빅데이터 혁신공유대학

파이썬으로 배우는 데이터 구조

한동대학교 전산전자공학부

김영섭 교수

Data Structures in Python
Chapter 7 - 2

- Binary Search Tree(BST)
- **BST Algorithms**
- AVL Tree
- AVL Algorithms

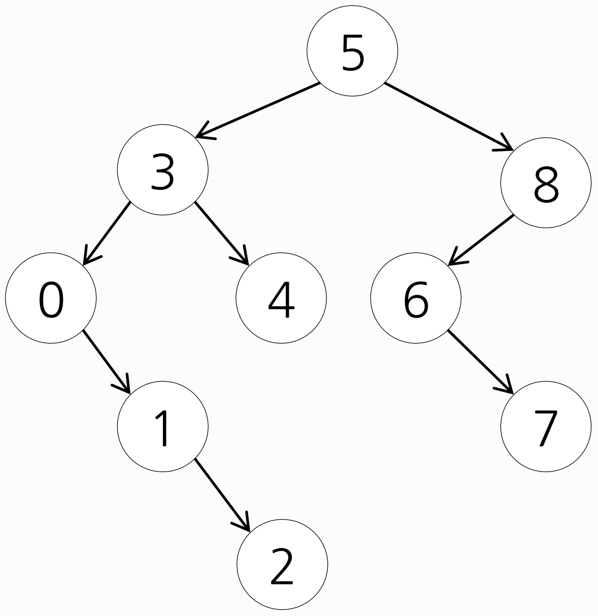
Agenda & Readings

- Binary Search Tree(BST) Algorithms
  - minimum() and maximum()
  - predecessor() and successor()
  - delete(), _delete()
  - Converting Binary Tree to BST
  - LCA(Lowest Common Ancestor)

- Reference:
  - Problem Solving with Algorithms and Data Structures
  - Chapter 6 - Tree

minimum(), maximum():

- minimum() and maximum() returns the node with min or max key.
  - Note that the entire tree does not need to be searched.
  - The minimum key is located at the left most node, the maximum at the right most node.
  - Complexity of algorithm to find the maximum or minimum will be O(log N) in almost balanced binary tree. If tree is skewed, then we have worst case complexity of O(N).

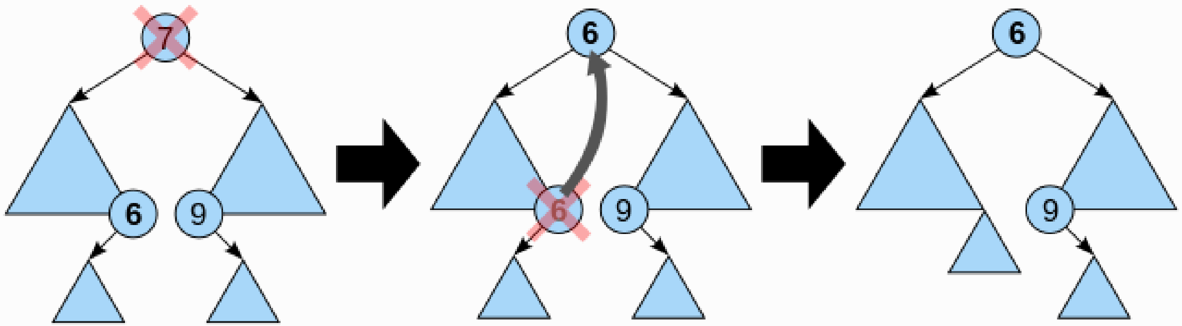



```
def minimum(self, node = None):
    if node is None: node = self.root
    return self._minimum(node)

def _minimum(self, node):
    if node.left == None: return node
    return self._minimum(node.left)
```

predecessor(), successor():

- Predecessor
  - The predecessor is **the largest node** that is smaller than the root (current node) - thus it is on the left branch of the Binary Search Tree, and the **rightmost leaf** (largest on the left branch).
- Successor
  - The successor is **the smallest node** that is bigger than the root/current - thus it is on the right branch of the BST, and also on **the leftmost leaf** (smallest on the right branch).
- Notice that either predecessor or successor has at most one child if any.
- Complexity of algorithm: O(log N) if balanced, and O(N) if the tree is skewed.

```
def successor(self, node = None):
    if node is None: node = self.root
    return self._successor(node)

def _successor(self, node):
    if node and node.right:
        return self._minimum(node.right)
    return None
```



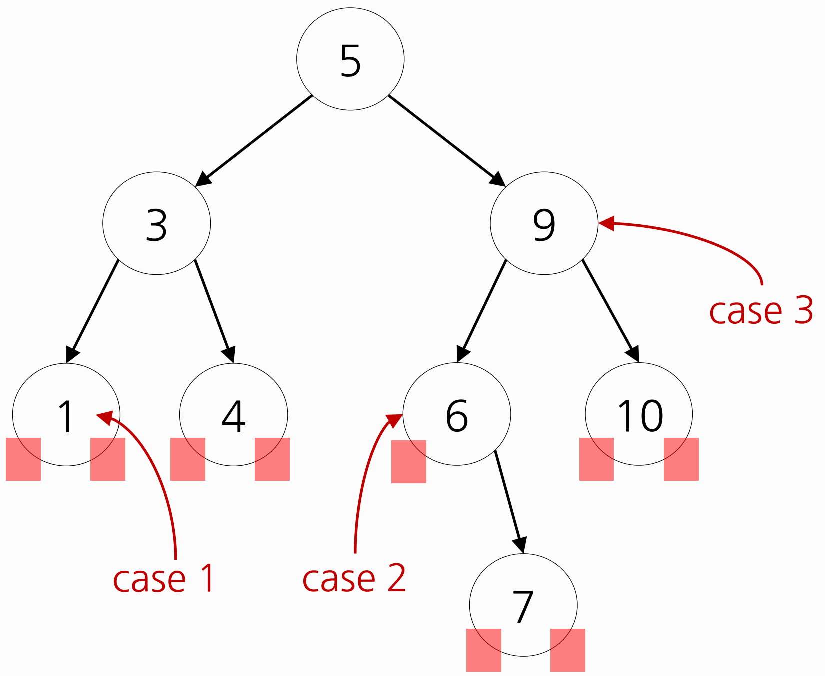
predecessor(), successor():

- Predecessor
  - The predecessor is **the largest node** that is smaller than the root (current node) - thus it is on the left branch of the Binary Search Tree, and the **rightmost leaf** (largest on the left branch).
- Successor
  - The successor is **the smallest node** that is bigger than the root/current - thus it is on the right branch of the BST, and also on **the leftmost leaf** (smallest on the right branch).
- Notice that either predecessor or successor has at most one child if any.
- Complexity of algorithm: O(log N) if balanced, and O(N) if the tree is skewed.

```
def successor(self, node = None):
    if node is None: node = self.root
    return self._successor(node)

def _successor(self, node):
    if node and node.right:
        return self._minimum(node.right)
    return None
```

delete()

- When we delete a node, **three possibilities** arise depending on how many children the node to be deleted has:
  - Case 1: No child
  - Case 2: One child
  - Case 3: Two children

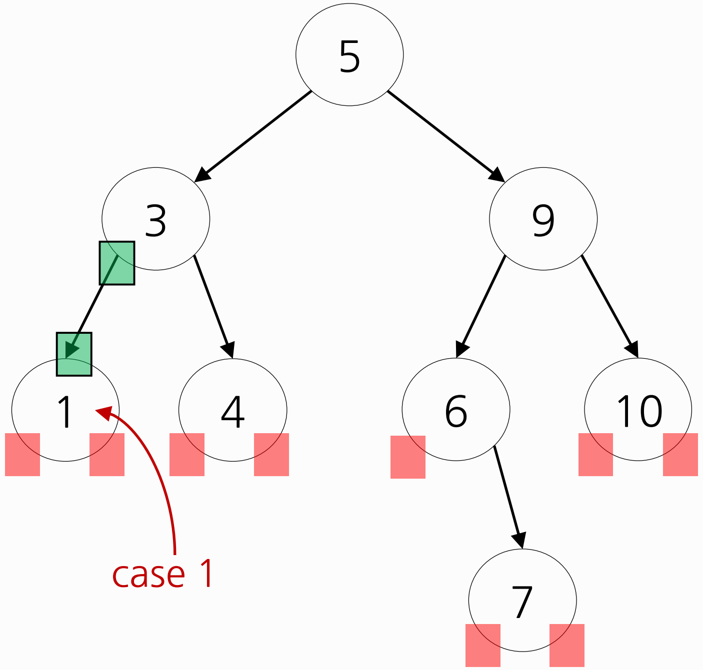



```
def _delete(self, node, key):
    if node is None: return node

    if key < node.key:
        node.left = self._delete(node.left, key)
    elif key > node.key:
        node.right = self._delete(node.right, key)
    else:     # key == node.key:
        if node.left and node.right: # two children

                two children case

        elif node.left or node.right: # one child
                one child case
        else:                         # no child
                no child case
    return node
```

find

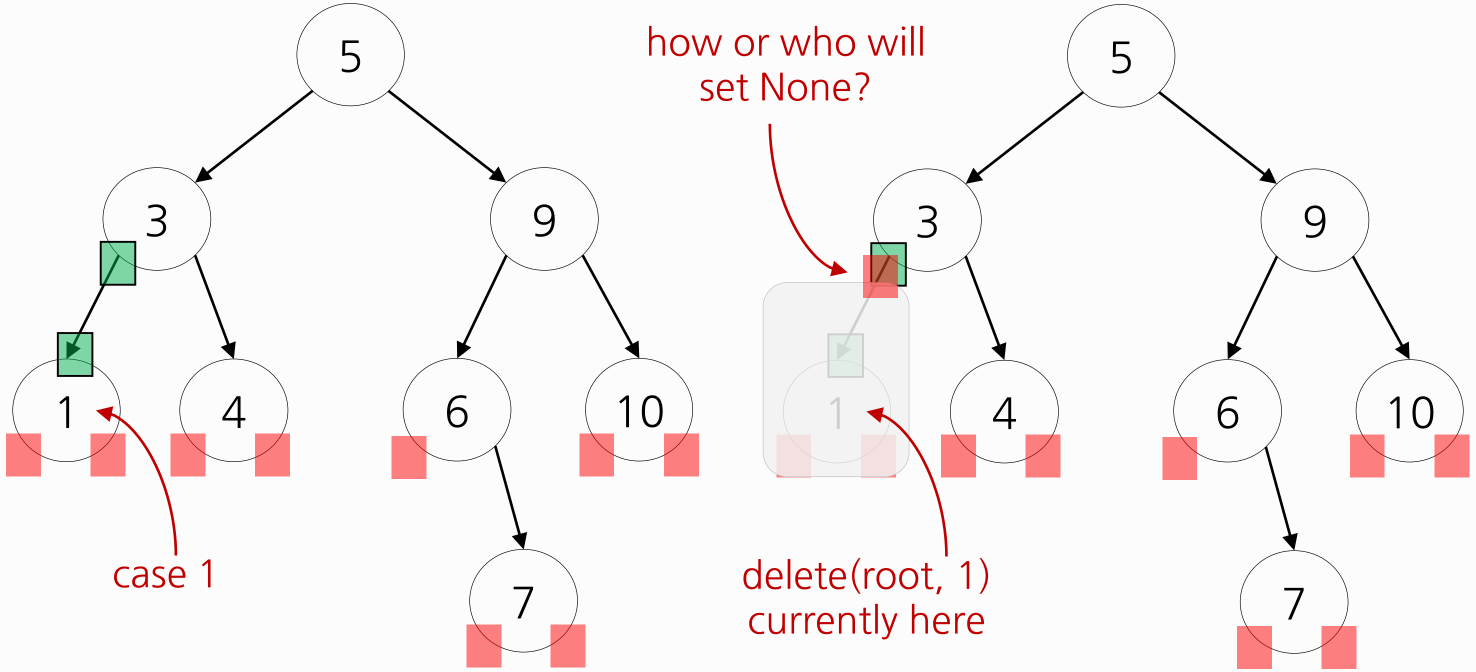
delete()

- **Case 1:** No child - Simply return **None**.



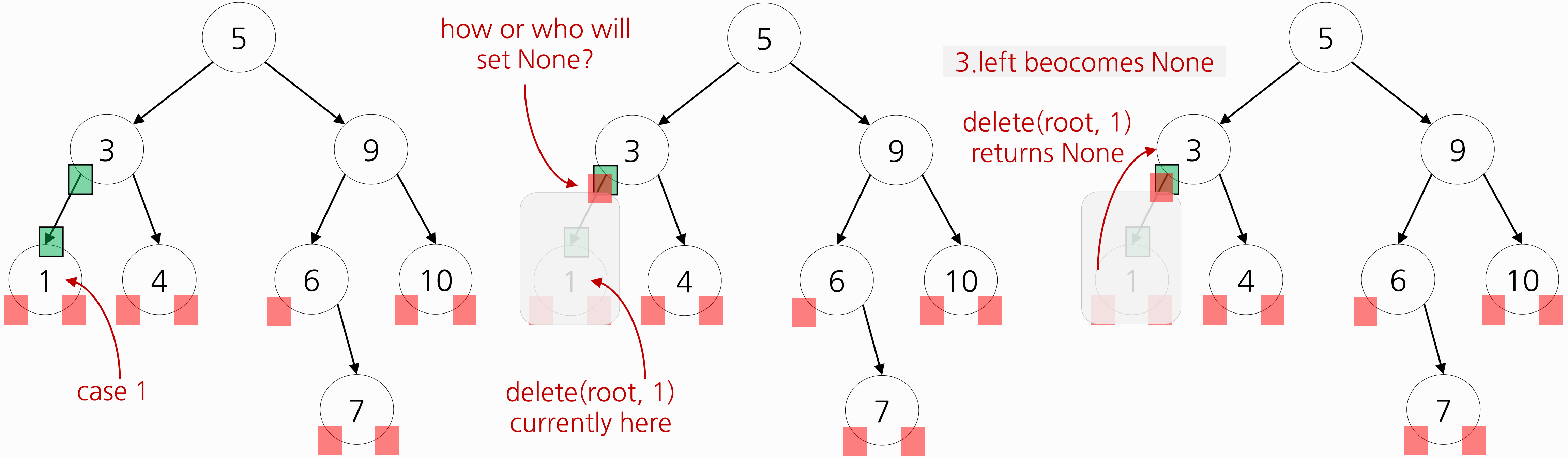
delete()

- **Case 1:** No child - Simply return **None**.



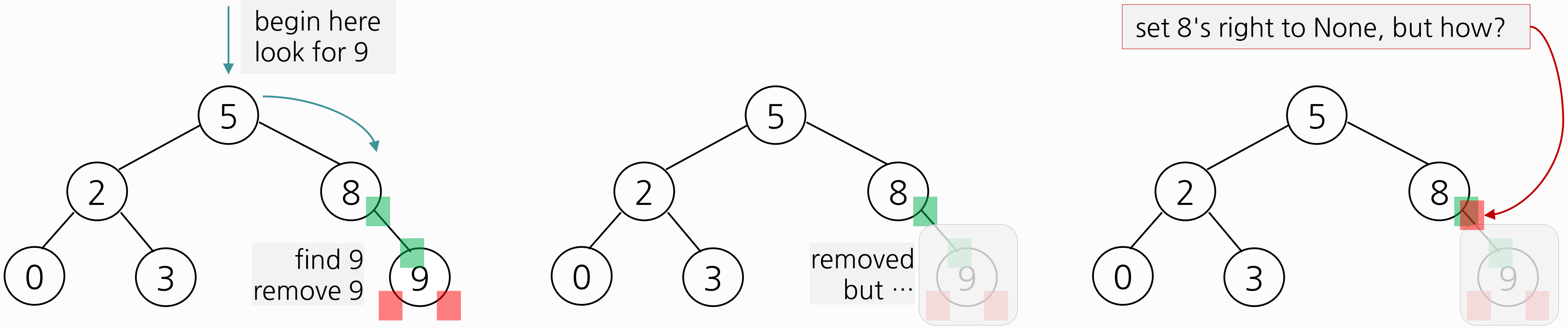
delete()

- **Case 1:** No child - Simply return **None**.



delete() Example

- Case 1: No child - Simply return **None**.

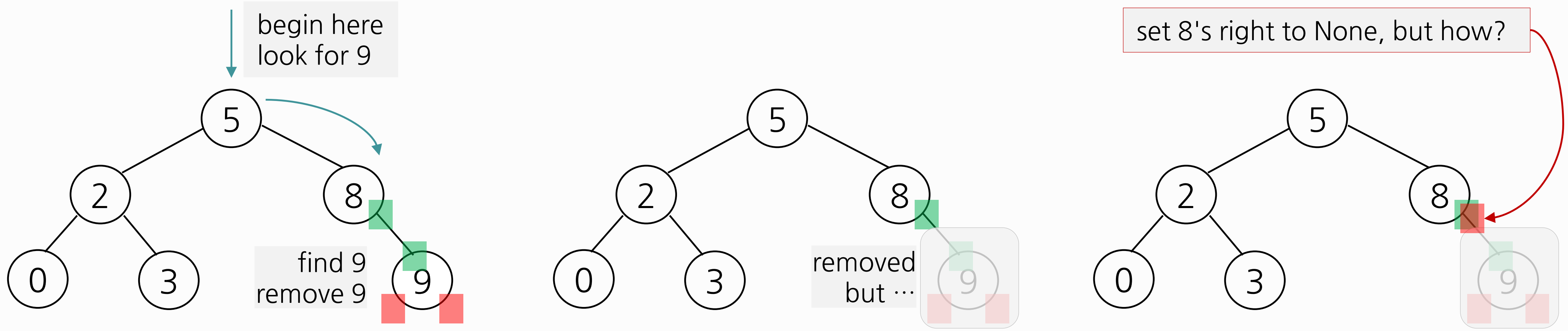


find

```
def _delete(self, node, key):
    ...
    elif key > node.key:
        node.right = self._delete(node.right, key)
    ...
    ...
    ...
    return node
```

delete() Example

- Case 1: No child - Simply return **None**.



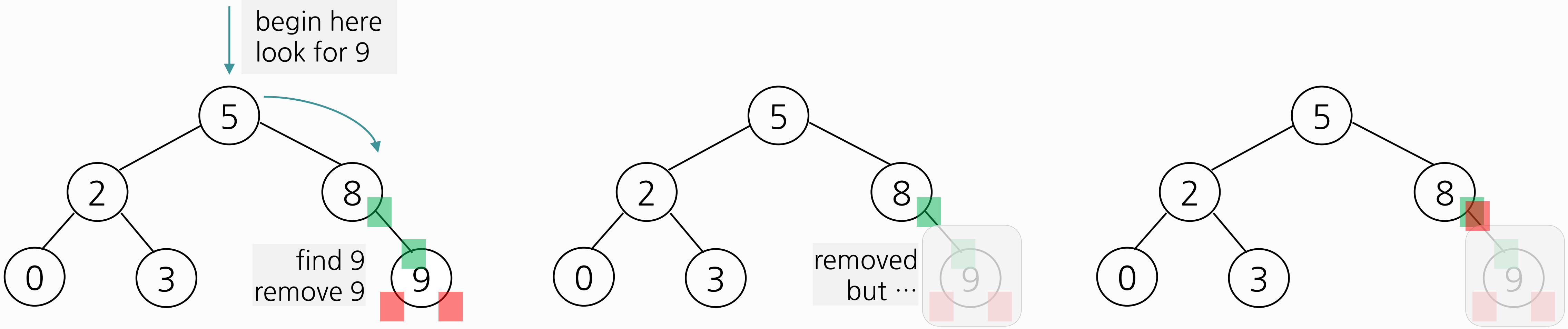
find

```
def _delete(self, node, key):
    ...
    elif key > node.key:
        node.right = self._delete(node.right, key)
    ...
    ...
    ...
    return node
```

```
def _delete(self, node, key):
    ...
    else:  # key == node.key:
        if node.left and node.right:
          two children case
        elif node.left or node.right:
          one child case
        else:                  # no child
            node = None
    return node
```

delete() Example

- Case 1: No child - Simply return **None**.



It sets 8's right to None.

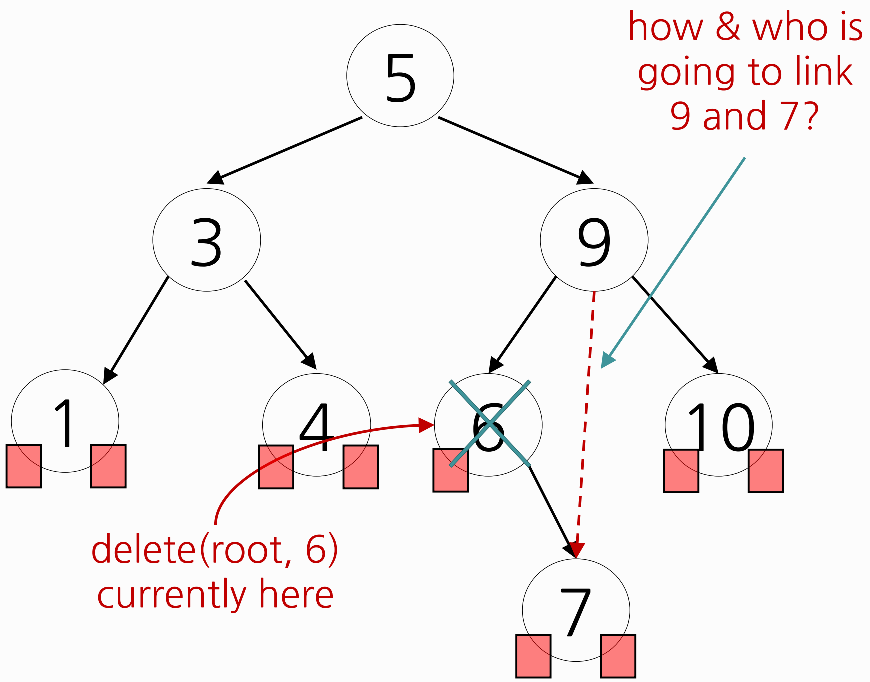
find

```
def _delete(self, node, key):
    ...
    elif key > node.key:
        node.right = self._delete(node.right, key)
    ...
    ...
    ...
    return node
```

```
def _delete(self, node, key):
    ...
    else:  # key == node.key:
        if node.left and node.right:
         two children case
        elif node.left or node.right:
         one child case
        else:               # no child
            node = None
    return node
```

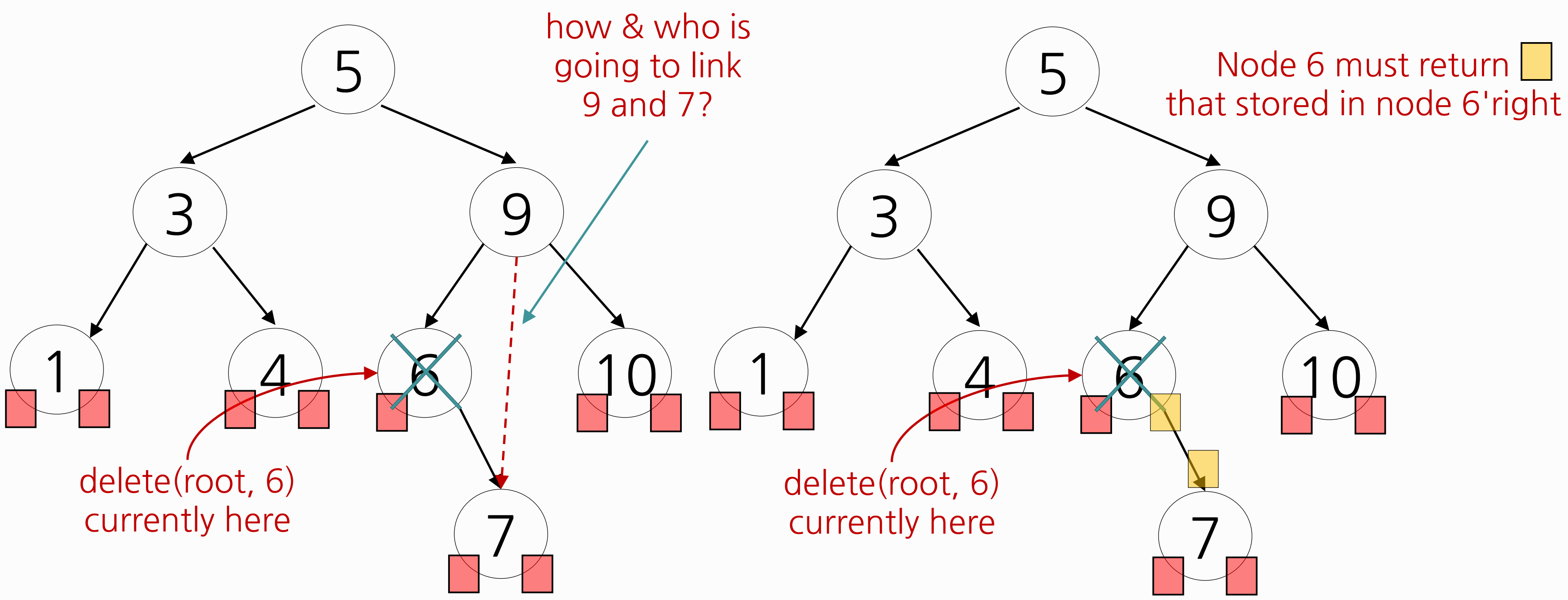
delete()

- **Case 2:** One child



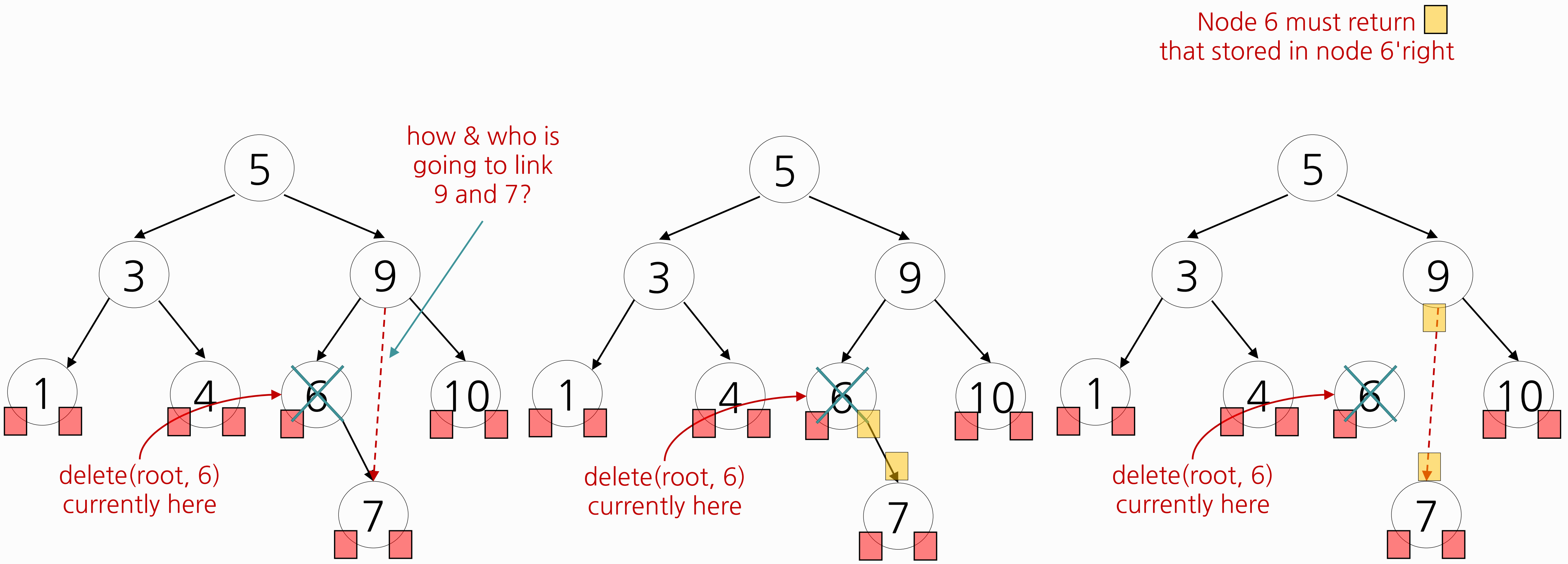
delete()

- **Case 2:** One child



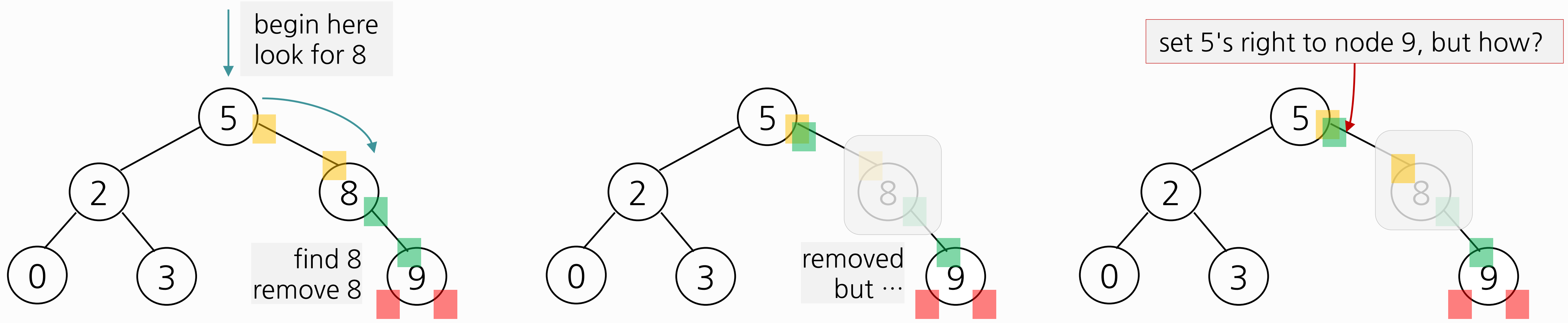
delete()

- **Case 2:** One child



delete() Example

- Case 2: One child

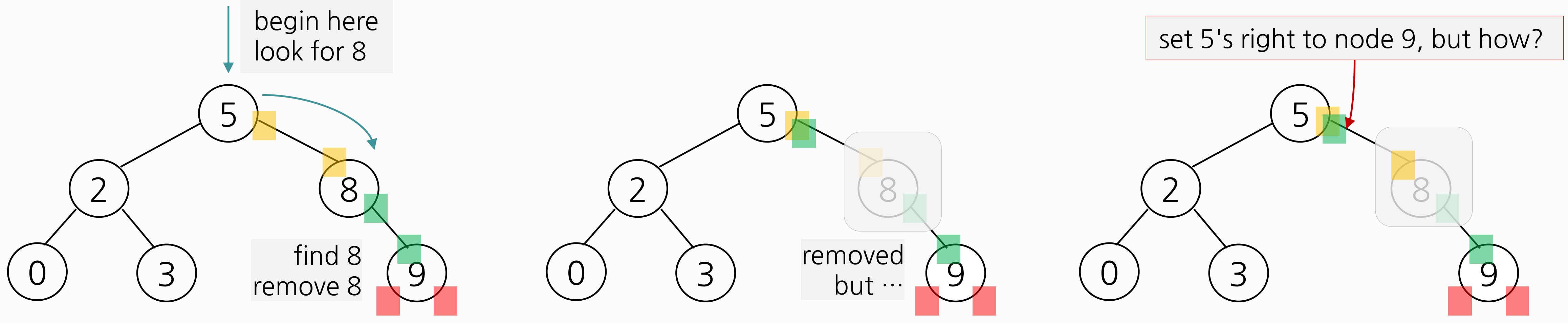


find

```
def _delete(self, node, key):
    ...
    elif key > node.key:
        node.right = self._delete(node.right, key)
    ...
    ...
    ...
    return node
```

delete() Example

- Case 2: One child



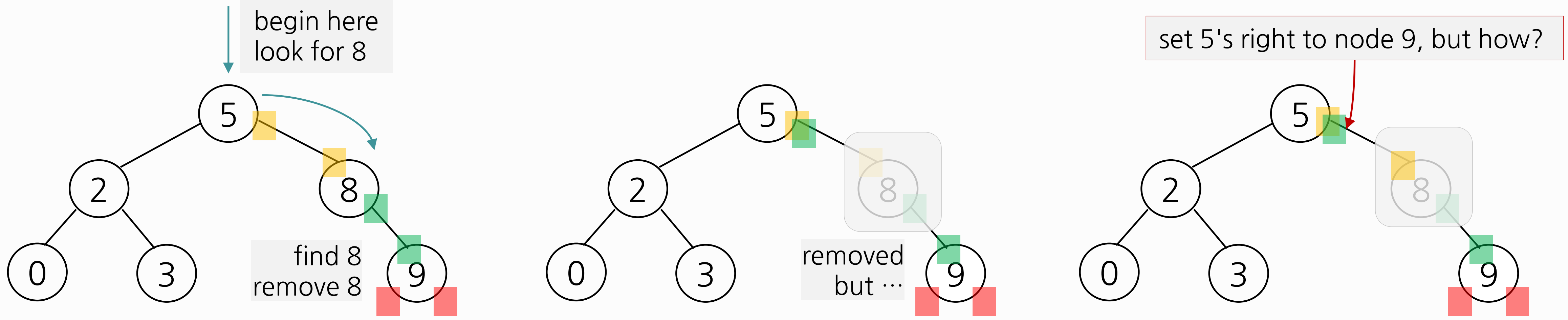
find

```
def _delete(self, node, key):
    ...
    elif key > node.key:
        node.right = self._delete(node.right, key)
    ...
    ...
    ...
    return node
```

```
def _delete(self, node, key):
        ...
        elif node.left or node.right: # one child
            if node.left:
                node = node.left
            else:
                node = node.right
        else:                         # no child
            node = None
    return node
```

delete() Example

- Case 2: One child



It sets 5's right to node 9.

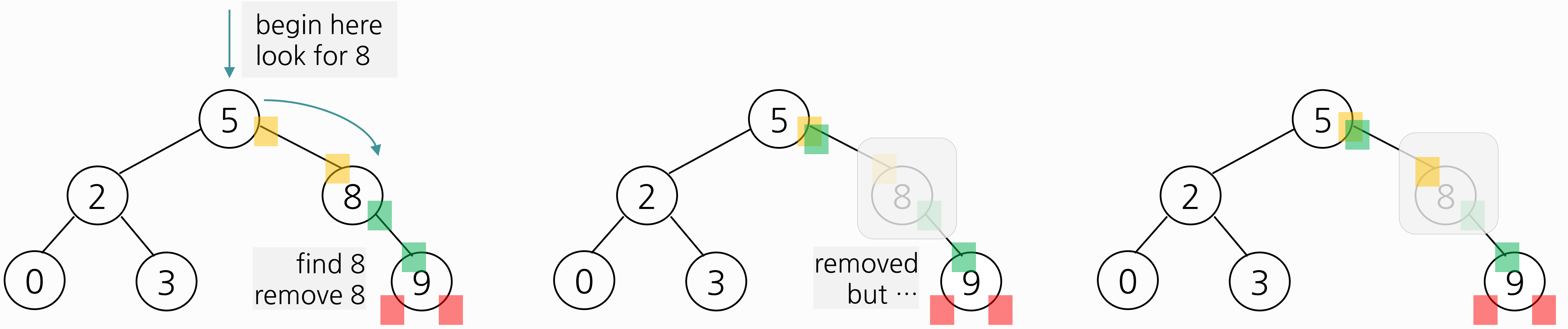
find

```
def _delete(self, node, key):
    ...
    elif key > node.key:
        node.right = self._delete(node.right, key)
    ...
    ...
    ...
    return node
```

```
def _delete(self, node, key):
        ...
        elif node.left or node.right: # one child
            if node.left:
                node = node.left
            else:
                node = node.right
        else:                         # no child
            node = None
    return node
```

delete() Example

- Case 2: One child



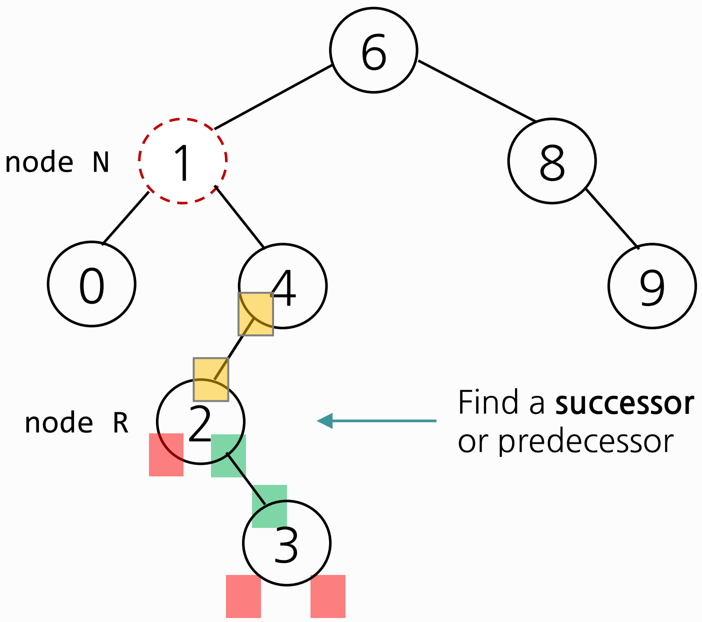

```
def _delete(self, node, key):
        ...
        elif node.left or node.right: # one child
            node = node.left or node.right
        else:                         # no child
            node = None
    return node
```

can be simplified

```
def _delete(self, node, key):
        ...
        elif node.left or node.right: # one child
            if node.left:
                node = node.left
            else:
                node = node.right
        else:                         # no child
            node = None
    return node
```

delete() Example

- Case 3: Two children
  1. Find the node **N** to delete, but do not delete it.
  2. Choose either its **successor** or its **predecessor** node, **R**.
  3. Simply, replace N's key with R's key
  4. Then, recursively call to delete on node R in the subtree. The node R must be in either Case 1 or Case 2.

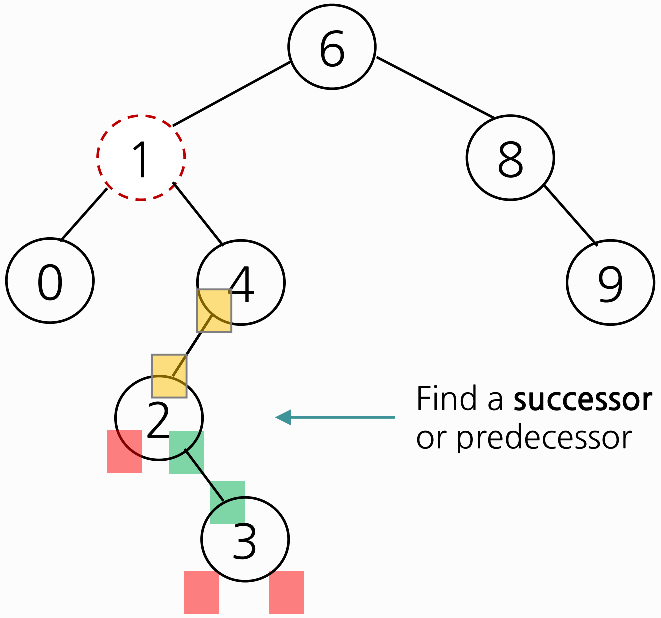



```
def _delete(self, node, key):
    if node is None: return node

    if key < node.key:
        node.left = self._delete(node.left, key)
    elif key > node.key:
        node.right = self._delete(node.right, key)
    else:     # key == node.key:
        if node.left and node.right: # two children

                two children case

        elif node.left or node.right: # one child
                one child case
        else:                         # no child
                no child case
    return node
```

find

delete() Example

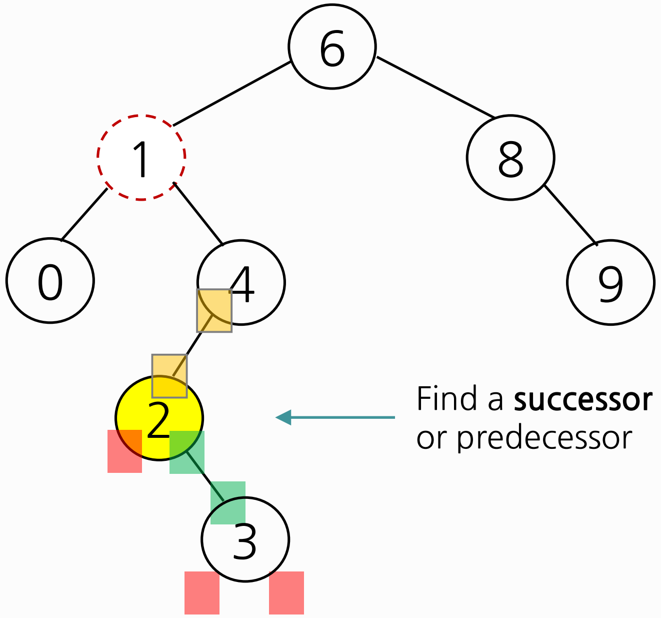
- Case 3: Two children




```
1. find the node 1 to delete
```

delete() Example

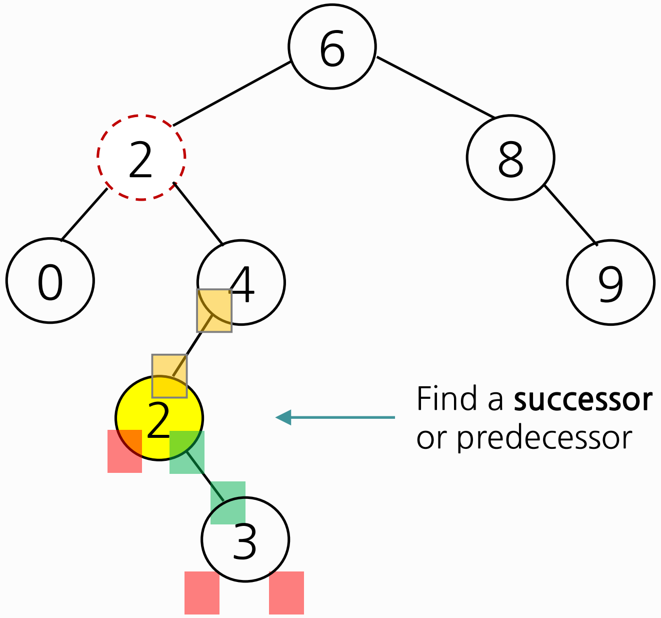
- Case 3: Two children




```
1. find the node 1 to delete
2. if (two children case),
   find 1's successor's key = 2
```

delete() Example

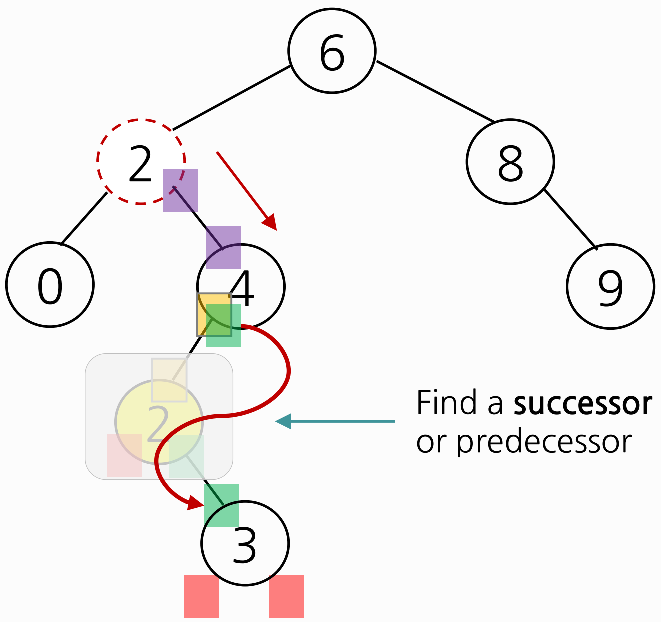
- Case 3: Two children




```
1. find the node 1 to delete
2. if (two children case),
   find 1's successor's key = 2
3. replace the node 1 with 2
```

delete() Example

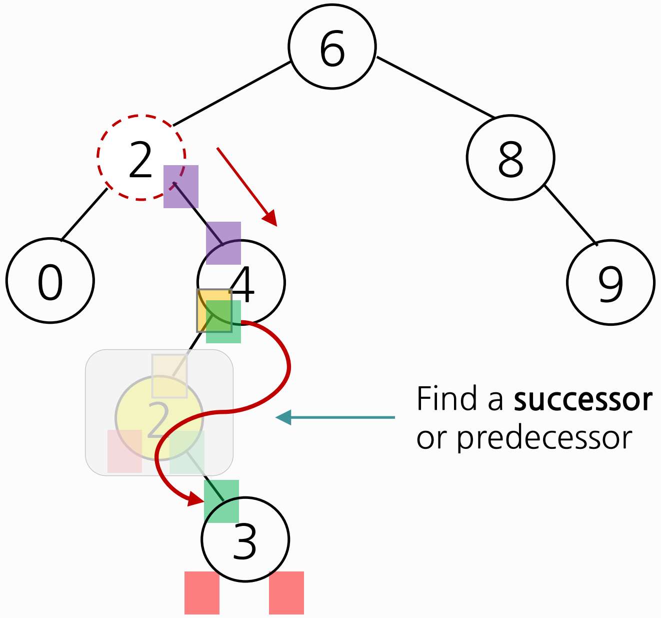
- Case 3: Two children




```
1. find the node 1 to delete
2. if (two children case),
   find 1's successor's key = 2
3. replace the node 1 with 2
4. invoke
   node.right = self._delete(node.right, 2)
```

delete() Example

- Case 3: Two children



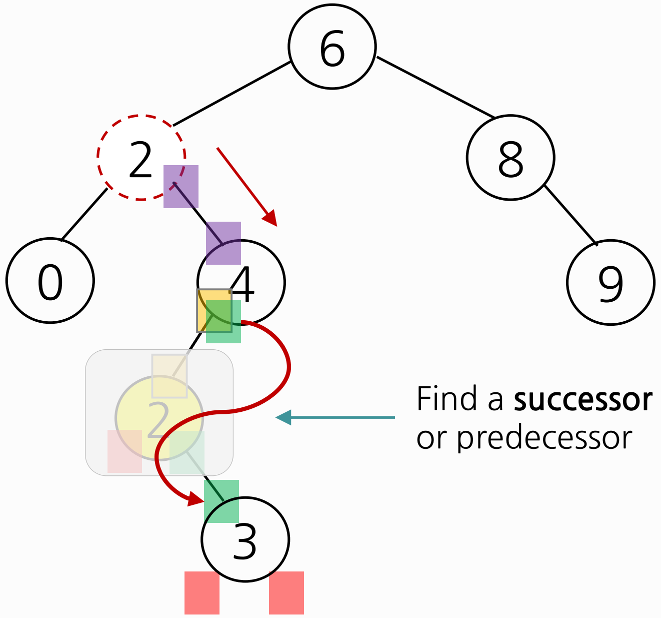

```
1. find the node 1 to delete
2. if (two children case),
   find 1's successor's key = 2
3. replace the node 1 with 2
4. invoke
   node.right = self._delete(node.right, 2)
```

Some thoughts:

- Step 2 Get the heights of two subtree first.
  - If right subtree height is larger, then use the successor. Otherwise use the predecessor to shorten the tree height.
- Step 4 simply uses the code for one-child case deletion, recusively.

delete() Example

- Case 3: Two children




```
1. find the node 1 to delete
2. if (two children case),
   find 1's successor's key = 2
3. replace the node 1 with 2
4. invoke
   node.right = self._delete(node.right, 2)
```

Some thoughts:

- Step 2 Get the heights of two subtree first.
  - If right subtree height is larger, then use the successor. Otherwise use the predecessor to shorten the tree height.
- Step 4 simply uses the code for one-child case deletion, recursively.

Some questions:

- What if successor has **two** children?
  - **Not possible !**
  - Because if it has two nodes, at least one of them is less than it, then in the process of finding successor, we won't pick it !

delete(): Exercise

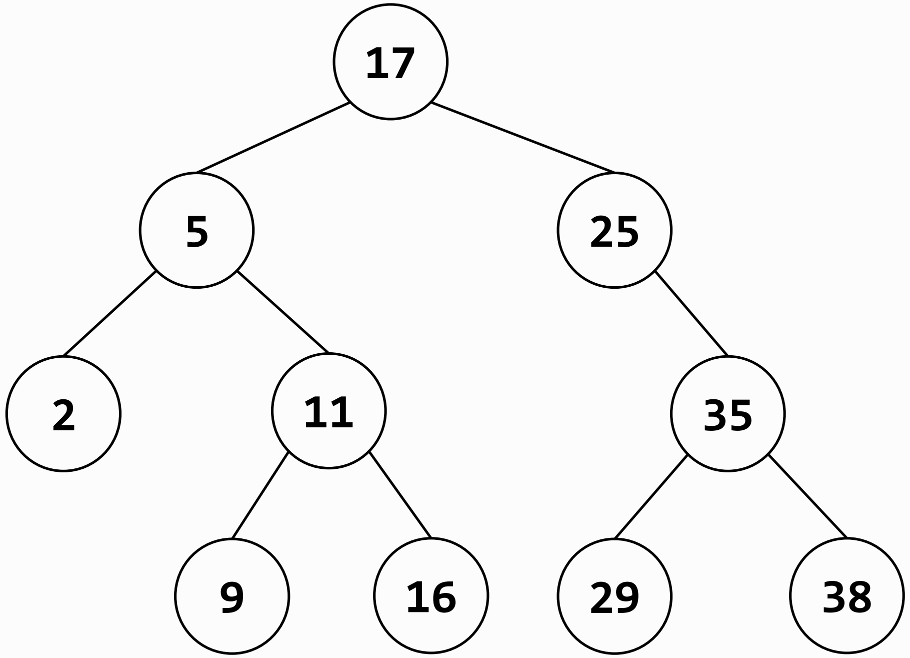
- Delete the root 5 times consecutively. Delete the node from the higher subtree if necessary.

```
        ____________17
       /              \
     5______           25___
    /       \              \
   2         11            _35
            /  \          /   \
         __9    16       29    39
        /
       7
        \
         8
```

isBST() - Validate Binary Search Tree

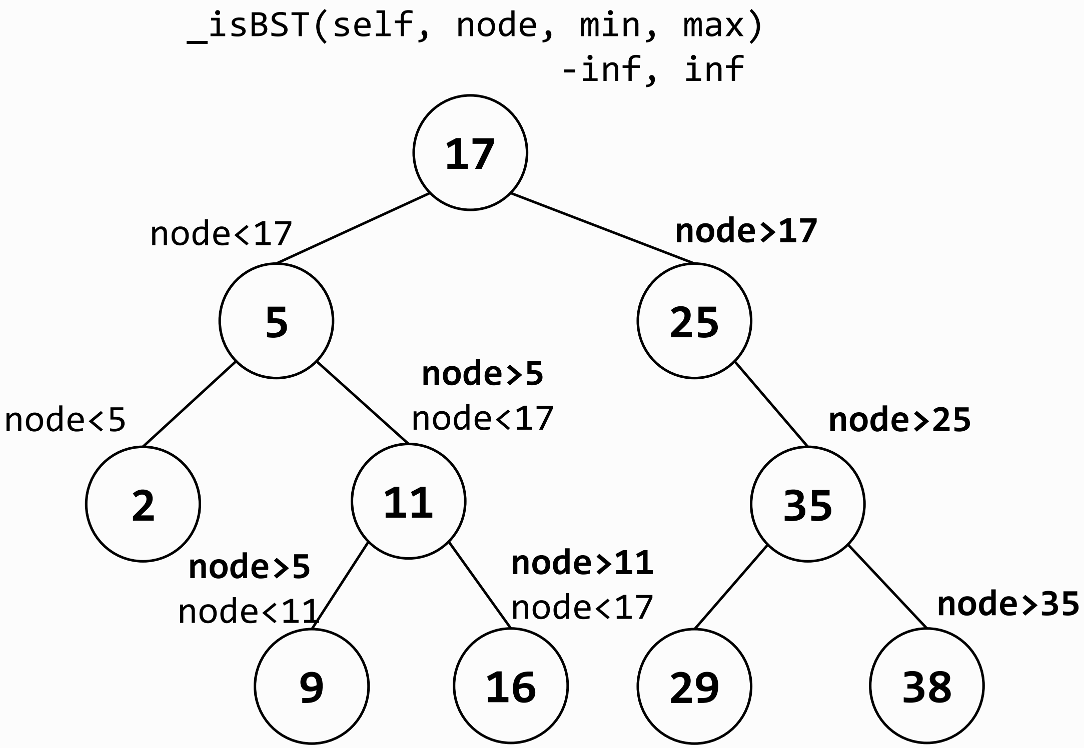
- A BST is defined as follows:
  - The left subtree of a node contains only nodes with **keys less than** the node's key.
  - The right subtree of a node contains only nodes with **keys greater than** the node's key.
  - Both the left and right subtrees must also be binary search trees.

```
_isBST(self, node, min, max)
```



isBST() - Validate Binary Search Tree

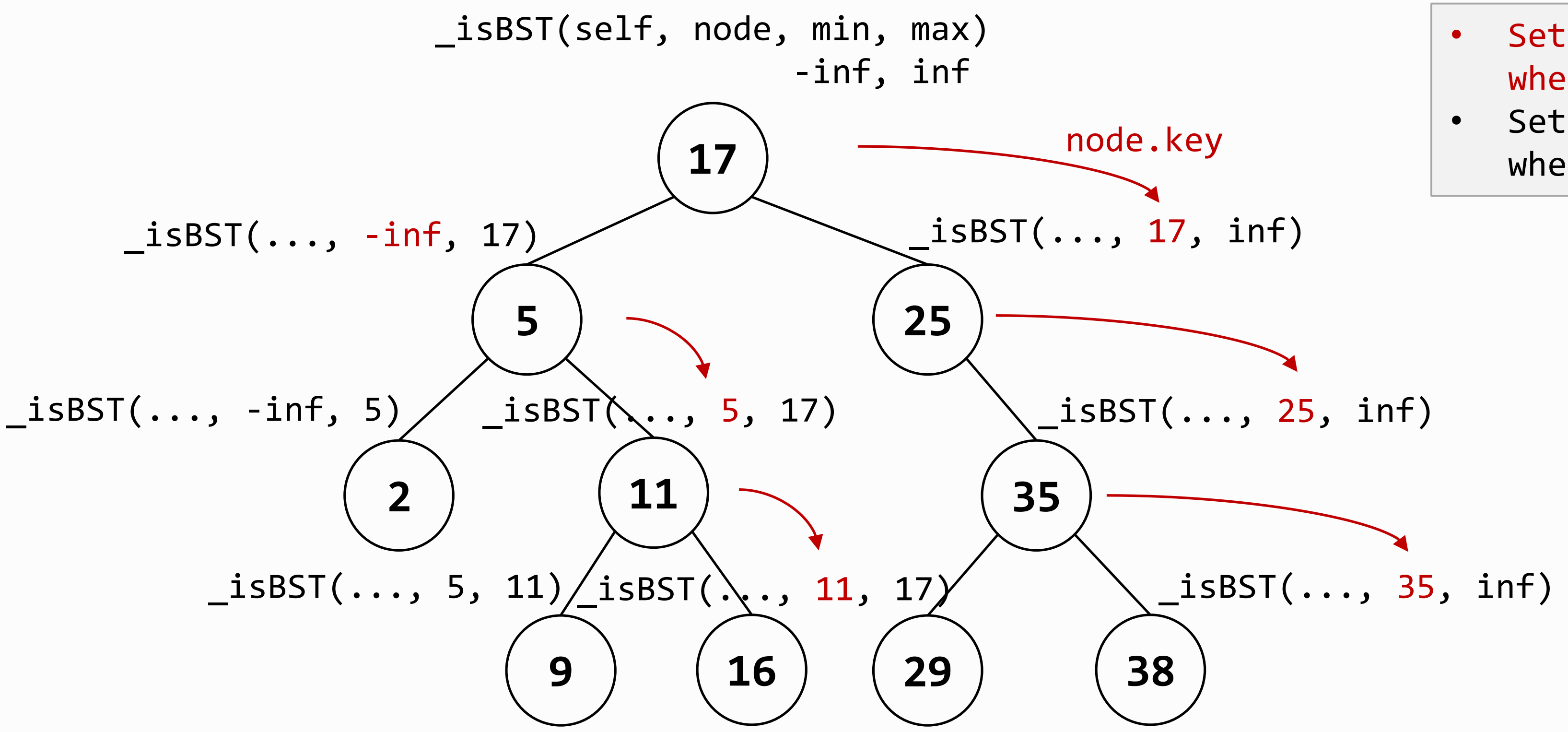
- A BST is defined as follows:
  - The left subtree of a node contains only nodes with **keys less than** the node's key.
  - The right subtree of a node contains only nodes with **keys greater than** the node's key.
  - Both the left and right subtrees must also be binary search trees.




```
def isBST(self, node = None):
    if node is None: node = self.root
    return self._isBST(node, float('-inf'), float('inf'))

def _isBST(self, root, min, max):
```

isBST() - Validate Binary Search Tree

- A BST is defined as follows:
  - The left subtree of a node contains only nodes with **keys less than** the node's key.
  - The right subtree of a node contains only nodes with **keys greater than** the node's key.
  - Both the left and right subtrees must also be binary search trees.



- Set 'min' with node.key when move to the right.
- Set 'max' with node.key when move to the left.

Summary

- To delete a node in a BST, we must consider three different cases.
  - no child
  - one child
  - two children
- Both predecessor and successor at a node in a BST always has only one child or none, never two children.

Data Structures in Python
Chapter 7 - 2

- Binary Search Tree(BST)
- **BST Algorithms**
- AVL Tree
- AVL Algorithms

Prof. Youngsup Kim, idebtor@gmail.com, CSEE Dept., Grace School Rm204 Handong Global University