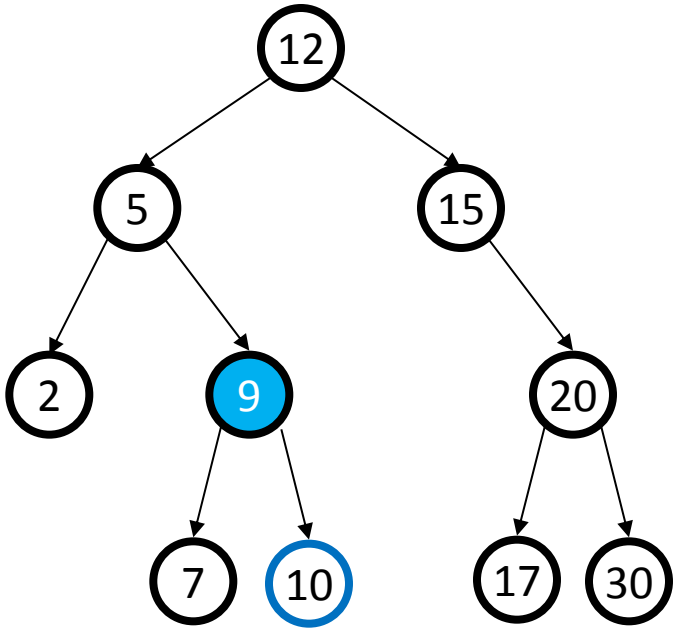
- Find *predecessor* of a non-leaf
- Find *successor* of a non-leaf
- Find *predecessor* of a leaf
- Find *successor* of a leaf



Other “Finding” Operations

- Find *minimum* node
- Find *maximum* node

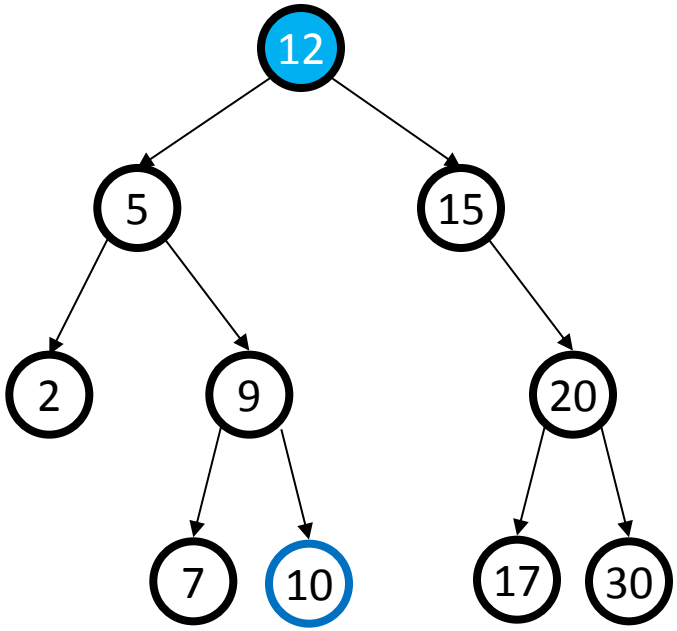
- Find *predecessor* of a non-leaf
- Find *successor* of a non-leaf
- Find *predecessor* of a leaf
- Find *successor* of a leaf



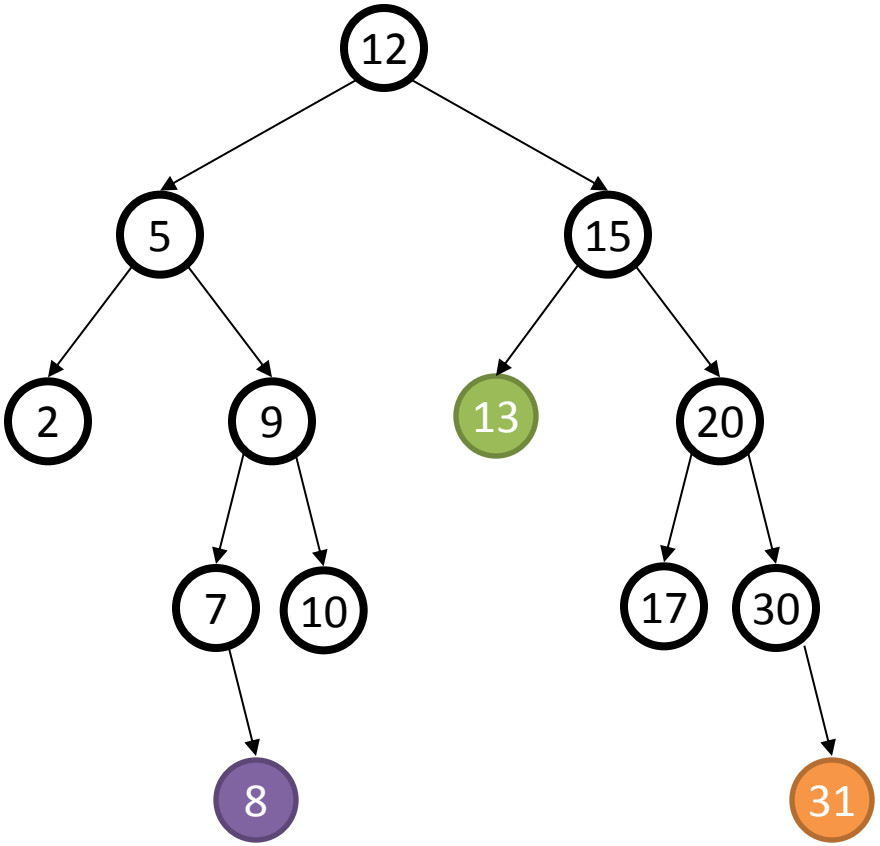
Other “Finding” Operations

- Find *minimum* node
- Find *maximum* node

- Find *predecessor* of a non-leaf
- Find *successor* of a non-leaf
- Find *predecessor* of a leaf
- Find *successor* of a leaf



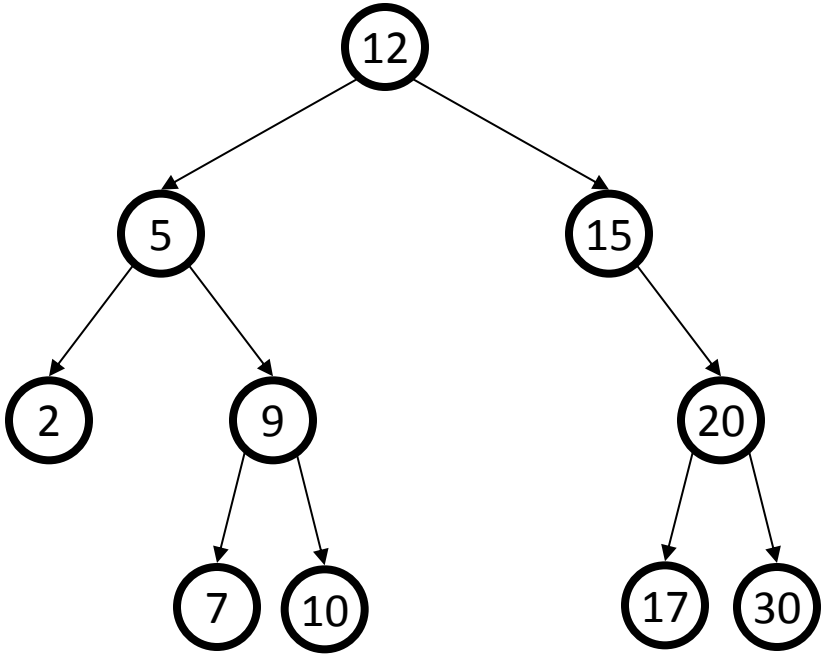
Insert in BST




```
insert(13)
insert(8)
insert(31)
```

(New) insertions happen only at leaves – easy!

Deletion in BST

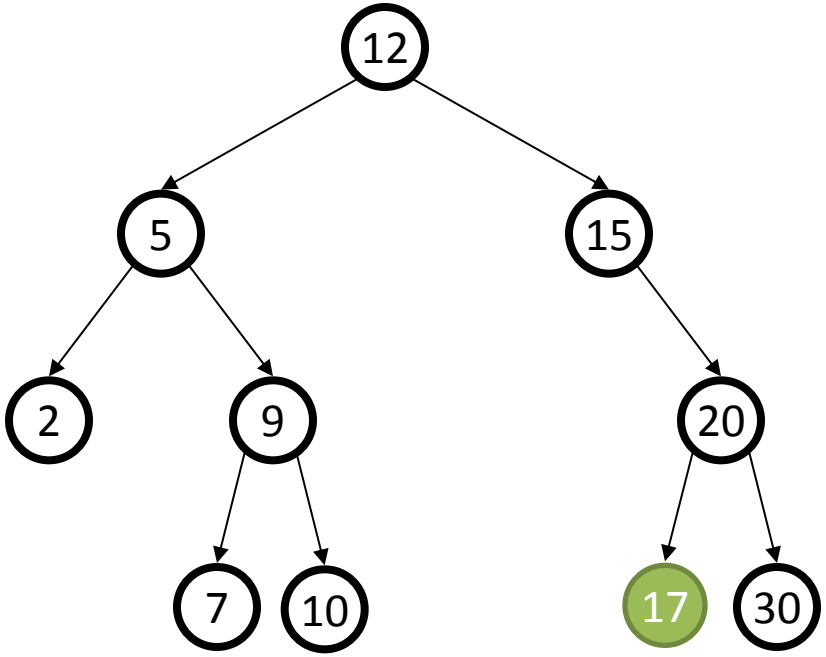


Deletion

- Removing an item disrupts the tree structure
- Basic idea: `find` the node to be removed, then "fix" the tree so that it is still a binary search tree
- Three cases:
  - Node has no children (leaf)
  - Node has one child
  - Node has two children

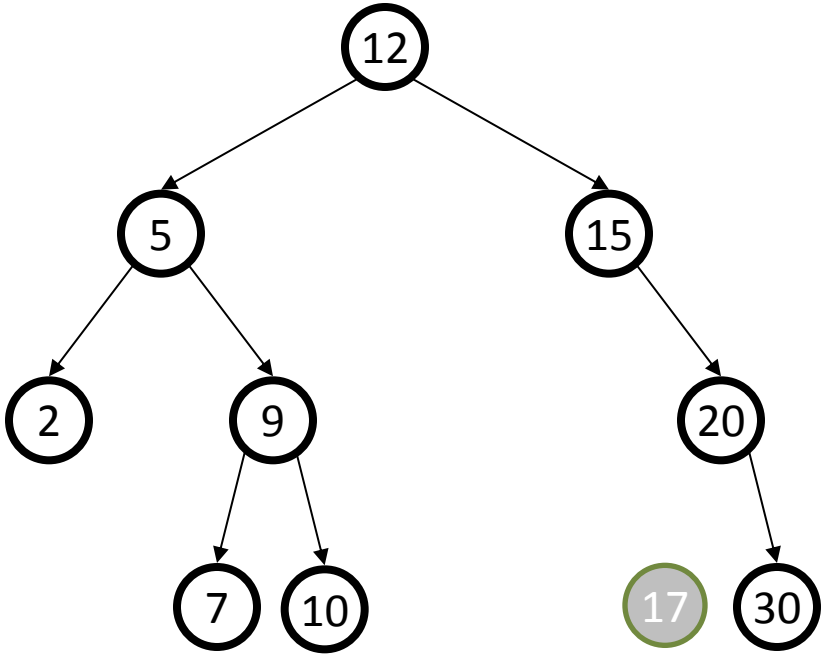
Deletion – The Leaf Case

```
delete(17)
```



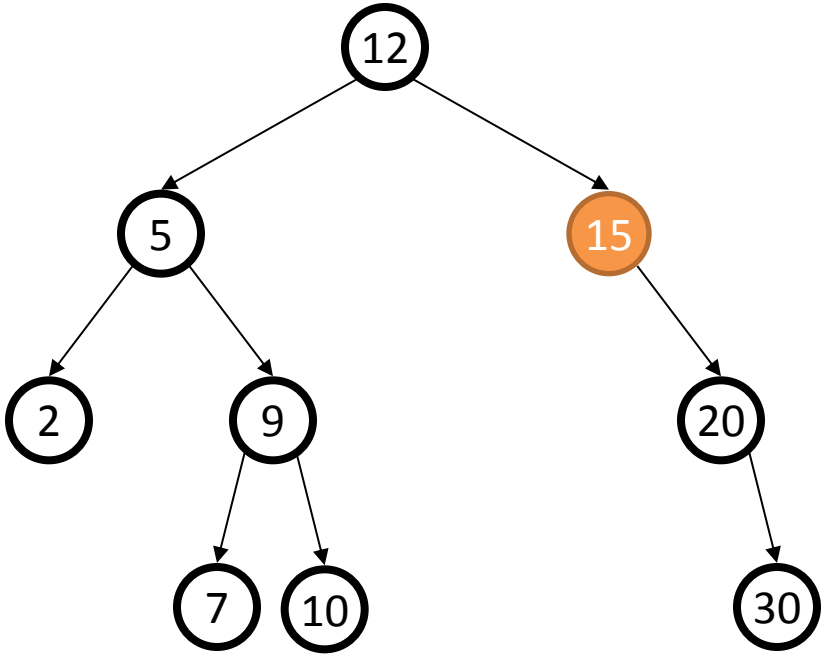
Deletion – The Leaf Case

```
delete(17)
```



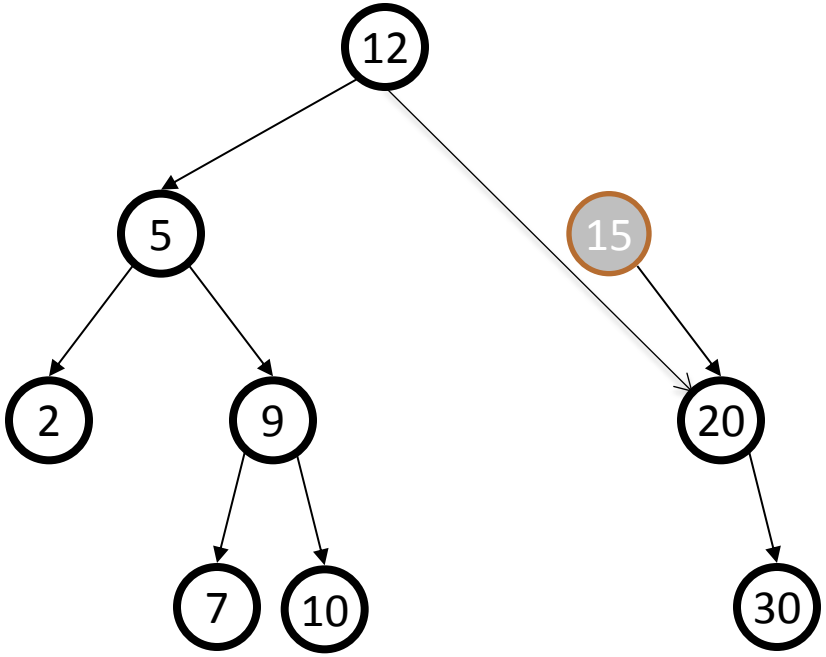
Deletion – The One Child Case

```
delete(15)
```



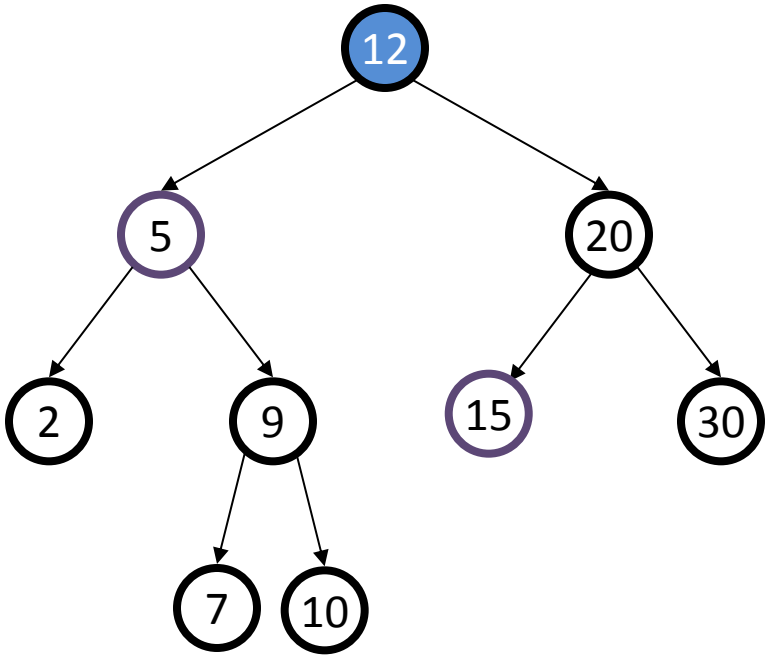
Deletion – The One Child Case

```
delete(15)
```



Deletion – The Two Child Case

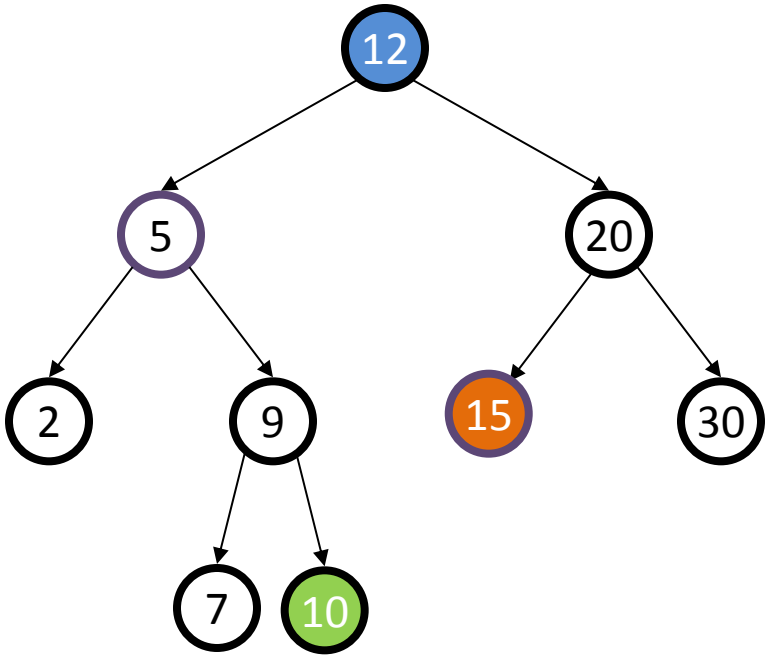
`delete(12)`



What can we replace 12 with?

Deletion – The Two Child Case

```
delete(12)
```

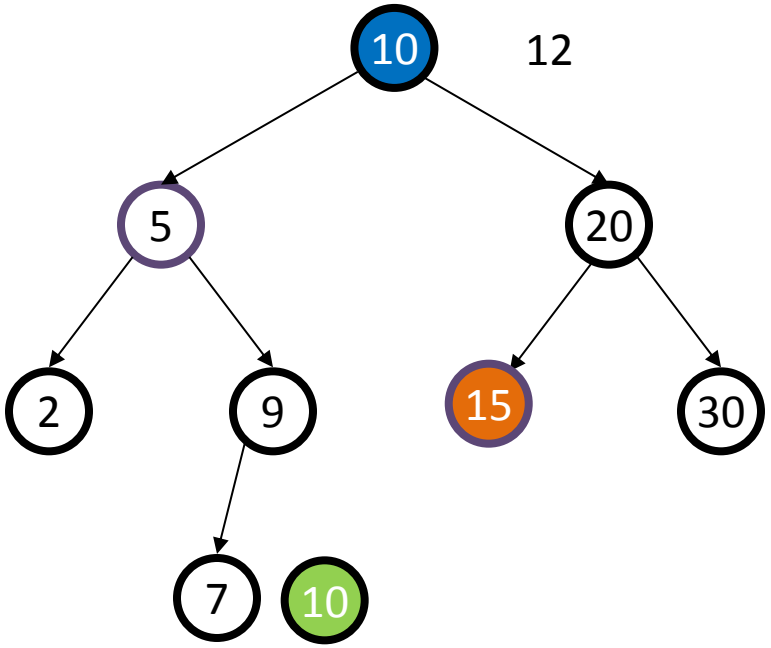


Two candidates:

- the maximum of the left sub-tree (10)
- the minimum of the right sub-tree (15)

Deletion – The Two Child Case

```
delete(12)
```

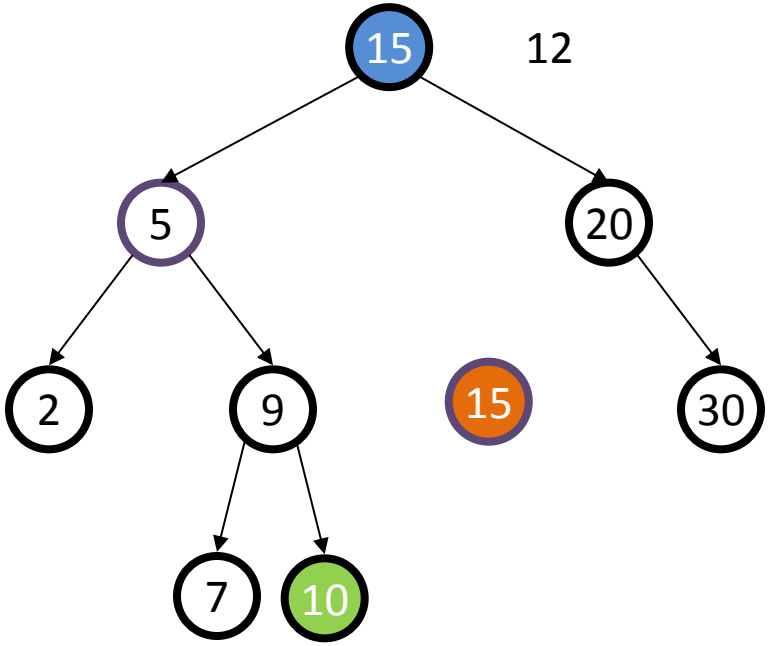


Two candidates:

- the maximum of the left sub-tree (10)
- ~~the minimum of the right sub-tree (15)~~

Deletion – The Two Child Case

```
delete(12)
```



Two candidates:

- ~~the maximum of the left sub-tree (10)~~
- the minimum of the right sub-tree (15)

Deletion – The Two Child Case

Replace the deleted node with a value guaranteed to be between the two child subtrees

Options:

- *successor* from right subtree: **`findMin(node.right)`**
- *predecessor* from left subtree: **`findMax(node.left)`**
  - These are the easy cases of predecessor/successor

Now delete the original node containing *successor* or *predecessor*

- Leaf or one child case – easy cases of delete!

BST, complexity

- O(h) where h is the height of the tree for all operations find, insert and delete
  - N is the size (number of nodes) and h is the height
  - Worst case: h = N - 1
  - Best case: h = log(N)
- Average case
  - Consider all BST obtained by inserting n distinct values in the empty BST
  - Compute the average of height of all those BST
  - Result is $\Theta(\log(N))$