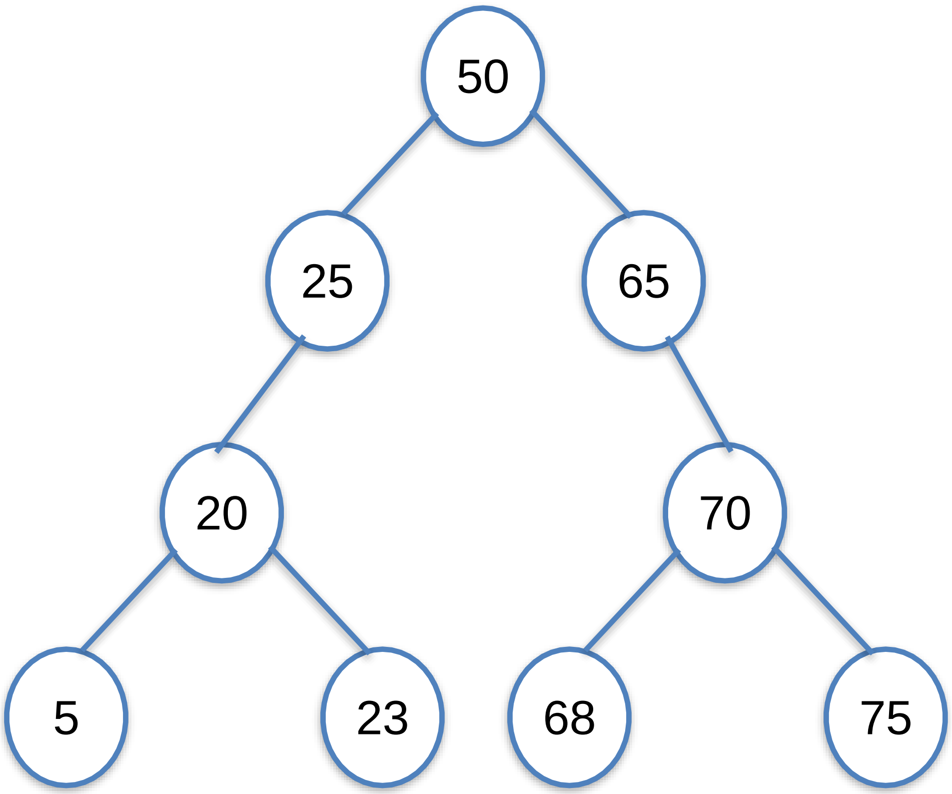
Binary Search Trees

BST property

- Keys in a BST satisfy the ***binary-search-tree property***
- Let x be a node in a binary search tree.
- If y is a node in the left subtree of x, then $y.key <= x.key$
- If y is a node in the right subtree of x, then $y.key >= x.key$



How to print the elements in BST?

Traversals:

- In order (In order tree walk)
    - Prints the key of the root of a subtree between printing the values in its left subtree and printing those in its right subtree
    - **LeftSubTree—Root—RightSubTree**
- Pre order (Pre order tree walk)
    - Prints the root before the values in either subtree
    - **Root—LeftSubTree—RightSubTree**
- Post order (Post order tree walk)
    - Prints the root after the values in its subtrees.
    - **LeftSubTree—RightSubTree—Root**

Querying a binary search tree

- **Query operations** - Search, Minimum, Maximum, Successor and Predecessor

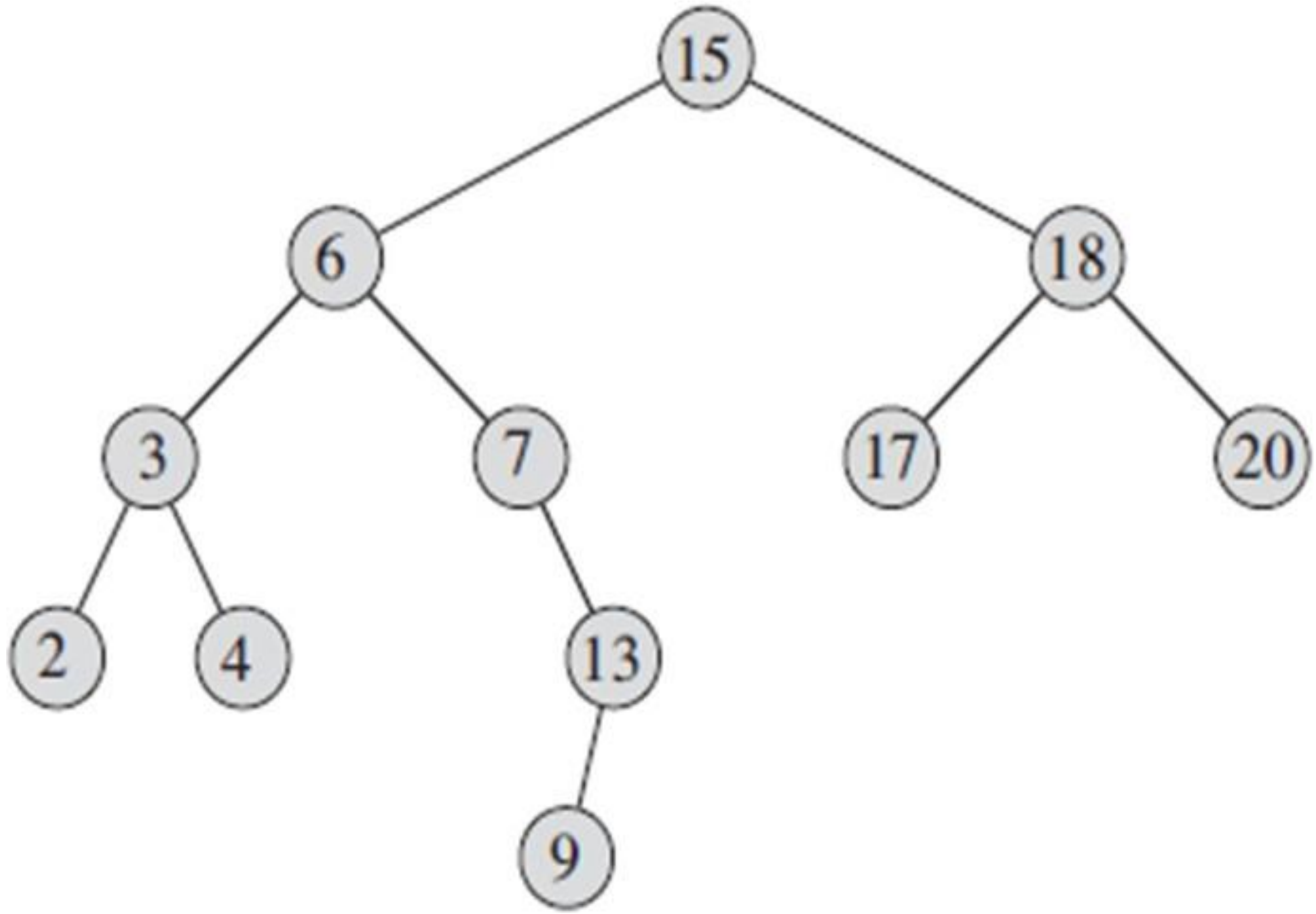
- BST support these operations each one in time **O(h)** on any binary search tree of **height h.**

TREE-SEARCH(x, k)

- Given a pointer to the root of the tree and a key k, TREE-SEARCH returns a pointer to a node with key k if one exists; otherwise, it returns NIL.

```
TREE-SEARCH(x, k)
if x == NIL or k == x.key
    return x
if k < x.key
    return TREE-SEARCH(x.left, k)
else return TREE-SEARCH(x.right, k)
```

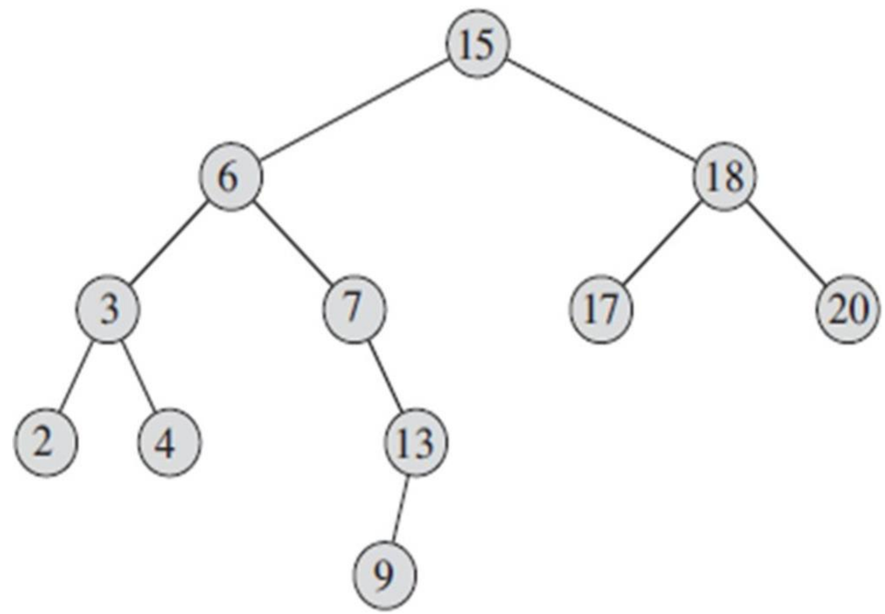
HOW DO WE FIND THE MINIMUM AND MAXIMUM ELEMENT IN BST?


15
6
18
3
7
17
20
2
4
13
9

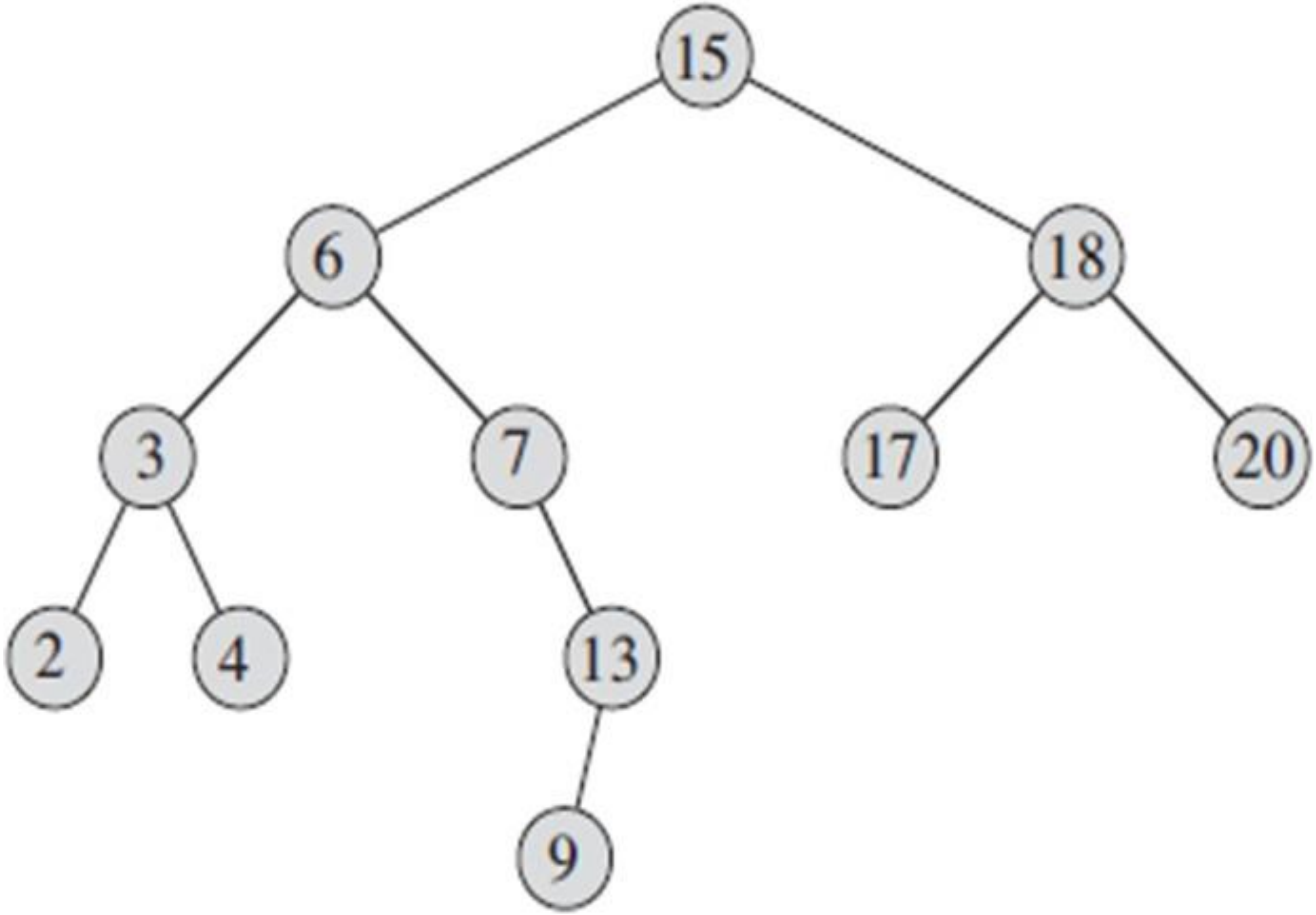
Minimum element

- The following procedure returns a pointer to the minimum element in the subtree rooted at a given node x, which we assume to be non-NIL:

```
TREE-MINIMUM(x)
while x.left ≠ NIL
    x = x.left
return x
```

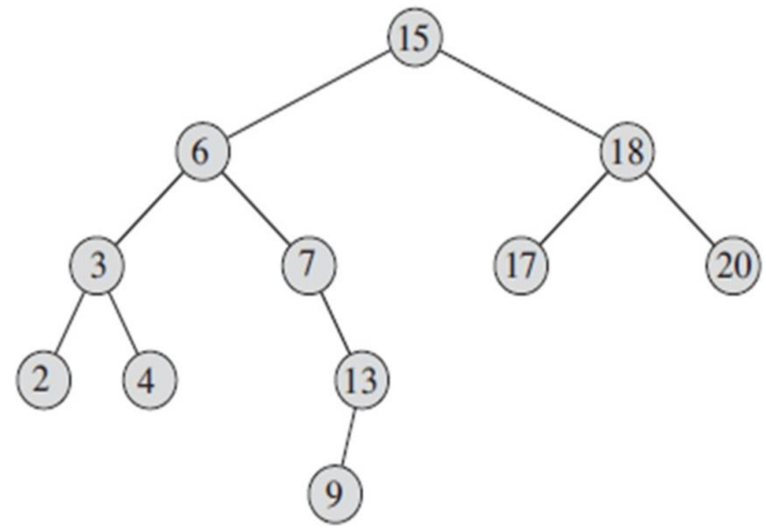


TREE -MAXIMUM



TREE-MAXIMUM(x)

```
TREE-MAXIMUM(x)
while x.right ≠ NIL
    x = x.right
return x
```



- Running time of Tree-Minimum and Tree-Maximum

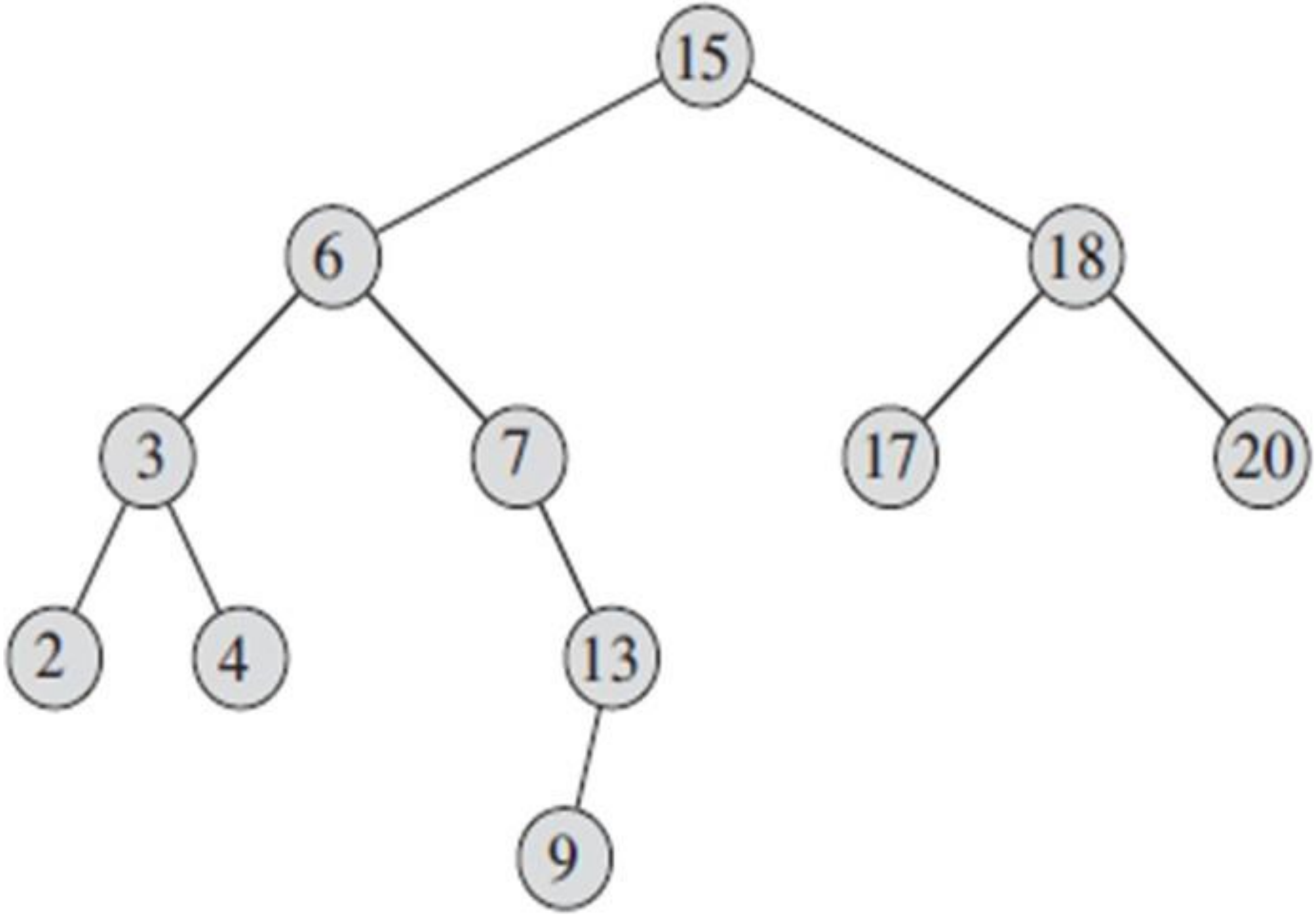
- In this case also, the sequence of nodes encountered forms a simple path downward from the root.

How do we find the Predecessor and Successor of a node in BST?

Successor and predecessor

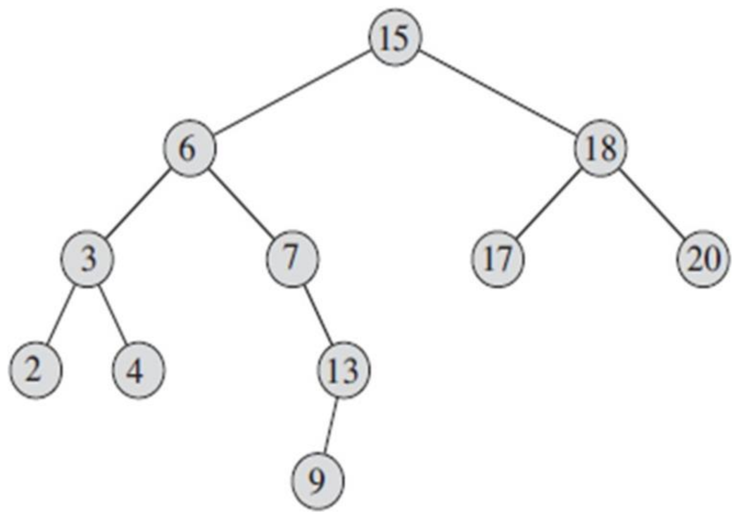
- Given a node in a BST, sometimes we need to find its successor in the sorted order determined by an in order tree walk.
- If all keys are distinct, the successor of a node x is the node with the smallest key greater than x.*key*.
- The structure of a BST allows us to determine the successor of a node without ever comparing keys.
- The standard procedure returns the successor of a node x in a BST if it exists, and NIL if x has the largest key in the tree

SUCCESSOR & PREDECESSOR OF A NODE



TREE-SUCCESSOR(x)

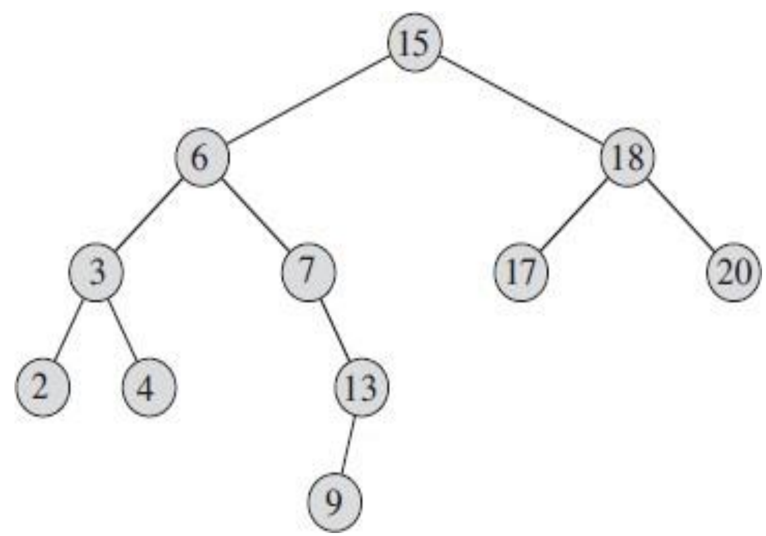
```
TREE-SUCCESSOR(x)
if x.right ≠ NIL                          // Case 1
    return TREE-MINIMUM(x.right)
y = x.p
while y ≠ NIL and x == y.right            // Case 2
    x = y
    y = y.p
return y
```



TREE-SUCCESSOR(x): 2 cases

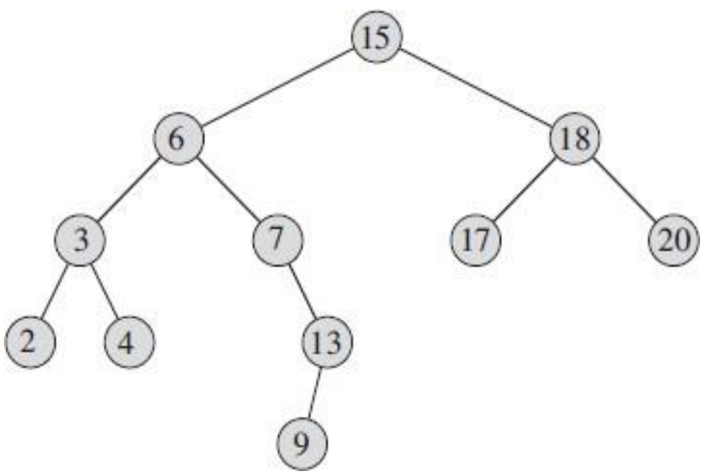
- Case 1: If the right subtree of node x is nonempty, then the successor of x is just the leftmost node in x’s right subtree, which we find in line 2 by calling TREE-MINIMUM(x.*right*)
- Eg: Successor of the node with key 15

```
TREE-SUCCESSOR(x)
1  if x.right ≠ NIL
2      return TREE-MINIMUM(x.right)
3  y = x.p
4  while y ≠ NIL and x == y.right
5      x = y
6      y = y.p
7  return y
```



- Case 2: If the right subtree of node x is empty and x has a successor y, then y is the lowest ancestor of x whose left child is also an ancestor of x.

Eg: Successor of the node with key 13?



TREE-SUCCESSOR(x)

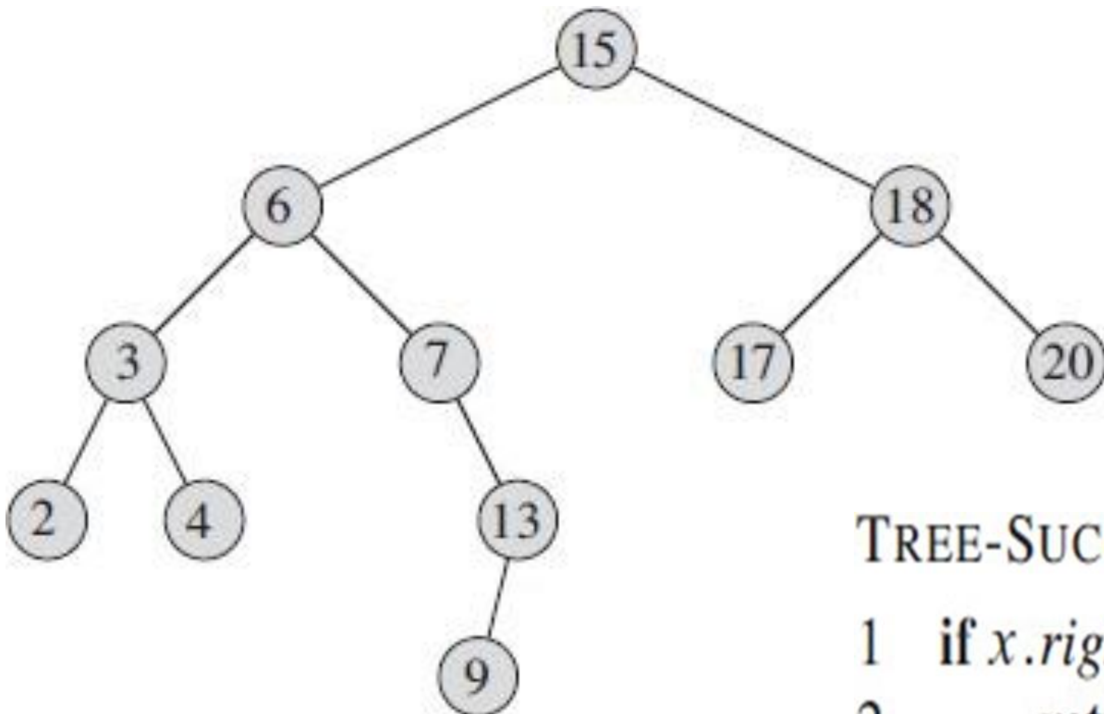
```
1  if x.right ≠ NIL
2      return TREE-MINIMUM(x.right)
3  y = x.p
4  while y ≠ NIL and x == y.right
5      x = y
6      y = y.p
7  return y
```

- To find y, we simply go up the tree from x until we encounter a node that is the left child of its parent; lines 3–7 of TREE-SUCCESSOR handle this case.

TREE-SUCCESSOR(x)

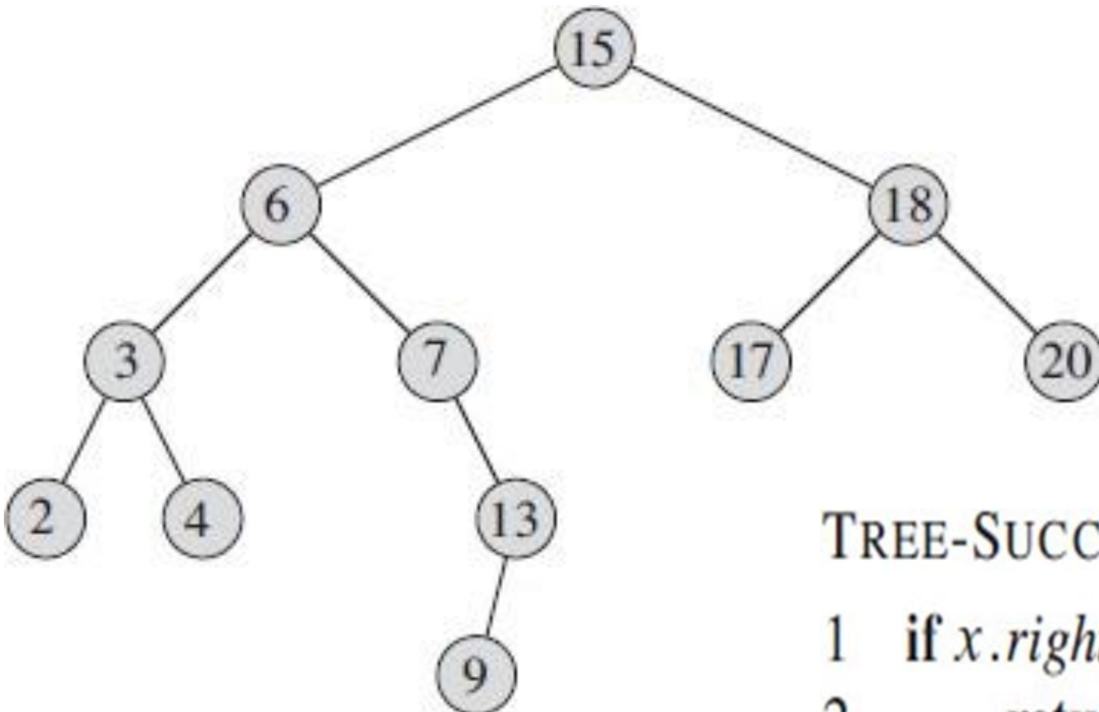
```
TREE-SUCCESSOR(x)
if x.right ≠ NIL                              // Case 1
    return TREE-MINIMUM(x.right)
y = x.p
while y ≠ NIL and x == y.right
    x = y                                     // Case 2
    y = y.p
return y
```

Trace the pseudo code to find the successor of 4




```
TREE-SUCCESSOR(x)
if x.right ≠ NIL
    return TREE-MINIMUM(x.right)
y = x.p
while y ≠ NIL and x == y.right
    x = y
    y = y.p
return y
```

Trace the pseudo code to find the successor of 9



TREE-SUCCESSOR(x)

```
if x.right ≠ NIL
    return TREE-MINIMUM(x.right)
y = x.p
while y ≠ NIL and x == y.right
    x = y
    y = y.p
return y
```

Running time

- The running time of TREE-SUCCESSOR on a tree of height h is O(h), since we either follow a simple path up the tree or follow a simple path down the tree.
- The procedure TREE-PREDECESSOR, which is symmetric to TREE-SUCCESSOR, also runs in time O(h)

Exercise

- Write the pseudo-code for TREE-PREDECESSOR(x)
- Even if keys are not distinct, we define the successor and predecessor of any node x as the node returned by calls made to TREE- SUCCESSOR(x) and TREE- PREDECESSOR(x) respectively

Running-time

We can implement the dynamic-set operations SEARCH, MINIMUM, MAXIMUM, SUCCESSOR, and PREDECESSOR so that each one runs in O(h) time on a binary search tree of height h.

References

- CLRS Book