Data Structures and Algorithms
Binary Search Tree and Heap

2022-08-19

Ping-Han Hsieh

Overview

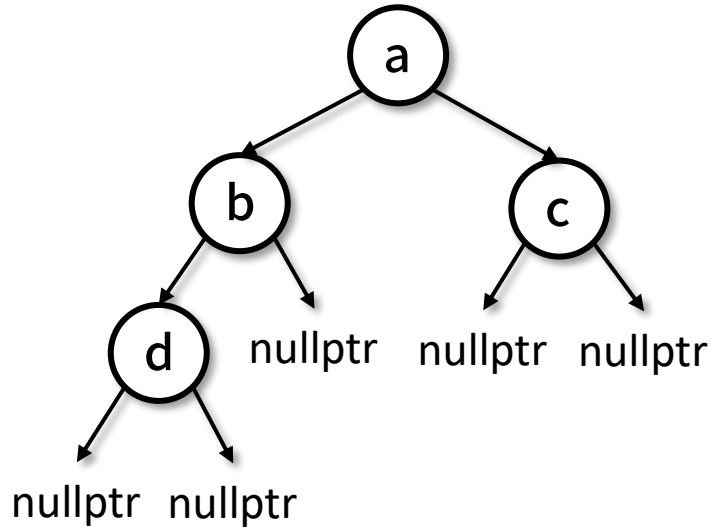
- Data Structures:
  - Linked-List, Array
  - Stack, Queue
  - Union Find, Hash Table
  - Binary Search Tree, Heap
  - Graph

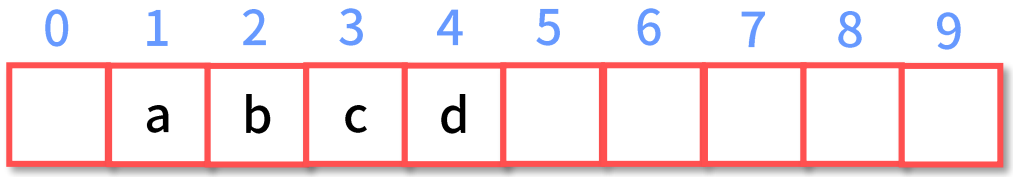
- Algorithms
  - Big-O Notation
  - Sorting
  - Graph Algorithms
  - Dynamic Programming

Binary Tree

Binary Tree (1)

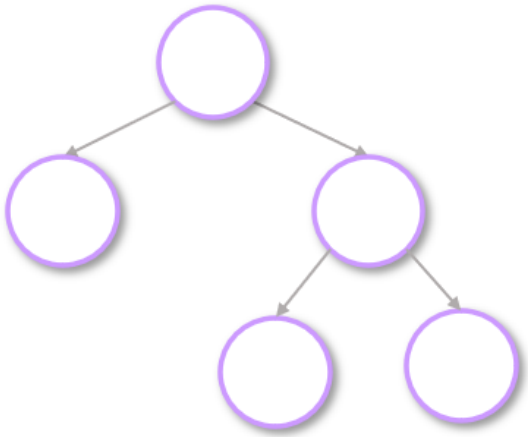
- Each parent node can have at most 2 children node.
- The top node is called the root node.
- The node without children is called the leaf node.
- Can be implemented using linked-list or array.





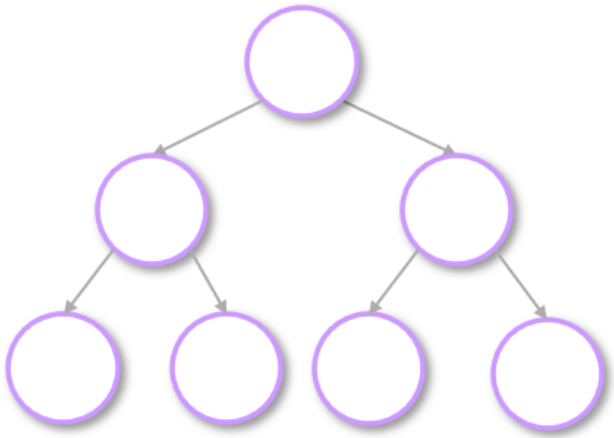
Binary Tree (2)

Full



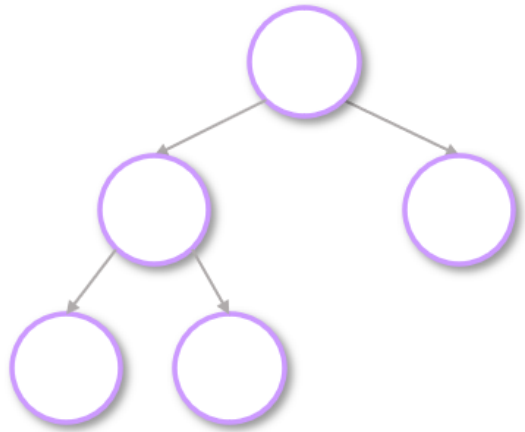
- all nodes have 0 or 2 children

Perfect



- all interior nodes have 2 children
- all leaves on the same level

Complete

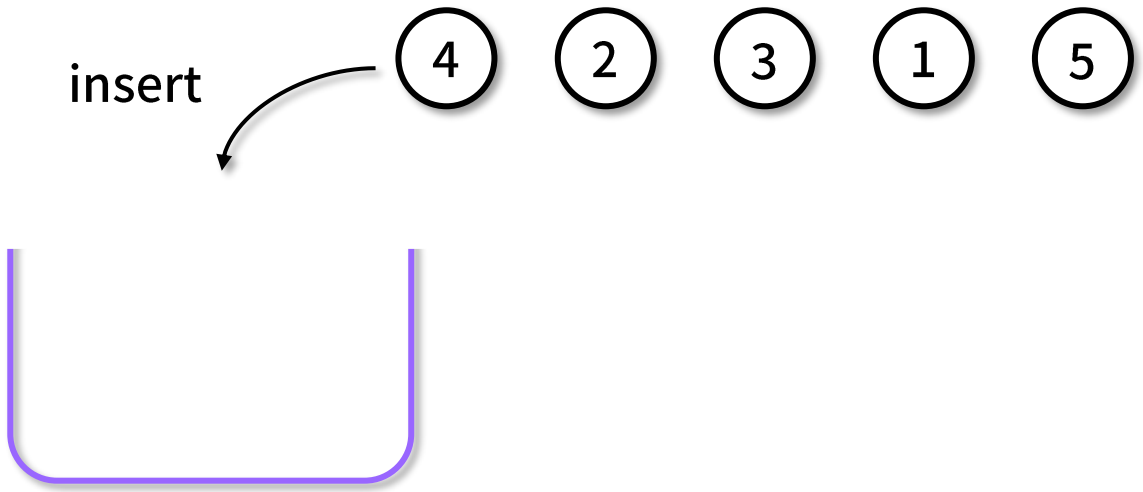


- all interior nodes (exclude last layer) is perfect
- all leaves are pushed to the left

Heap

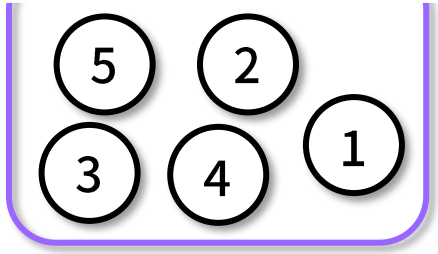
Minimum Heap (1)

- Always pop the smallest element.



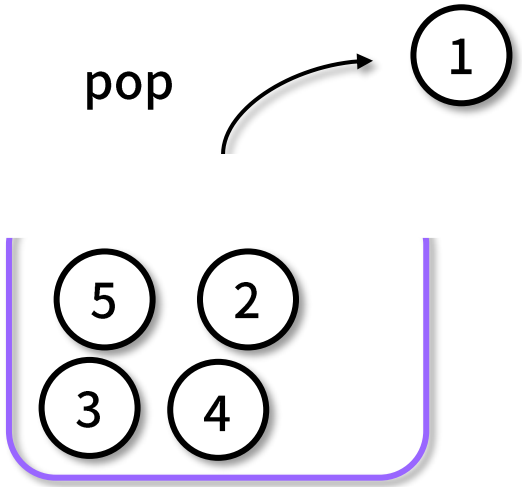
Minimum Heap (2)

- Always pop the smallest element.



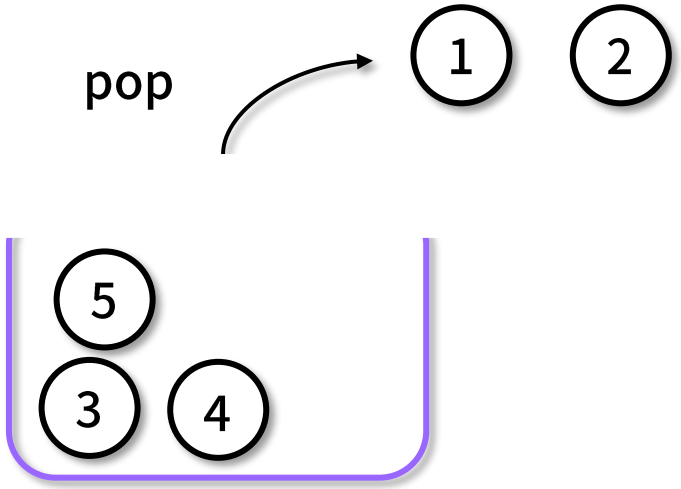
Minimum Heap (2)

- Always pop the smallest element.



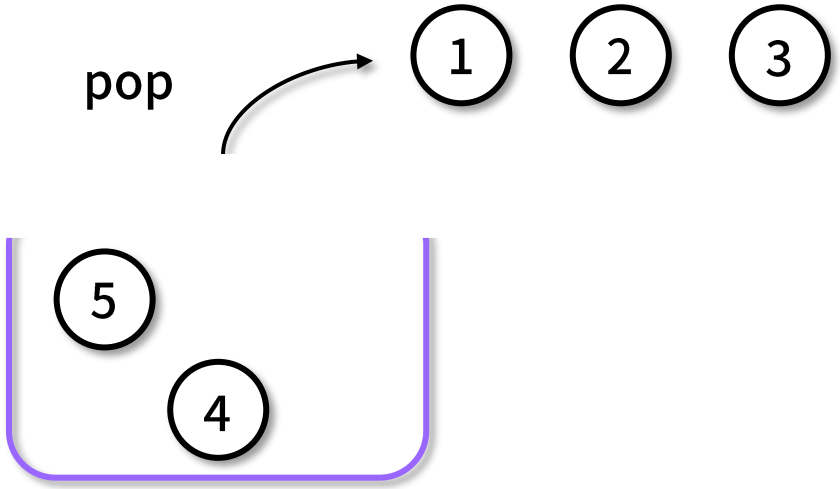
Minimum Heap (3)

- Always pop the smallest element.



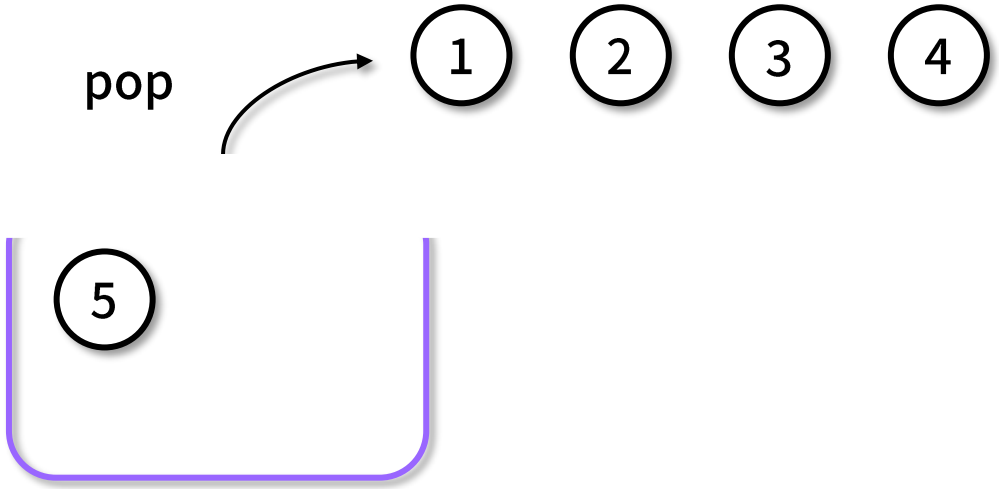
Minimum Heap (4)

- Always pop the smallest element.



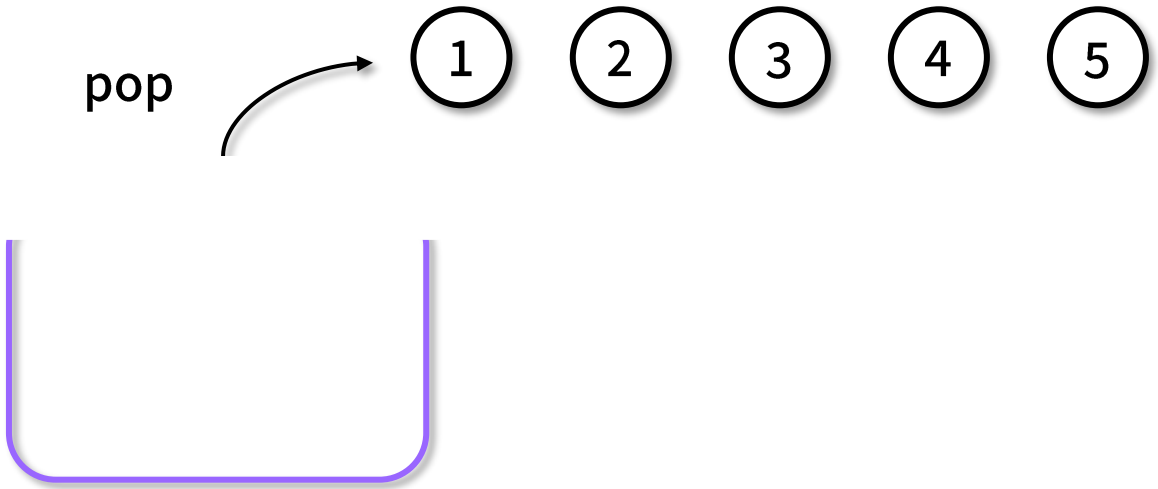
Minimum Heap (5)

- Always pop the smallest element.



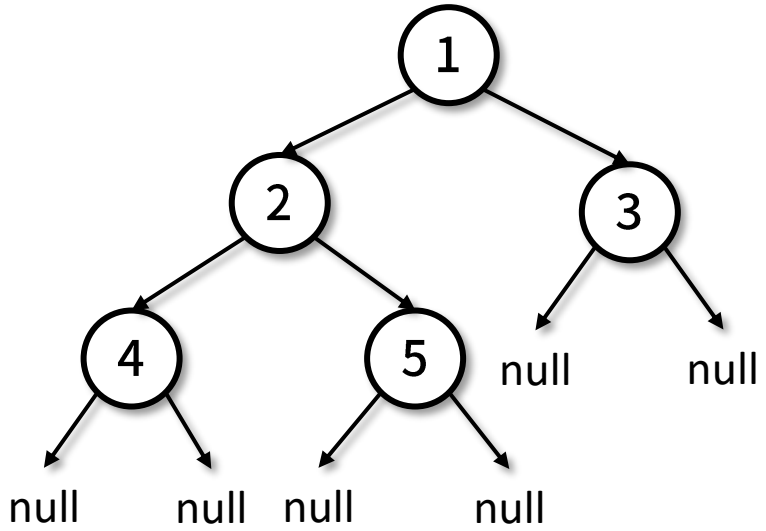
Minimum Heap (6)

- Always pop the smallest element.



Minimum Heap (7)

- Use a binary tree to implement the heap.
- The key of the parent node cannot be larger than the key of the children node.
- Need additional operation for *insert* and *pop* to maintain the structure to ensure the first rule is invariant.

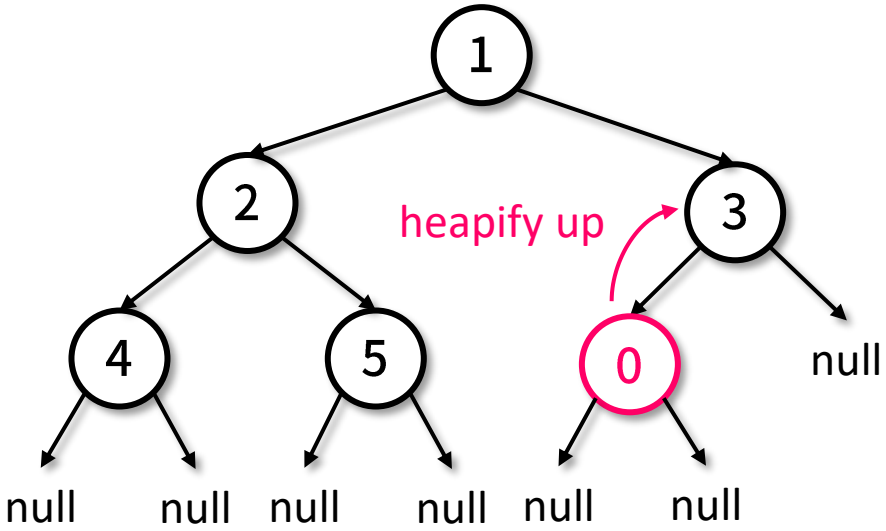


Minimum Heap

Insert (1)

1. Insert to the next empty leaf node (e.g. insert node with key 0).

2. Swap the inserted node up to the tree until smaller item is encountered (heapify up).

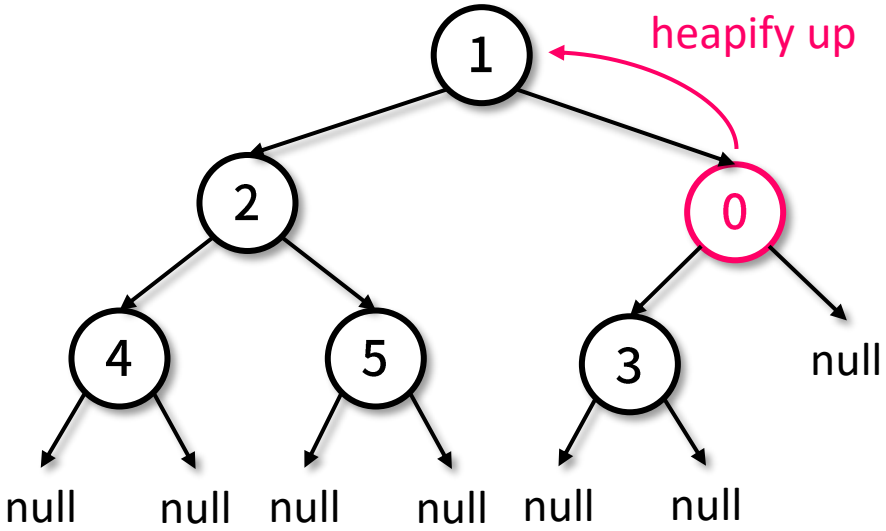


Minimum Heap

Insert (2)

1. Insert to the next empty leaf node (e.g. insert node with key 0).

2. Swap the inserted node up to the tree until smaller item is encountered (heapify up).

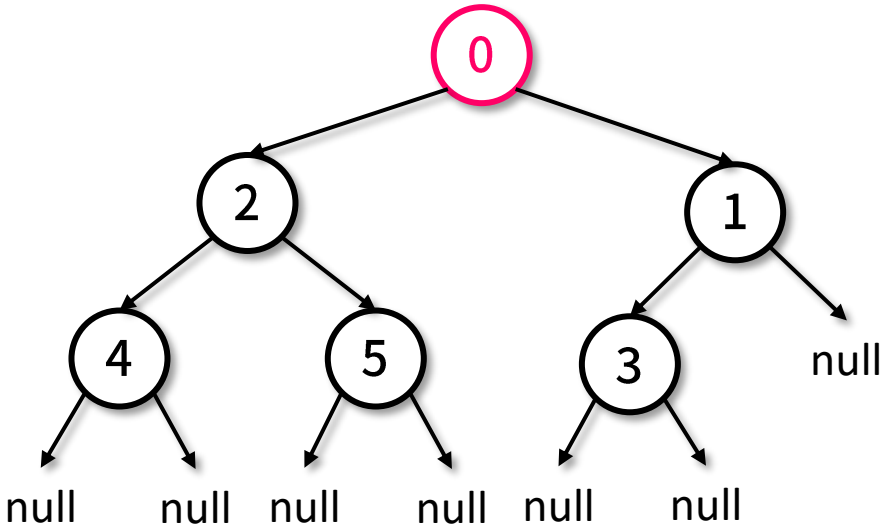


Minimum Heap

Insert (3)

1. Insert to the next empty leaf node (e.g. insert node with key 0).

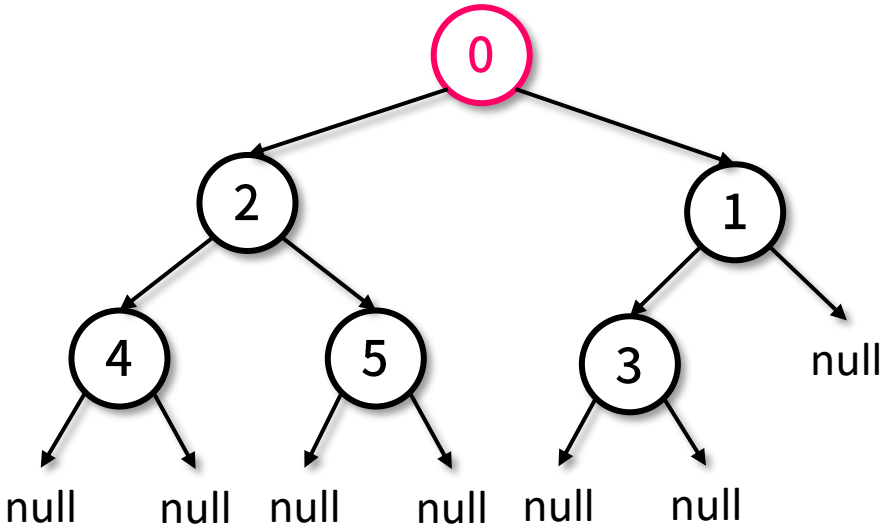
2. Swap the inserted node up to the tree until smaller item is encountered (heapify up).



Minimum Heap

Pop (1)

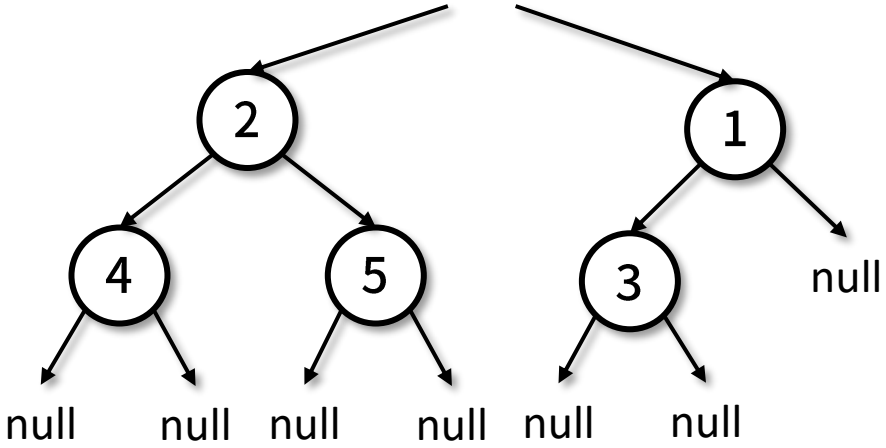
- The value to be return is the value of the root node.



Minimum Heap

Pop (2)

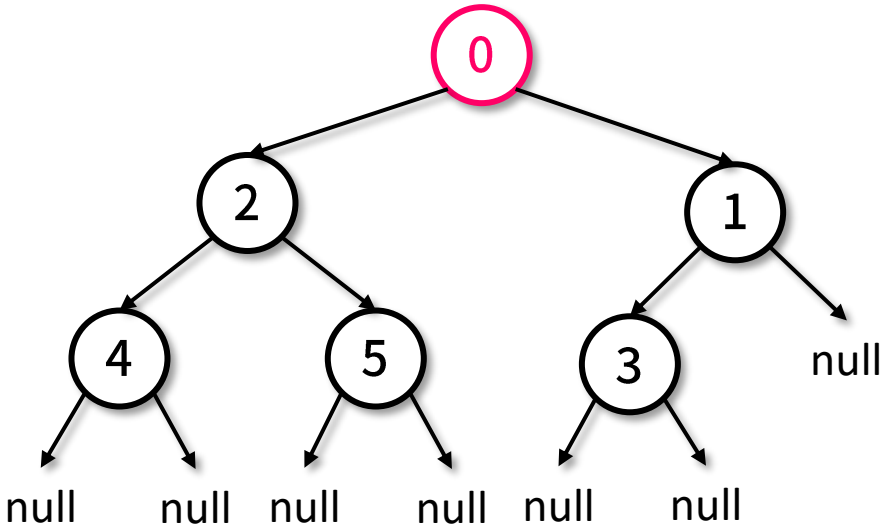
- The value to be return is the value of the root node.
- Simply removing the node will break the binary tree.



Minimum Heap

Pop (3)

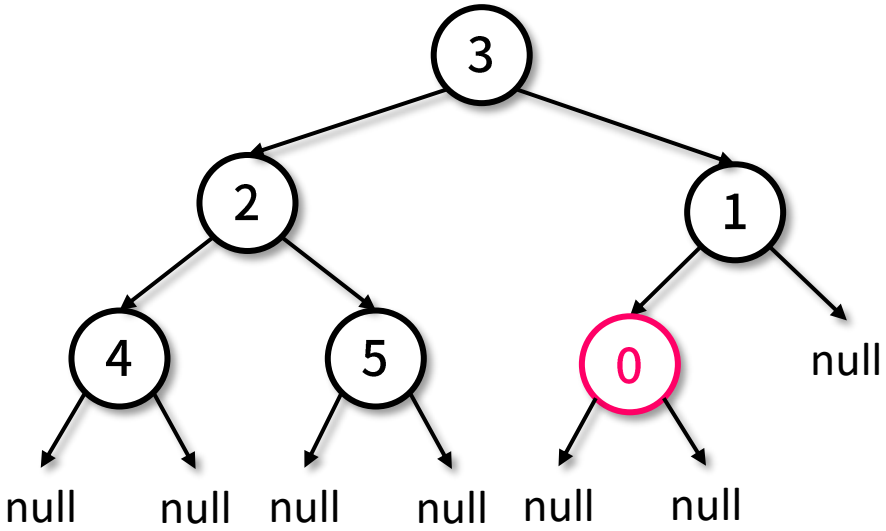
- The value to be return is the value of the root node.
- Simply removing the node will break the binary tree.
- Instead, we swap with the last node (relatively large) and pop it.



Minimum Heap

Pop (4)

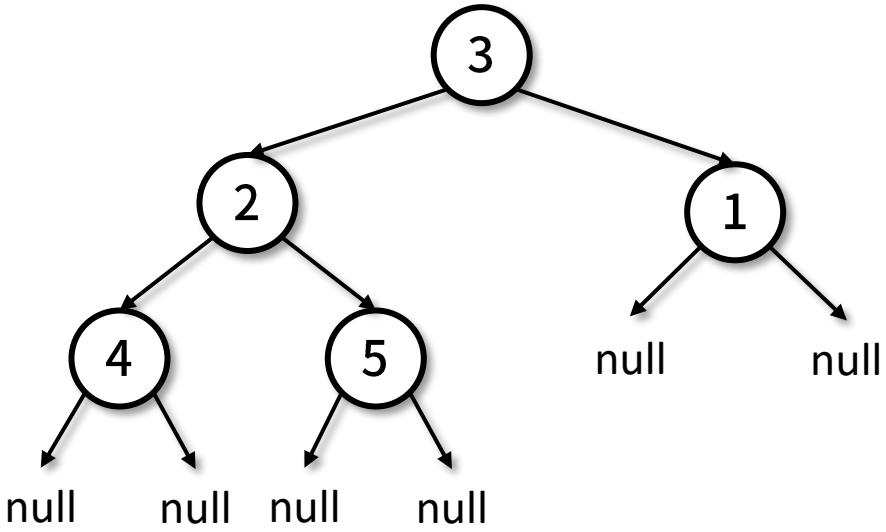
- The value to be return is the value of the root node.
- Simply removing the node will break the binary tree.
- Instead, we swap with the last node (relatively large) and pop it.



Minimum Heap

Pop (5)

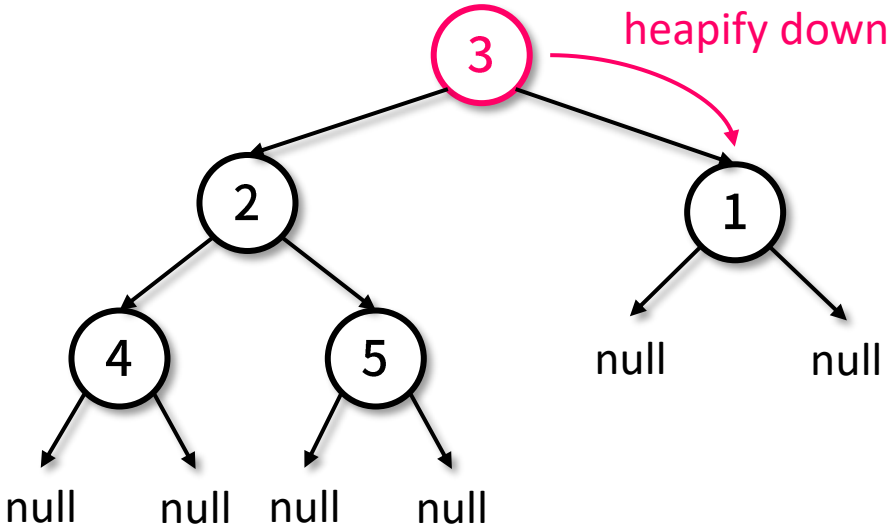
- The value to be return is the value of the root node.
- Simply removing the node will break the binary tree.
- Instead, we swap with the last node (relatively large) and pop it.
- Then, we swap the root node down to the tree until larger node is encountered or reach the leaf (heapify down).



Minimum Heap

Pop (6)

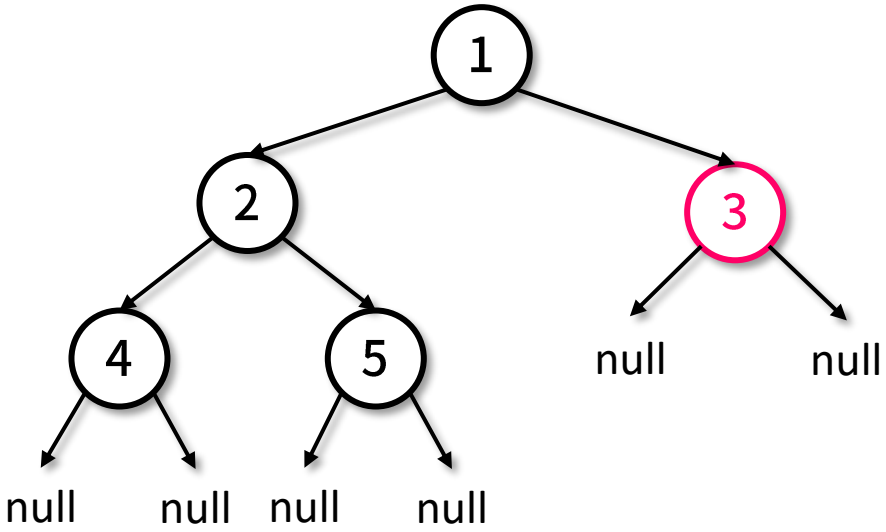
- The value to be return is the value of the root node.
- Simply removing the node will break the binary tree.
- Instead, we swap with the last node (relatively large) and pop it.
- Then, we swap the root node down to the tree until larger node is encountered or reach the leaf (heapify down).



Minimum Heap

Pop (6)

- The value to be return is the value of the root node.
- Simply removing the node will break the binary tree.
- Instead, we swap with the last node (relatively large) and pop it.
- Then, we swap the root node down to the tree until larger node is encountered or reach the leaf (heapify down).



Minimum Heap (8)

Initialization

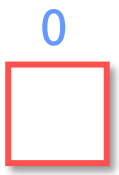
```
class MinHeap(Iterable):
  def __init__(self):
    self.data: Array = Array(1)
    self.n = 0

  def __len__(self) -> int: ···

  def __iter__(self) -> None: ···

  def insert(self, x: ComparableT) -> Iterator: ···
      insert an item
  def peek(self) -> ComparableT: ···
      check what is the minimum value in the Heap
  def pop(self) -> ComparableT: ···
      return and remove the minimum value in the Heap
  def heapify_up(self, index: int) -> None: ···
      reorganize the  Heap after insertion
  def heapify_down(self, index: int) -> None: ···
      reorganize the  Heap after deletion
```

Command

```
heap = MinHeap()
```

Visualization

n = 0



Minimum Heap

Insert (1)

Insertion

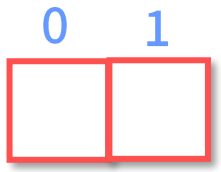
```python
def insert(self, x: ComparableT) -> None:
  if self.n == len(self.data) - 1:
    self.data.resize(2 * len(self.data))

  self.n += 1
  self.data[self.n] = x
  self.heapify_up(self.n)
  return
def heapify_up(self, index: int) -> None:
  parent_index = index // 2
  while index > 1 and self.data[parent_index] > self.data[index]:
    self.data.exchange(index, parent_index)
    index = parent_index
    parent_index = index // 2
  return
```

Command

```
heap.insert(4)
```

Visualization

n = 1



Tree Visualization

Minimum Heap

Insert (2)

Insertion

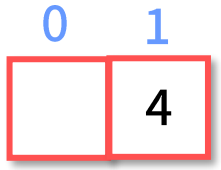
```
def insert(self, x: ComparableT) -> None:
  if self.n == len(self.data) - 1:
    self.data.resize(2 * len(self.data))

  self.n += 1
  self.data[self.n] = x
  self.heapify_up(self.n)
  return
def heapify_up(self, index: int) -> None:
  parent_index = index // 2
  while index > 1 and self.data[parent_index] > self.data[index]:
    self.data.exchange(index, parent_index)
    index = parent_index
    parent_index = index // 2
  return
```

Command

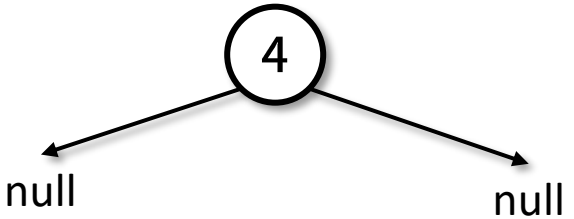
```
heap.insert(4)
```

Visualization

n = 1



Tree Visualization



Minimum Heap

Insert (3)

Insertion

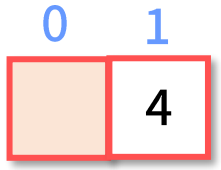
```
def insert(self, x: ComparableT) -> None:
  if self.n == len(self.data) - 1:
    self.data.resize(2 * len(self.data))

  self.n += 1
  self.data[self.n] = x
  self.heapify_up(self.n)
  return
def heapify_up(self, index: int) -> None:
  parent_index = index // 2
  while index > 1 and self.data[parent_index] > self.data[index]:
    self.data.exchange(index, parent_index)
    index = parent_index
    parent_index = index // 2
  return
```

Command

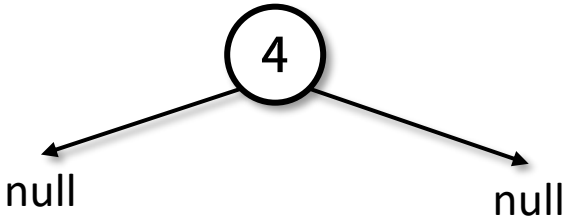
```
heap.insert(4)
```

Visualization

n = 1



Tree Visualization



Minimum Heap

Insert (4)

Insertion

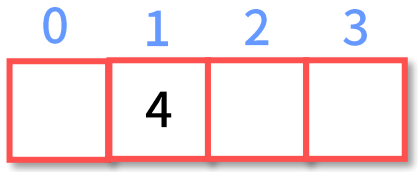
```
def insert(self, x: ComparableT) -> None:
  if self.n == len(self.data) - 1:
    self.data.resize(2 * len(self.data))

  self.n += 1
  self.data[self.n] = x
  self.heapify_up(self.n)
  return
def heapify_up(self, index: int) -> None:
  parent_index = index // 2
  while index > 1 and self.data[parent_index] > self.data[index]:
    self.data.exchange(index, parent_index)
    index = parent_index
    parent_index = index // 2
  return
```

Command

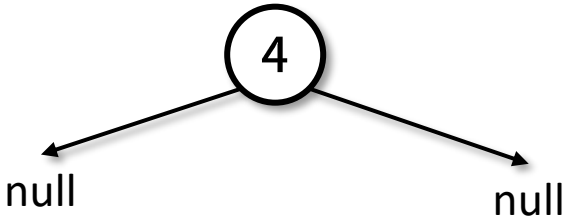
```
heap.insert(2)
```

Visualization

n = 1



Tree Visualization



Minimum Heap

Insert (5)

Insertion

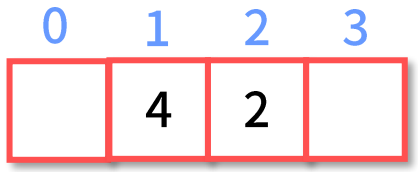
```
def insert(self, x: ComparableT) -> None:
  if self.n == len(self.data) - 1:
    self.data.resize(2 * len(self.data))

  self.n += 1
  self.data[self.n] = x
  self.heapify_up(self.n)
  return
def heapify_up(self, index: int) -> None:
  parent_index = index // 2
  while index > 1 and self.data[parent_index] > self.data[index]:
    self.data.exchange(index, parent_index)
    index = parent_index
    parent_index = index // 2
  return
```

Command

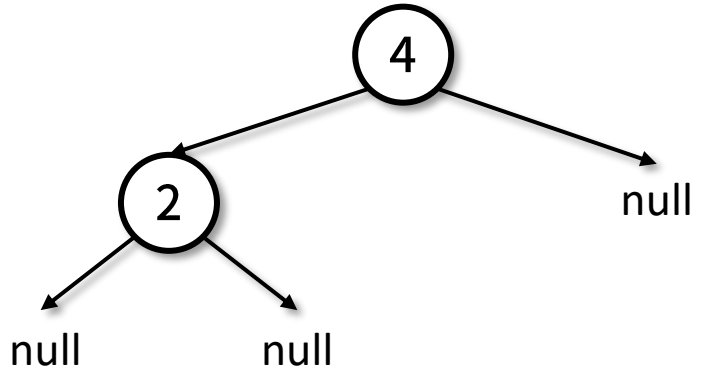
```
heap.insert(2)
```

Visualization

n = 2



Tree Visualization



Minimum Heap

Insert (6)

Insertion

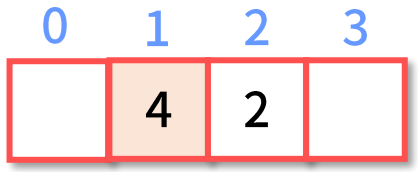
```
def insert(self, x: ComparableT) -> None:
  if self.n == len(self.data) - 1:
    self.data.resize(2 * len(self.data))

  self.n += 1
  self.data[self.n] = x
  self.heapify_up(self.n)
  return
def heapify_up(self, index: int) -> None:
  parent_index = index // 2
  while index > 1 and self.data[parent_index] > self.data[index]:
    self.data.exchange(index, parent_index)
    index = parent_index
    parent_index = index // 2
  return
```

Command

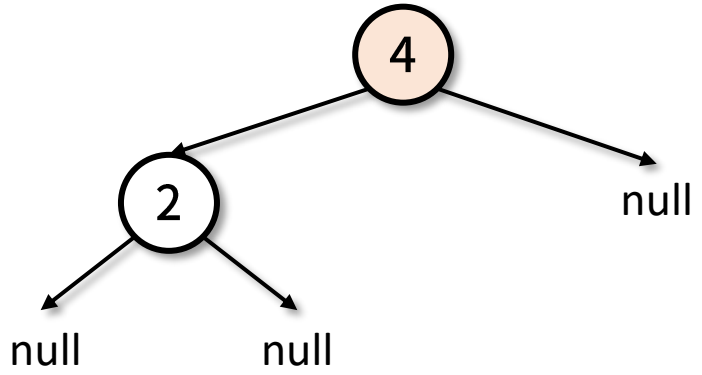
```
heap.insert(4)
```

Visualization

n = 2



Tree Visualization



Minimum Heap

Insert (7)

Insertion

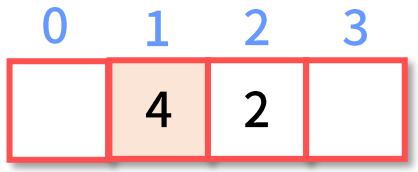
```
def insert(self, x: ComparableT) -> None:
  if self.n == len(self.data) - 1:
    self.data.resize(2 * len(self.data))

  self.n += 1
  self.data[self.n] = x
  self.heapify_up(self.n)
  return
def heapify_up(self, index: int) -> None:
  parent_index = index // 2
  while index > 1 and self.data[parent_index] > self.data[index]:
    self.data.exchange(index, parent_index)
    index = parent_index
    parent_index = index // 2
  return
```

Command

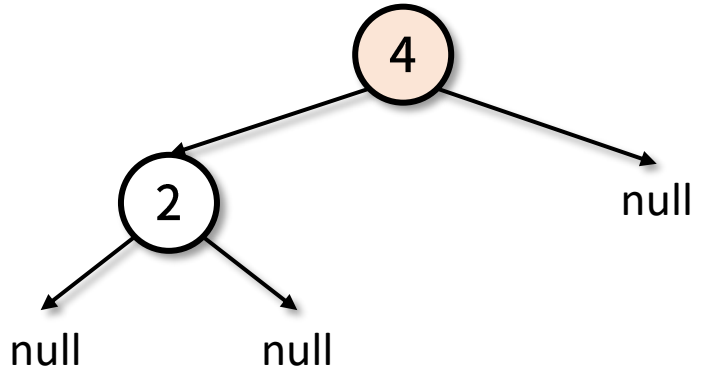
```
heap.insert(4)
```

Visualization

n = 2



Tree Visualization



Minimum Heap

Insert (8)

Insertion

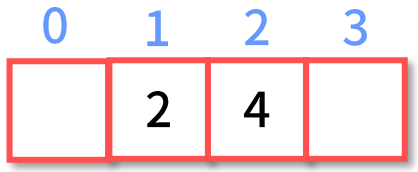
```
def insert(self, x: ComparableT) -> None:
  if self.n == len(self.data) - 1:
    self.data.resize(2 * len(self.data))

  self.n += 1
  self.data[self.n] = x
  self.heapify_up(self.n)
  return
def heapify_up(self, index: int) -> None:
  parent_index = index // 2
  while index > 1 and self.data[parent_index] > self.data[index]:
    self.data.exchange(index, parent_index)
    index = parent_index
    parent_index = index // 2
  return
```

Command

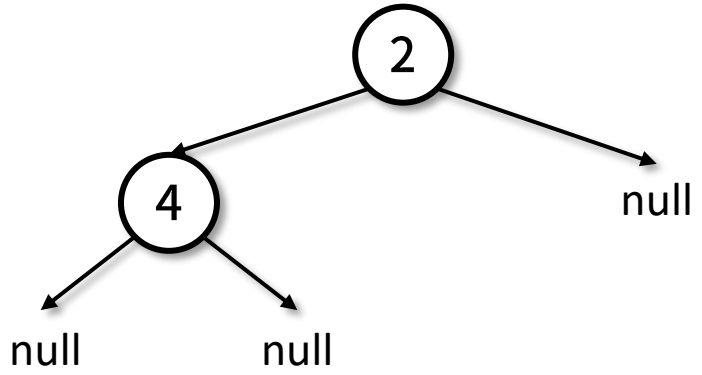
```
heap.insert(4)
```

Visualization

n = 2



Tree Visualization



Minimum Heap

Insert (9)

Insertion

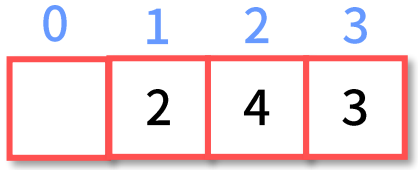
```
def insert(self, x: ComparableT) -> None:
  if self.n == len(self.data) - 1:
    self.data.resize(2 * len(self.data))

  self.n += 1
  self.data[self.n] = x
  self.heapify_up(self.n)
  return
def heapify_up(self, index: int) -> None:
  parent_index = index // 2
  while index > 1 and self.data[parent_index] > self.data[index]:
    self.data.exchange(index, parent_index)
    index = parent_index
    parent_index = index // 2
  return
```

Command

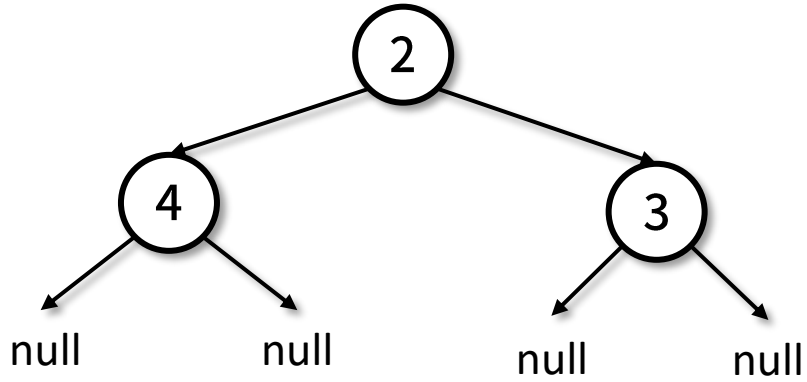
```
heap.insert(3)
heap.insert(1)
```

Visualization

n = 3



Tree Visualization



Minimum Heap

Insert (10)

Insertion

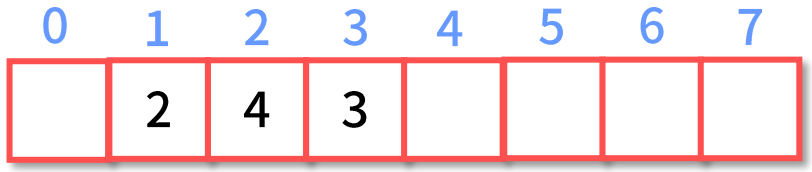
```
def insert(self, x: ComparableT) -> None:
  if self.n == len(self.data) - 1:
    self.data.resize(2 * len(self.data))

  self.n += 1
  self.data[self.n] = x
  self.heapify_up(self.n)
  return
def heapify_up(self, index: int) -> None:
  parent_index = index // 2
  while index > 1 and self.data[parent_index] > self.data[index]:
    self.data.exchange(index, parent_index)
    index = parent_index
    parent_index = index // 2
  return
```

Command

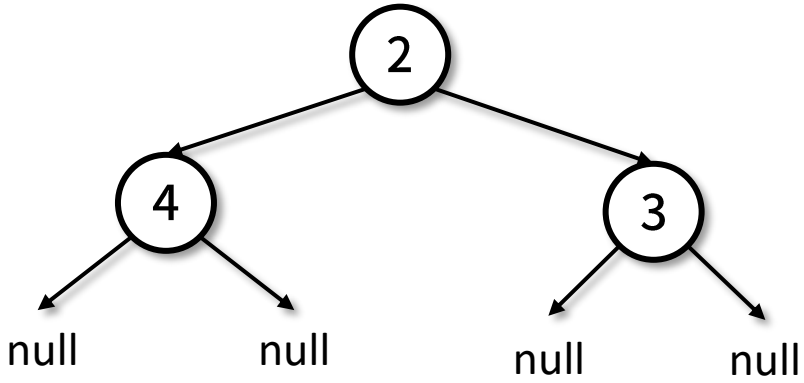
```
heap.insert(3)
heap.insert(1)
```

Visualization

n = 3



Tree Visualization



Minimum Heap

Insert (11)

Insertion

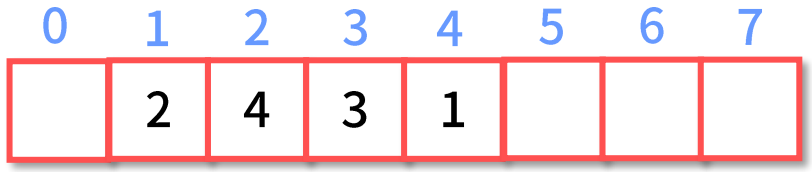
```
def insert(self, x: ComparableT) -> None:
  if self.n == len(self.data) - 1:
    self.data.resize(2 * len(self.data))

  self.n += 1
  self.data[self.n] = x
  self.heapify_up(self.n)
  return
def heapify_up(self, index: int) -> None:
  parent_index = index // 2
  while index > 1 and self.data[parent_index] > self.data[index]:
    self.data.exchange(index, parent_index)
    index = parent_index
    parent_index = index // 2
  return
```

Command

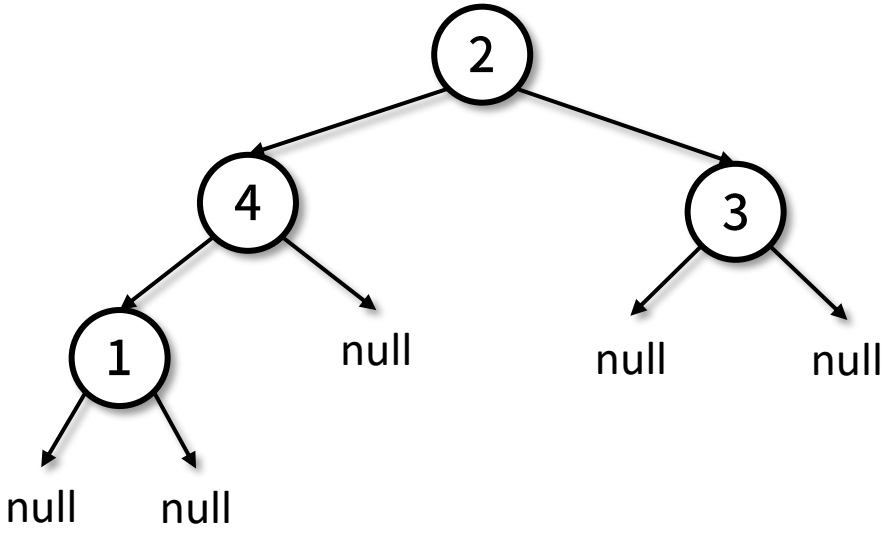
```
heap.insert(3)
heap.insert(1)
```

Visualization

n = 4



Tree Visualization



Minimum Heap

Insert (12)

Insertion

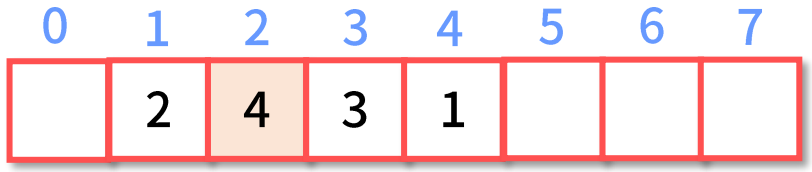
```
def insert(self, x: ComparableT) -> None:
  if self.n == len(self.data) - 1:
    self.data.resize(2 * len(self.data))

  self.n += 1
  self.data[self.n] = x
  self.heapify_up(self.n)
  return
def heapify_up(self, index: int) -> None:
  parent_index = index // 2
  while index > 1 and self.data[parent_index] > self.data[index]:
    self.data.exchange(index, parent_index)
    index = parent_index
    parent_index = index // 2
  return
```

Command

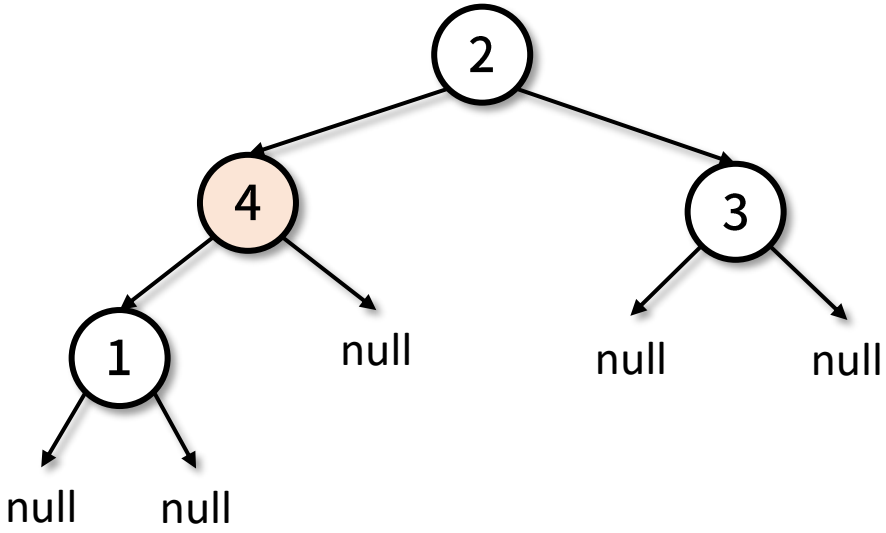
```
heap.insert(3)
heap.insert(1)
```

Visualization

n = 4



Tree Visualization



Minimum Heap

Insert (13)

Insertion

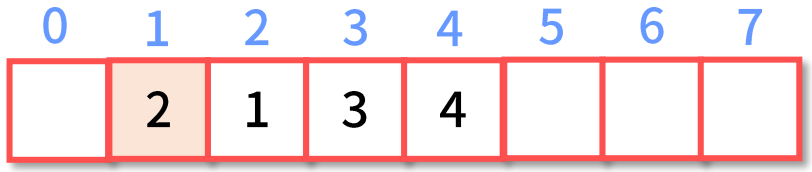
```
def insert(self, x: ComparableT) -> None:
  if self.n == len(self.data) - 1:
    self.data.resize(2 * len(self.data))

  self.n += 1
  self.data[self.n] = x
  self.heapify_up(self.n)
  return
def heapify_up(self, index: int) -> None:
  parent_index = index // 2
  while index > 1 and self.data[parent_index] > self.data[index]:
    self.data.exchange(index, parent_index)
    index = parent_index
    parent_index = index // 2
  return
```

Command

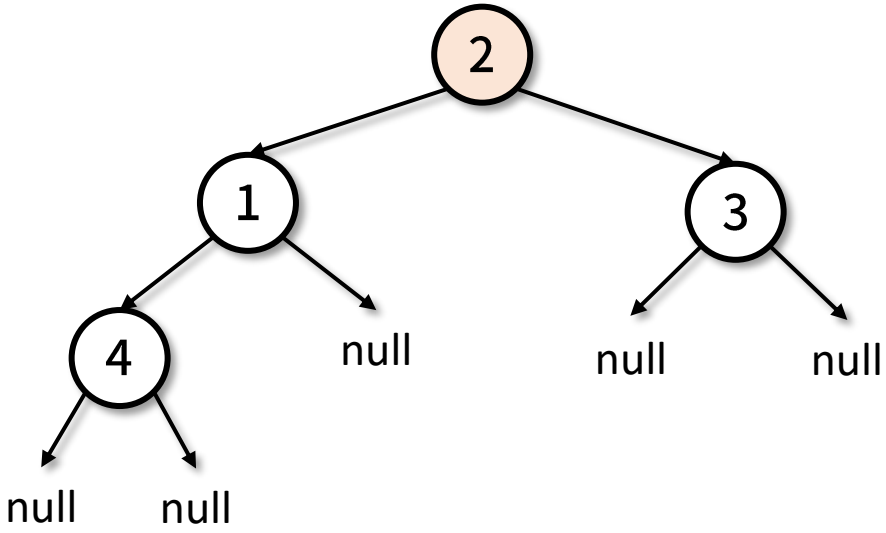
```
heap.insert(3)
heap.insert(1)
```

Visualization

n = 4



Tree Visualization



Minimum Heap

Insert (14)

Insertion

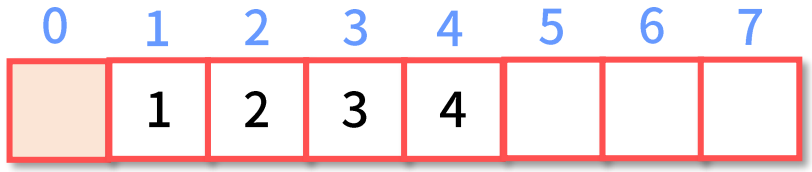
```
def insert(self, x: ComparableT) -> None:
  if self.n == len(self.data) - 1:
    self.data.resize(2 * len(self.data))

  self.n += 1
  self.data[self.n] = x
  self.heapify_up(self.n)
  return
def heapify_up(self, index: int) -> None:
  parent_index = index // 2
  while index > 1 and self.data[parent_index] > self.data[index]:
    self.data.exchange(index, parent_index)
    index = parent_index
    parent_index = index // 2
  return
```

Command

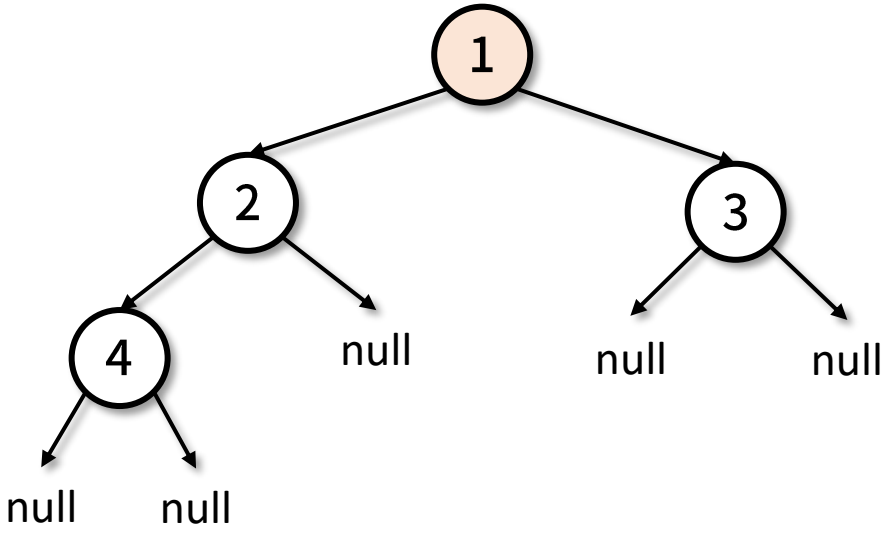
```
heap.insert(3)
heap.insert(1)
```

Visualization

n = 4



Tree Visualization



Minimum Heap

Insert (15)

Insertion

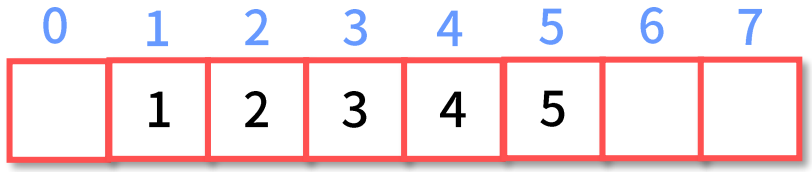
```
def insert(self, x: ComparableT) -> None:
  if self.n == len(self.data) - 1:
    self.data.resize(2 * len(self.data))

  self.n += 1
  self.data[self.n] = x
  self.heapify_up(self.n)
  return
def heapify_up(self, index: int) -> None:
  parent_index = index // 2
  while index > 1 and self.data[parent_index] > self.data[index]:
    self.data.exchange(index, parent_index)
    index = parent_index
    parent_index = index // 2
  return
```

Command

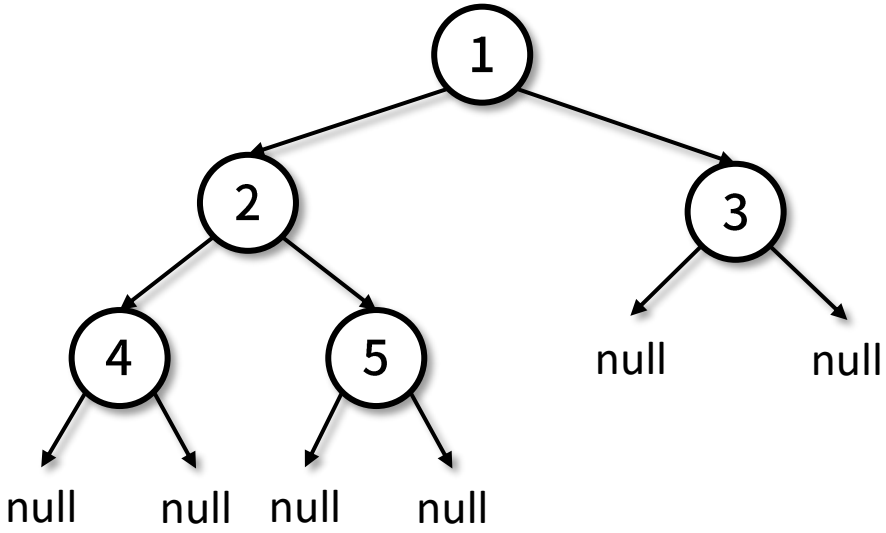
```
heap.insert(5)
```

Visualization

n = 5



Tree Visualization



Minimum Heap

Pop (1)

Pop

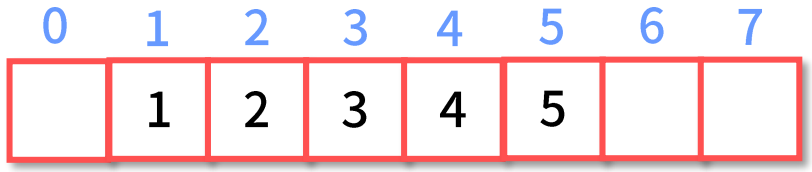
```
def pop(self) -> ComparableT:
    result = self.data[1]
    self.data.exchange(1, self.n)
    self.n -= 1
    self.heapify_down(1)
    self.data[self.n + 1] = None
    if self.n > 0 and self.n <= (len(self.data) - 1) / 4:
        self.data.resize(int(0.5 * len(self.data)))
    return result
```

Command

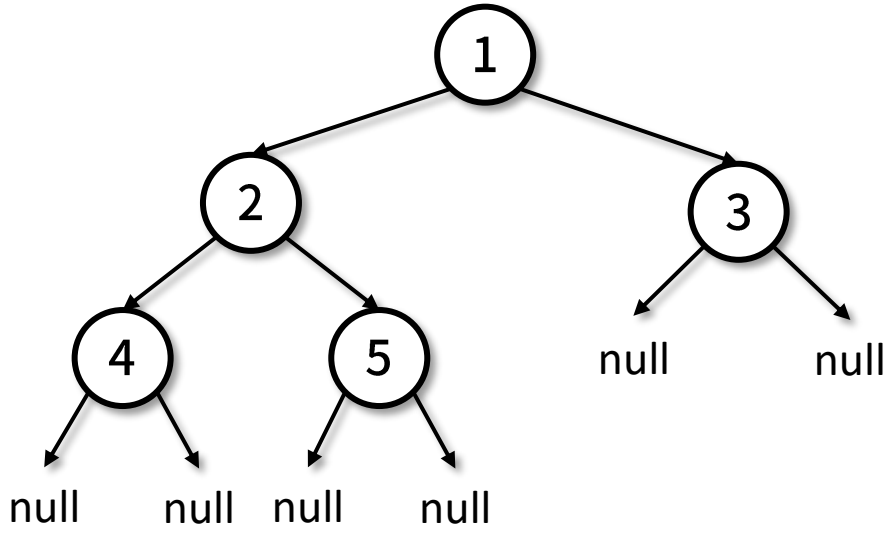
```
heap.pop()
```

Visualization

n = 5 result = 1



Tree Visualization



Minimum Heap

Pop (2)

Pop

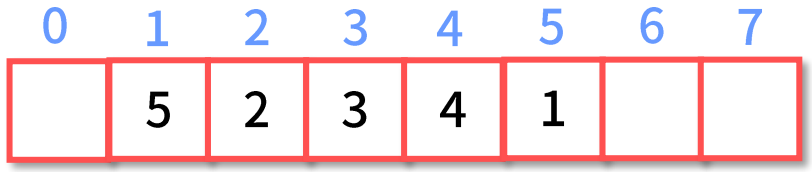
```
def pop(self) -> ComparableT:
  result = self.data[1]
  self.data.exchange(1, self.n)
  self.n -= 1
  self.heapify_down(1)
  self.data[self.n + 1] = None
  if self.n > 0 and self.n <= (len(self.data) - 1) / 4:
    self.data.resize(int(0.5 * len(self.data)))
  return result
```

Command

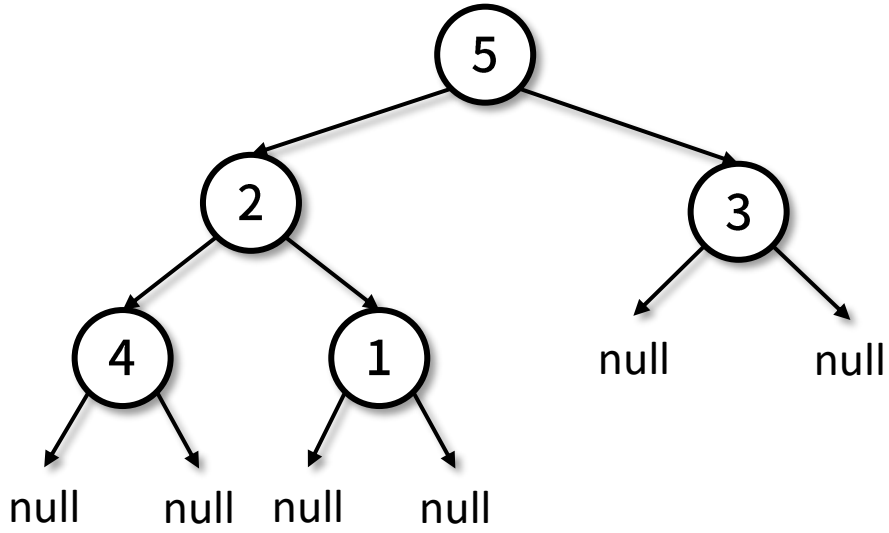
```
heap.pop()
```

Visualization

n = 5 result = 1



Tree Visualization



Minimum Heap

Pop (3)

Pop

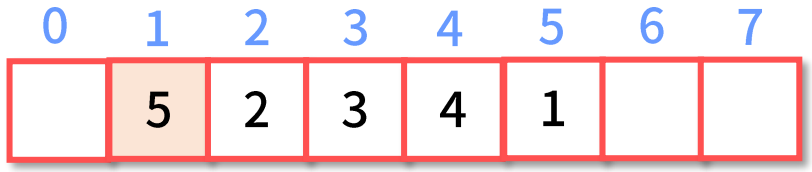
```
def pop(self) -> ComparableT:
    result = self.data[1]
    self.data.exchange(1, self.n)
    self.n -= 1
    self.heapify_down(1)
    self.data[self.n + 1] = None
    if self.n > 0 and self.n <= (len(self.data) - 1) / 4:
        self.data.resize(int(0.5 * len(self.data)))
    return result
```

Command

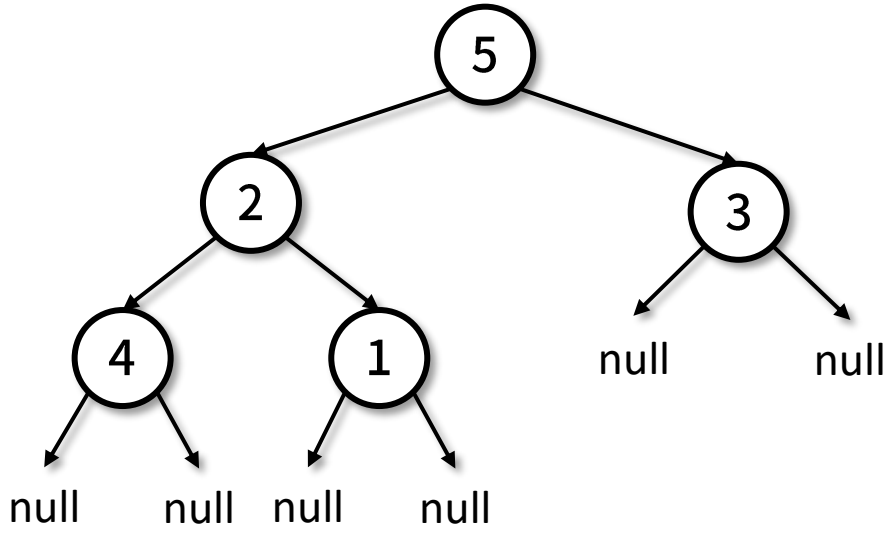
```
heap.pop()
```

Visualization

n = 4 result = 1



Tree Visualization



Minimum Heap

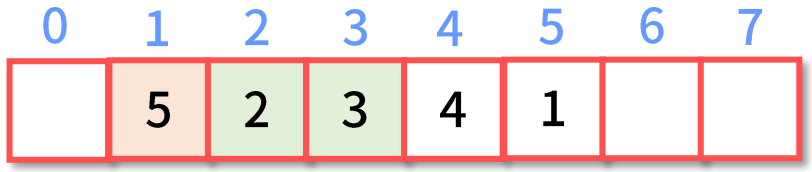
Pop (4)

Pop

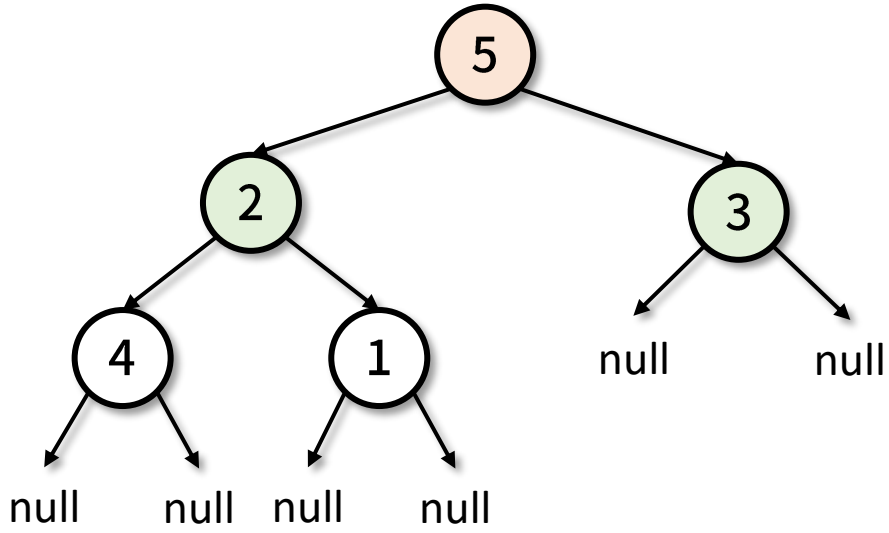
```python
def heapify_down(self, index: int) -> None:
    while 2 * index <= self.n:
        child_a_index = index * 2
        child_b_index = index * 2 + 1
        if child_b_index > self.n:
            exchange_to = child_a_index
        elif self.data[child_b_index] < self.data[child_a_index]:
            exchange_to = child_b_index
        else:
            exchange_to = child_a_index
        if self.data[index] <= self.data[exchange_to]:
            break
        self.data.exchange(index, exchange_to)
        index = exchange_to
    return
```

Visualization

n = 4 result = 1



Tree Visualization



Minimum Heap

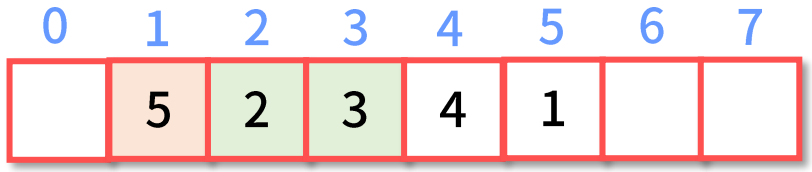
Pop (5)

Pop

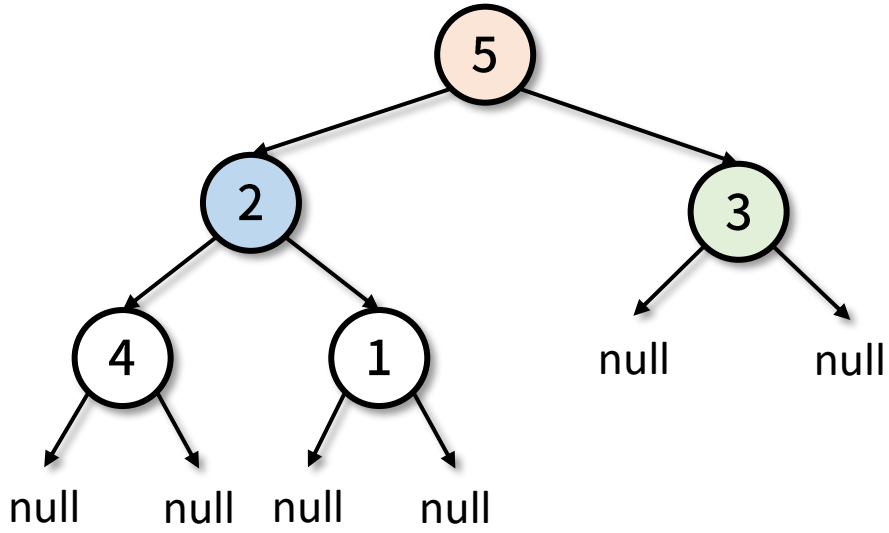
```python
def heapify_down(self, index: int) -> None:
    while 2 * index <= self.n:
        child_a_index = index * 2
        child_b_index = index * 2 + 1
        if child_b_index > self.n:
            exchange_to = child_a_index
        elif self.data[child_b_index] < self.data[child_a_index]:
            exchange_to = child_b_index
        else:
            exchange_to = child_a_index
        if self.data[index] <= self.data[exchange_to]:
            break
        self.data.exchange(index, exchange_to)
        index = exchange_to
    return
```

Visualization

n = 4 result = 1



Tree Visualization



Minimum Heap

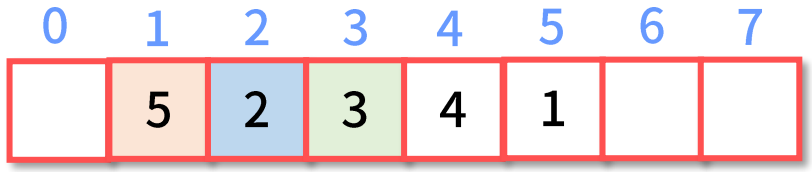
Pop (6)

Pop

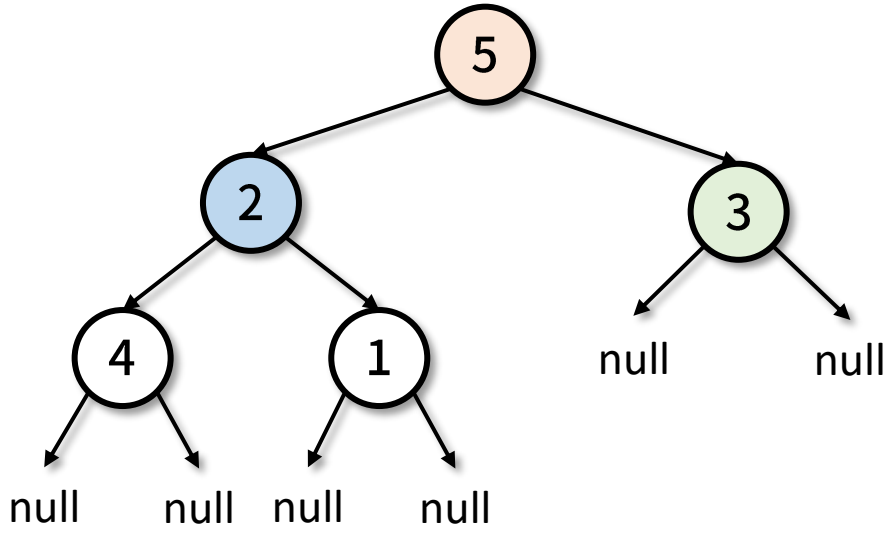
```python
def heapify_down(self, index: int) -> None:
    while 2 * index <= self.n:
        child_a_index = index * 2
        child_b_index = index * 2 + 1
        if child_b_index > self.n:
            exchange_to = child_a_index
        elif self.data[child_b_index] < self.data[child_a_index]:
            exchange_to = child_b_index
        else:
            exchange_to = child_a_index
        if self.data[index] <= self.data[exchange_to]:
            break
        self.data.exchange(index, exchange_to)
        index = exchange_to
    return
```

Visualization

n = 4 result = 1



Tree Visualization



Minimum Heap

Pop (7)

Pop

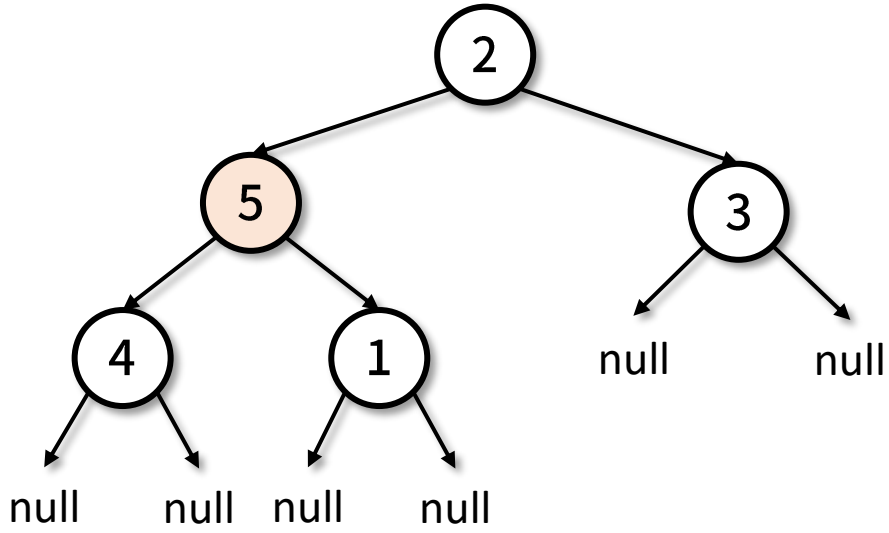
```python
def heapify_down(self, index: int) -> None:
    while 2 * index <= self.n:
        child_a_index = index * 2
        child_b_index = index * 2 + 1
        if child_b_index > self.n:
            exchange_to = child_a_index
        elif self.data[child_b_index] < self.data[child_a_index]:
            exchange_to = child_b_index
        else:
            exchange_to = child_a_index
        if self.data[index] <= self.data[exchange_to]:
            break
        self.data.exchange(index, exchange_to)
        index = exchange_to
    return
```

Visualization

n = 4 result = 1

0	1	2	3	4	5	6	7
	2	5	3	4	1		

Tree Visualization



Minimum Heap

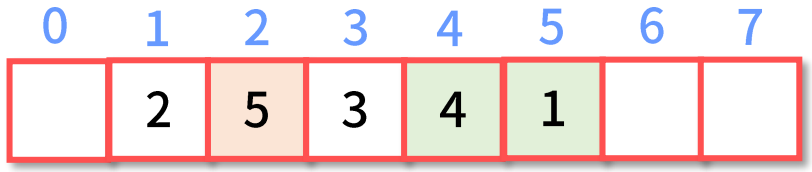
Pop (7)

Pop

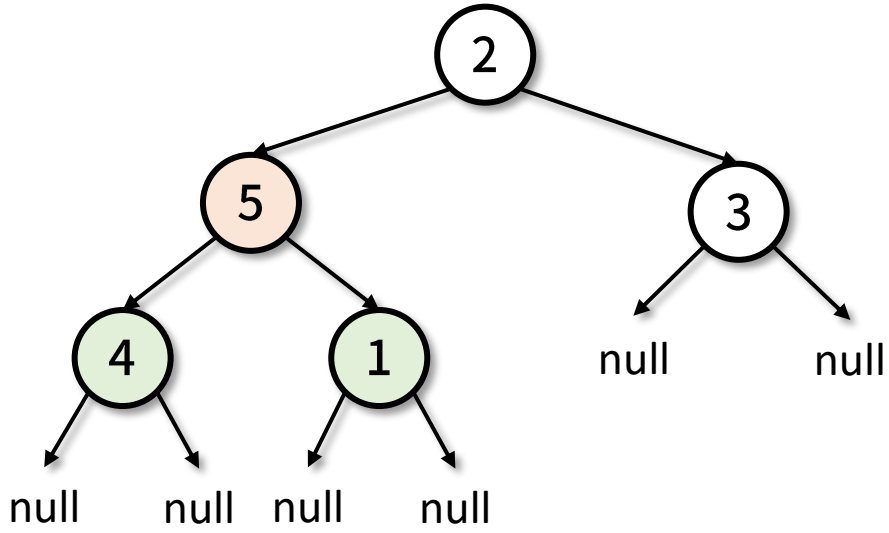
```python
def heapify_down(self, index: int) -> None:
    while 2 * index <= self.n:
        child_a_index = index * 2
        child_b_index = index * 2 + 1
        if child_b_index > self.n:
            exchange_to = child_a_index
        elif self.data[child_b_index] < self.data[child_a_index]:
            exchange_to = child_b_index
        else:
            exchange_to = child_a_index
        if self.data[index] <= self.data[exchange_to]:
            break
        self.data.exchange(index, exchange_to)
        index = exchange_to
    return
```

Visualization

n = 4 result = 1



Tree Visualization



Minimum Heap

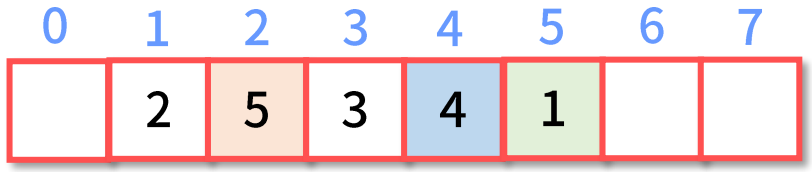
Pop (8)

Pop

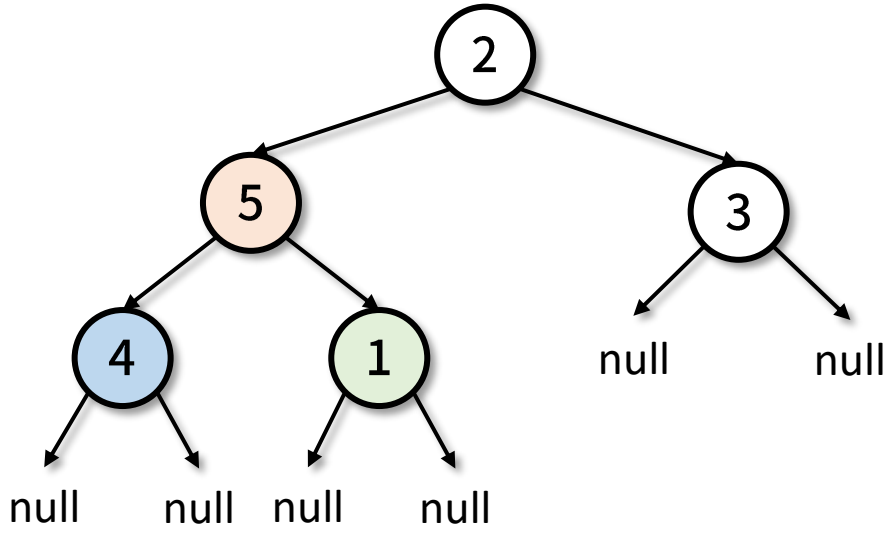
```python
def heapify_down(self, index: int) -> None:
    while 2 * index <= self.n:
        child_a_index = index * 2
        child_b_index = index * 2 + 1
        if child_b_index > self.n:
            exchange_to = child_a_index
        elif self.data[child_b_index] < self.data[child_a_index]:
            exchange_to = child_b_index
        else:
            exchange_to = child_a_index
        if self.data[index] <= self.data[exchange_to]:
            break
        self.data.exchange(index, exchange_to)
        index = exchange_to
    return
```

Visualization

n = 4 result = 1



Tree Visualization



Minimum Heap

Pop (9)

Pop

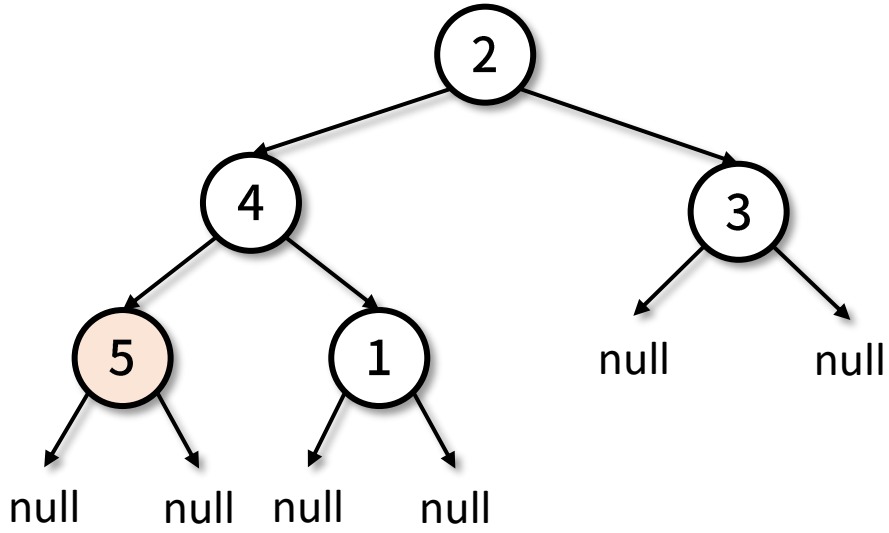
```python
def heapify_down(self, index: int) -> None:
  while 2 * index <= self.n:
    child_a_index = index * 2
    child_b_index = index * 2 + 1
    if child_b_index > self.n:
      exchange_to = child_a_index
    elif self.data[child_b_index] < self.data[child_a_index]:
      exchange_to = child_b_index
    else:
      exchange_to = child_a_index
    if self.data[index] <= self.data[exchange_to]:
      break
    self.data.exchange(index, exchange_to)
    index = exchange_to
  return
```

Visualization

n = 4 result = 1

0	1	2	3	4	5	6	7
	2	4	3	5	1		

Tree Visualization



Minimum Heap

Pop (10)

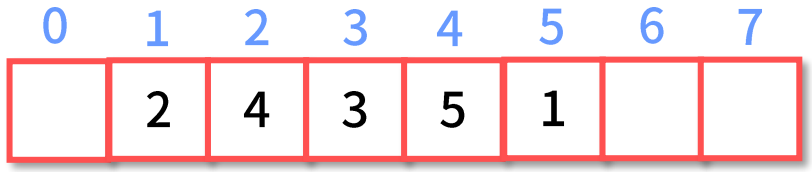
Pop

```
def heapify_down(self, index: int) -> None:
  while 2 * index <= self.n:
    child_a_index = index * 2
    child_b_index = index * 2 + 1
    if child_b_index > self.n:
      exchange_to = child_a_index
    elif self.data[child_b_index] < self.data[child_a_index]:
      exchange_to = child_b_index
    else:
      exchange_to = child_a_index
    if self.data[index] <= self.data[exchange_to]:
      break
    self.data.exchange(index, exchange_to)
    index = exchange_to
  return
```

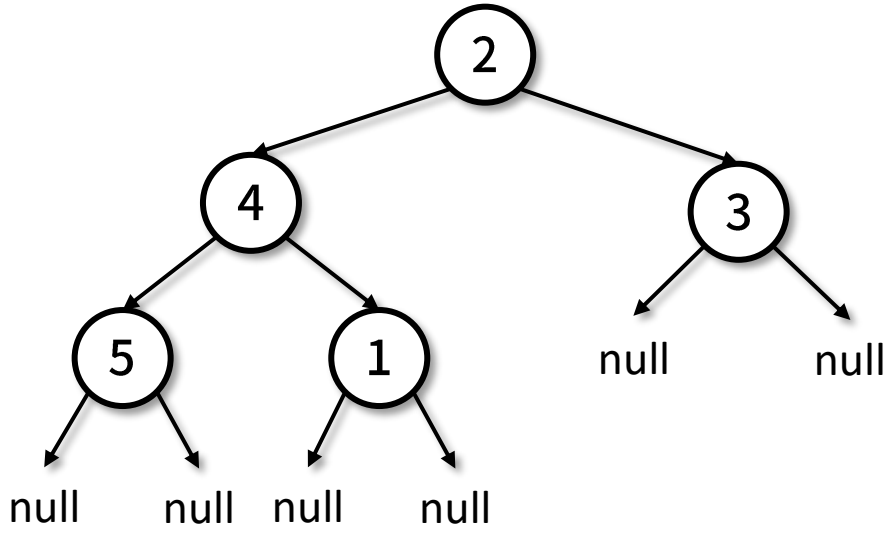
can we change the second if statement to the opposite?

Visualization

n = 4 result = 1



Tree Visualization



Minimum Heap

Pop (11)

Pop

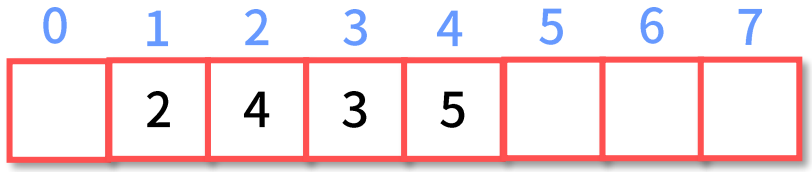
```
def pop(self) -> ComparableT:
  result = self.data[1]
  self.data.exchange(1, self.n)
  self.n -= 1
  self.heapify_down(1)
  self.data[self.n + 1] = None
  if self.n > 0 and self.n <= (len(self.data) - 1) / 4:
    self.data.resize(int(0.5 * len(self.data)))
  return result
```

Command

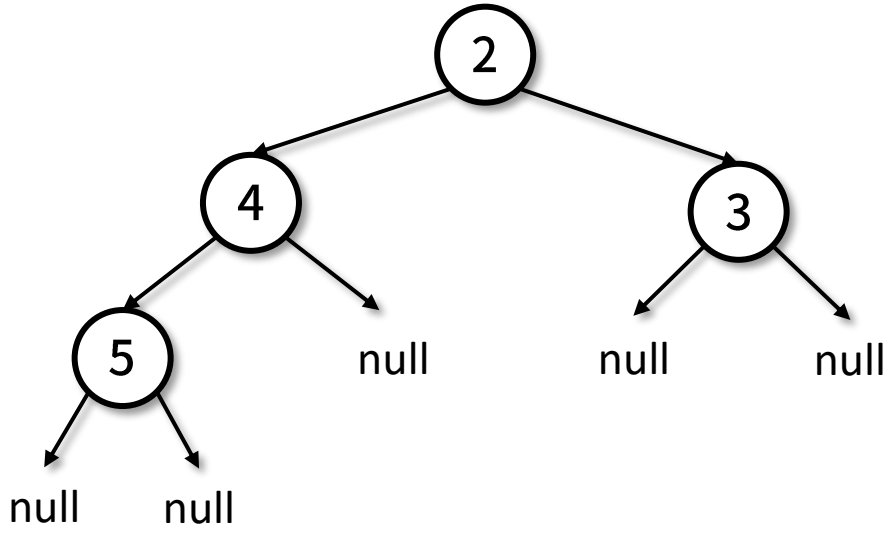
```
heap.pop()
```

Visualization

n = 4 result = 1



Tree Visualization



Heap

Summary

Worst Time Complexity

	Heap	Sorted Array
Insert	$O(\log N)^*$	$O(N)$
Pop	$O(\log N)^*$	$O(1)^*$

*amortized time complexity

Minimum Heap

Practices

- Maximum Product of Two Elements in an Array (Leetcode Problem 1464)
- Kth Largest Element in an Array (Leetcode Problem 215)

Binary Search Tree

Binary Search

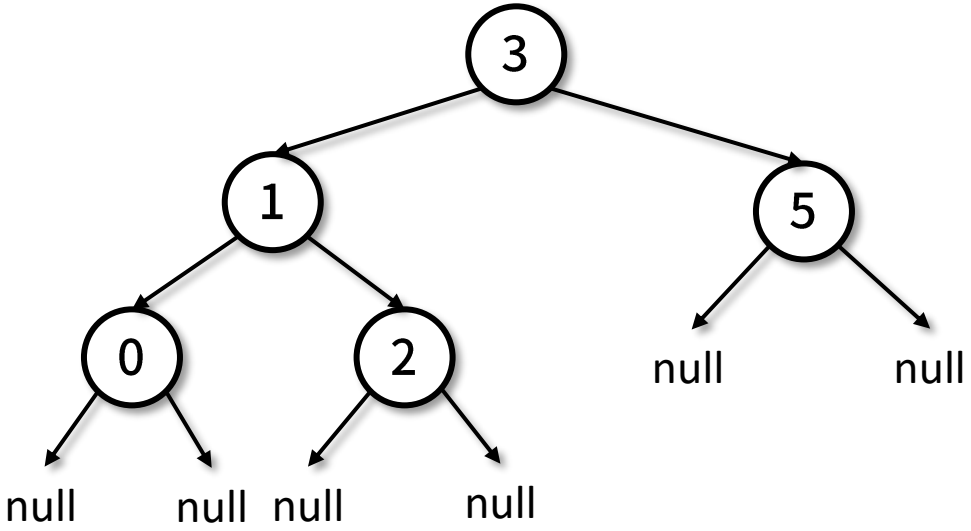
Function

```
def binary_search(sorted_array, item, start: int, end: int):
  if end < start:
    return -1
  middle = (start + end) // 2
  if sorted_array[middle] == item:
    return middle
  elif sorted_array[middle] > item:
    return binary_search(sorted_array, item, start, middle-1)
  elif sorted_array[middle] < item:
    return binary_search(sorted_array, item, middle+1, end)
```

- Given a sorted array, we could search an item in logarithm time complexity.
- However, insert new element into a sorted array takes linear time.
- Binary search tree is a data structure that uses the idea of binary search to ensure logarithm time complexity for *search* and *insertion*.
- It also support arbitrary comparable key.

Binary Search Tree (1)

- The key of the parent node cannot be larger than any of the key node in the right subtree.
- The key of the parent node cannot be smaller than any of the key node in the left subtree.
- Need additional operation for *insert* and *delete* to maintain the structure to ensure the first rule is invariant.



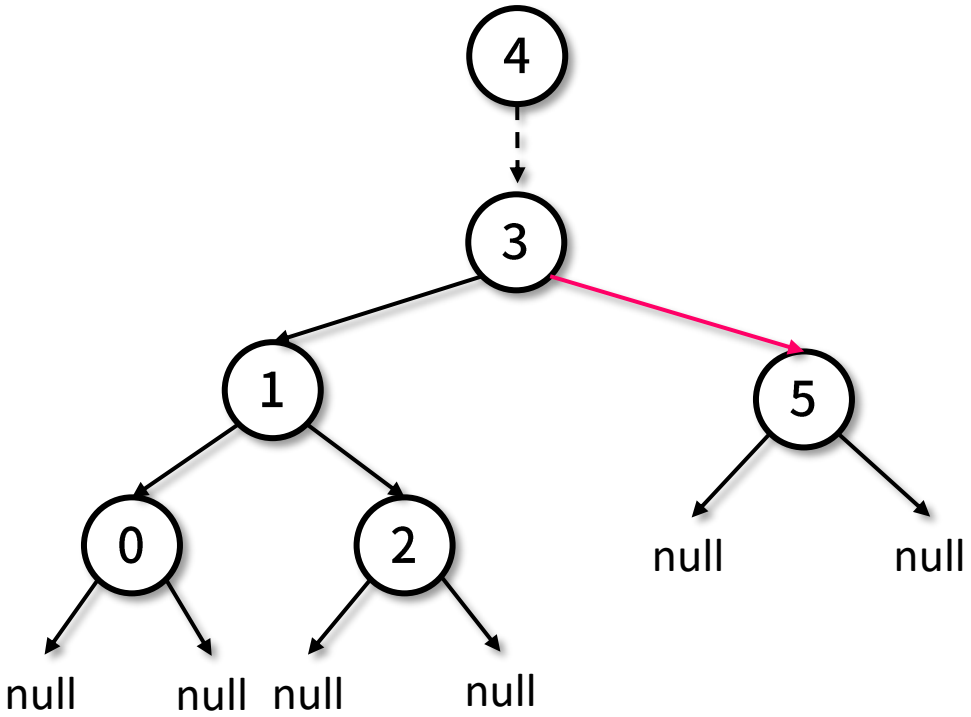
Binary Search Tree

Insert (1)

1. From the root, if the inserted node with key (e.g. 4) is larger than the root, check the right subtree.

2. If the inserted node with key is smaller than the root, check the left subtree.

3. Iterate until the same key is found, or reach the.



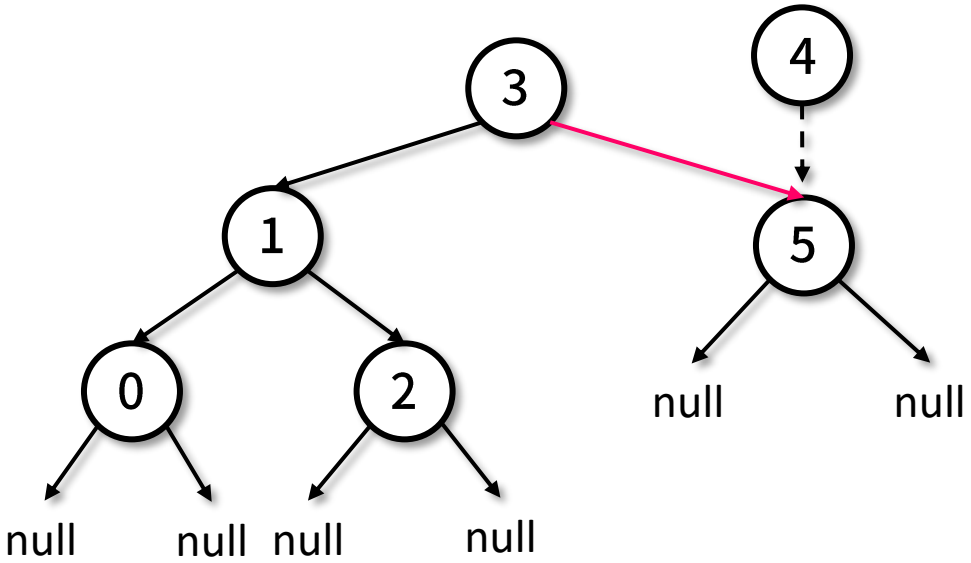
Binary Search Tree

Insert (2)

1. From the root, if the inserted node with key (e.g. 4) is larger than the root, check the right subtree.

2. If the inserted node with key is smaller than the root, check the left subtree.

3. Iterate until the same key is found, or reach the.



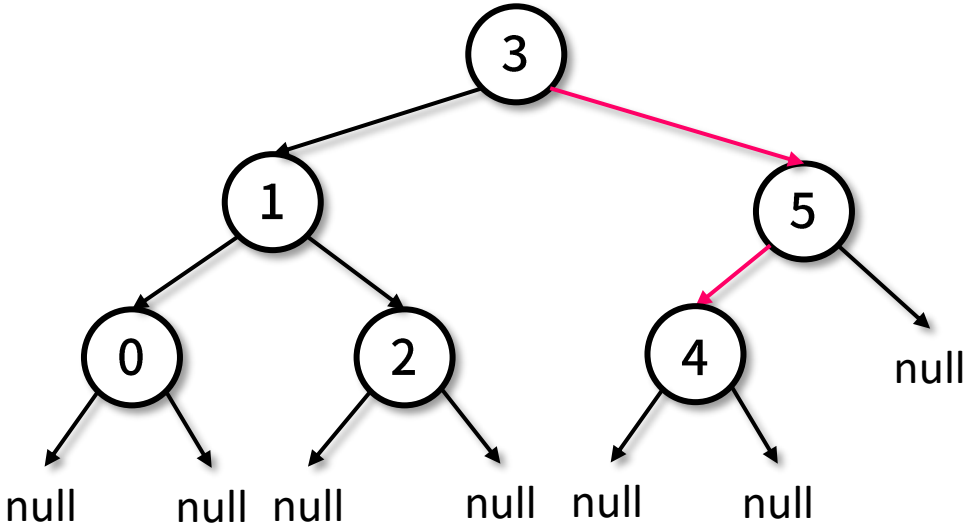
Binary Search Tree

Insert (3)

1. From the root, if the inserted node with key (e.g. 4) is larger than the root, check the right subtree.

2. If the inserted node with key is smaller than the root, check the left subtree.

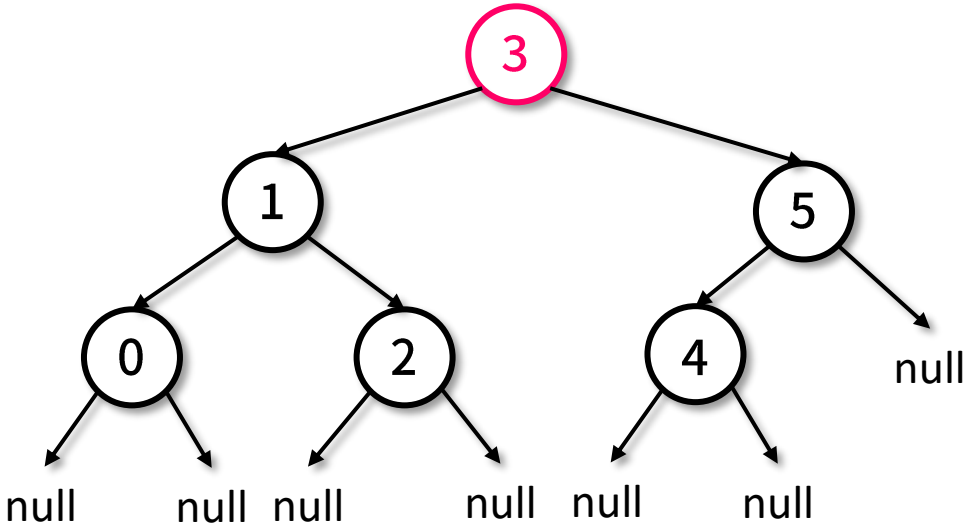
3. Iterate until the same key is found, or reach the.



Binary Search Tree

Deletion (1)

1. Use the same approach to reach we want to delete (e.g. node with key 3)

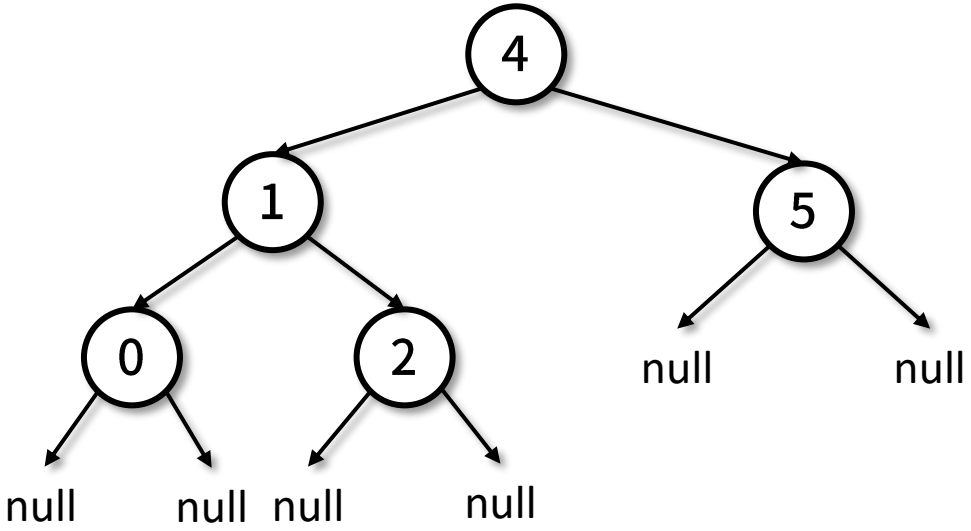


Binary Search Tree

Deletion (2)

1. Use the same approach to reach we want to delete (e.g. node with key 3)

2. Replace the removed node with the smallest node in the right subtree.



Binary Search Tree (2)

Initialization

```
class BinarySearchTree(MutableMapping):
  def __init__(self):
    self.n: int = 0
    self.root: BinarySearchTreeNode = None

  def __setitem__(self, key: ComparableT, value: Any) -> None:
      insert or update item
  def __getitem__(self, key: ComparableT) -> Any: ···
      get access to the item with given key
  def __delitem__(self, key: ComparableT) -> None: ···
      delete an item with given key
  def insert( ···
      helper function for insertion
  def delete( ···
      helper function for deletion
  def min_of_subtree( ···
      return the smallest node in the subtree
  def __iter__(self) -> BinarySearchTreeIterator: ···

  def __len__(self) -> int: ···
```

```
class BinarySearchTreeNode:
  def __init__(self, key: ComparableT, value: Any):
    self.key = key
    self.value = value
    self.left: BinarySearchTreeNode = None
    self.right: BinarySearchTreeNode = None
```

Command

```
bst = BinarySearchTree()
```

Visualization

n = 0

Binary Search Tree

Set Item (1)

Set Item

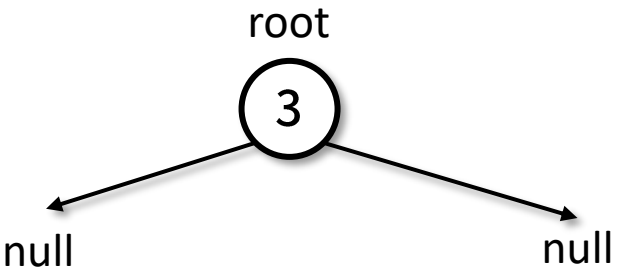
```
def __setitem__(self, key: ComparableT, value: Any) -> None:
  if self.n == 0:
    self.root = BinarySearchTreeNode(key, value)
    self.n += 1
    return
  else:
    self.root = self.insert(self.root, key, value)
    return
def insert(
    self,
    node: BinarySearchTreeNode,
    key: ComparableT,
    value: Any) -> BinarySearchTreeNode:
  if node is None:
    self.n += 1
    return BinarySearchTreeNode(key, value)

  if key < node.key:
    node.left = self.insert(node.left, key, value)
  elif key > node.key:
    node.right = self.insert(node.right, key, value)
  else:
    node.value = value
  return node
```

Command

```
bst[3] = '3'
```

Visualization

n = 1



Binary Search Tree

Set Item (2)

Set Item

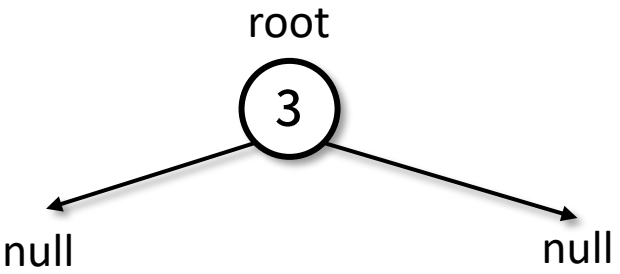
```
def __setitem__(self, key: ComparableT, value: Any) -> None:
  if self.n == 0:
    self.root = BinarySearchTreeNode(key, value)
    self.n += 1
    return
  else:
    self.root = self.insert(self.root, key, value)
    return
def insert(
    self,
    node: BinarySearchTreeNode,
    key: ComparableT,
    value: Any) -> BinarySearchTreeNode:
  if node is None:
    self.n += 1
    return BinarySearchTreeNode(key, value)

  if key < node.key:
    node.left = self.insert(node.left, key, value)
  elif key > node.key:
    node.right = self.insert(node.right, key, value)
  else:
    node.value = value
  return node
```

Command

```
bst[1] = '1'
```

Visualization

n = 1



Binary Search Tree

Set Item (3)

Set Item

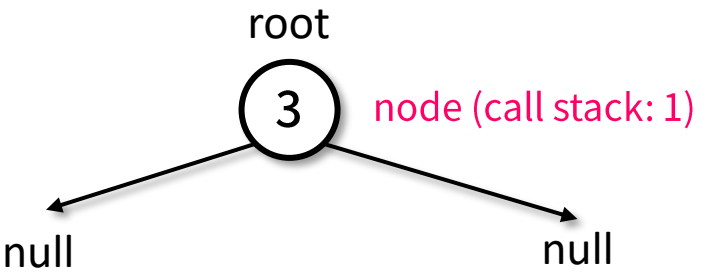
```
def __setitem__(self, key: ComparableT, value: Any) -> None:
  if self.n == 0:
    self.root = BinarySearchTreeNode(key, value)
    self.n += 1
    return
  else:
    self.root = self.insert(self.root, key, value)
    return
def insert(
    self,
    node: BinarySearchTreeNode,
    key: ComparableT,
    value: Any) -> BinarySearchTreeNode:
  if node is None:
    self.n += 1
    return BinarySearchTreeNode(key, value)

  if key < node.key:
    node.left = self.insert(node.left, key, value)
  elif key > node.key:
    node.right = self.insert(node.right, key, value)
  else:
    node.value = value
  return node
```

Command

```
bst[1] = '1'
```

Visualization

n = 1



Binary Search Tree

Set Item (4)

Set Item

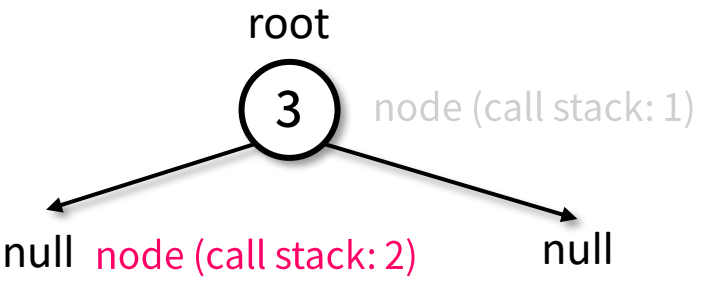
```
def __setitem__(self, key: ComparableT, value: Any) -> None:
  if self.n == 0:
    self.root = BinarySearchTreeNode(key, value)
    self.n += 1
    return
  else:
    self.root = self.insert(self.root, key, value)
    return
def insert(
    self,
    node: BinarySearchTreeNode,
    key: ComparableT,
    value: Any) -> BinarySearchTreeNode:
  if node is None:
    self.n += 1
    return BinarySearchTreeNode(key, value)

  if key < node.key:
    node.left = self.insert(node.left, key, value)
  elif key > node.key:
    node.right = self.insert(node.right, key, value)
  else:
    node.value = value
  return node
```

Command

```
bst[1] = '1'
```

Visualization

n = 2



Binary Search Tree

Set Item (5)

Set Item

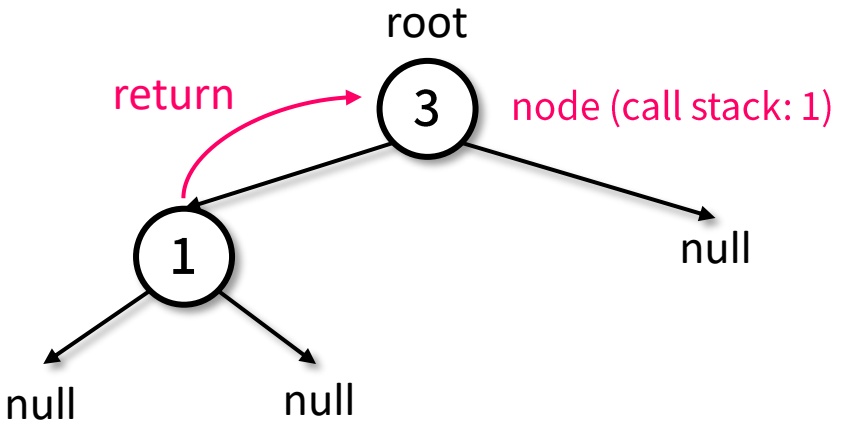
```
def __setitem__(self, key: ComparableT, value: Any) -> None:
  if self.n == 0:
    self.root = BinarySearchTreeNode(key, value)
    self.n += 1
    return
  else:
    self.root = self.insert(self.root, key, value)
    return
def insert(
    self,
    node: BinarySearchTreeNode,
    key: ComparableT,
    value: Any) -> BinarySearchTreeNode:
  if node is None:
    self.n += 1
    return BinarySearchTreeNode(key, value)

  if key < node.key:
    node.left = self.insert(node.left, key, value)
  elif key > node.key:
    node.right = self.insert(node.right, key, value)
  else:
    node.value = value
  return node
```

Command

```
bst[1] = '1'
```

Visualization

n = 2



Binary Search Tree

Set Item (6)

Set Item

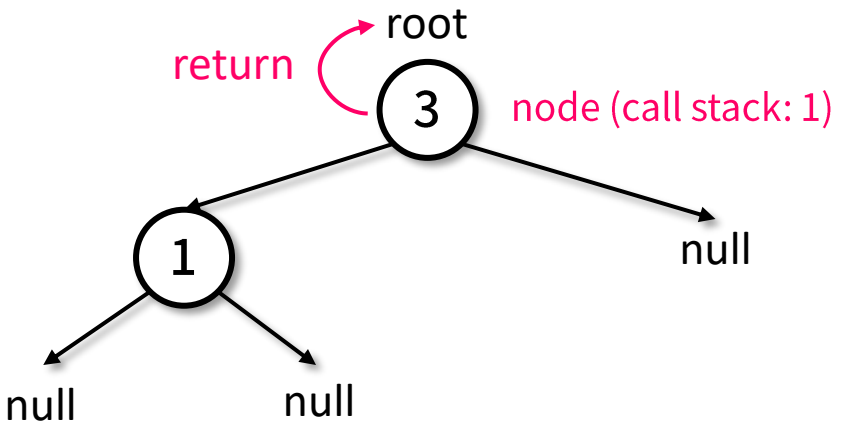
```
def __setitem__(self, key: ComparableT, value: Any) -> None:
  if self.n == 0:
    self.root = BinarySearchTreeNode(key, value)
    self.n += 1
    return
  else:
    self.root = self.insert(self.root, key, value)
    return
def insert(
    self,
    node: BinarySearchTreeNode,
    key: ComparableT,
    value: Any) -> BinarySearchTreeNode:
  if node is None:
    self.n += 1
    return BinarySearchTreeNode(key, value)

  if key < node.key:
    node.left = self.insert(node.left, key, value)
  elif key > node.key:
    node.right = self.insert(node.right, key, value)
  else:
    node.value = value
  return node
```

Command

```
bst[1] = '1'
```

Visualization

n = 2



Binary Search Tree

Set Item (7)

Set Item

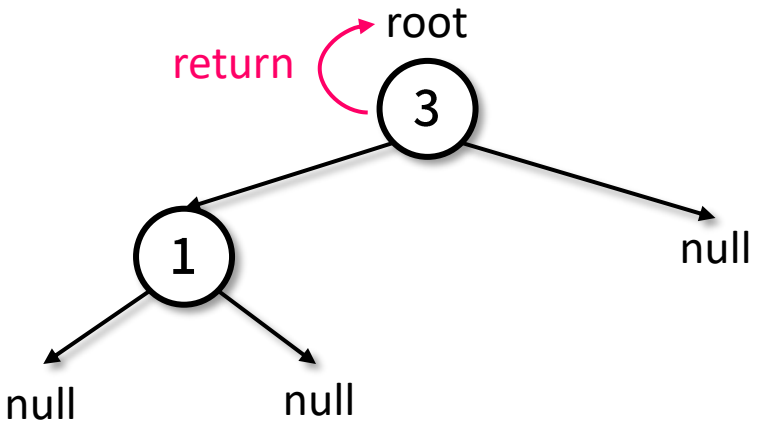
```
def __setitem__(self, key: ComparableT, value: Any) -> None:
  if self.n == 0:
    self.root = BinarySearchTreeNode(key, value)
    self.n += 1
    return
  else:
    self.root = self.insert(self.root, key, value)
    return
def insert(
    self,
    node: BinarySearchTreeNode,
    key: ComparableT,
    value: Any) -> BinarySearchTreeNode:
  if node is None:
    self.n += 1
    return BinarySearchTreeNode(key, value)

  if key < node.key:
    node.left = self.insert(node.left, key, value)
  elif key > node.key:
    node.right = self.insert(node.right, key, value)
  else:
    node.value = value
  return node
```

Command

```
bst[1] = '1'
```

Visualization

n = 2



Binary Search Tree

Set Item (8)

Set Item

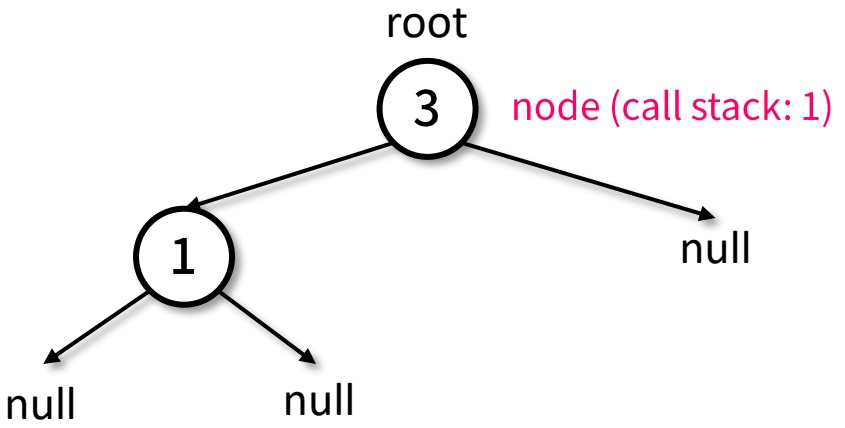
```
def __setitem__(self, key: ComparableT, value: Any) -> None:
  if self.n == 0:
    self.root = BinarySearchTreeNode(key, value)
    self.n += 1
    return
  else:
    self.root = self.insert(self.root, key, value)
    return
def insert(
    self,
    node: BinarySearchTreeNode,
    key: ComparableT,
    value: Any) -> BinarySearchTreeNode:
  if node is None:
    self.n += 1
    return BinarySearchTreeNode(key, value)

  if key < node.key:
    node.left = self.insert(node.left, key, value)
  elif key > node.key:
    node.right = self.insert(node.right, key, value)
  else:
    node.value = value
  return node
```

Command

```
bst[2] = '2'
```

Visualization

n = 2



Binary Search Tree

Set Item (9)

Set Item

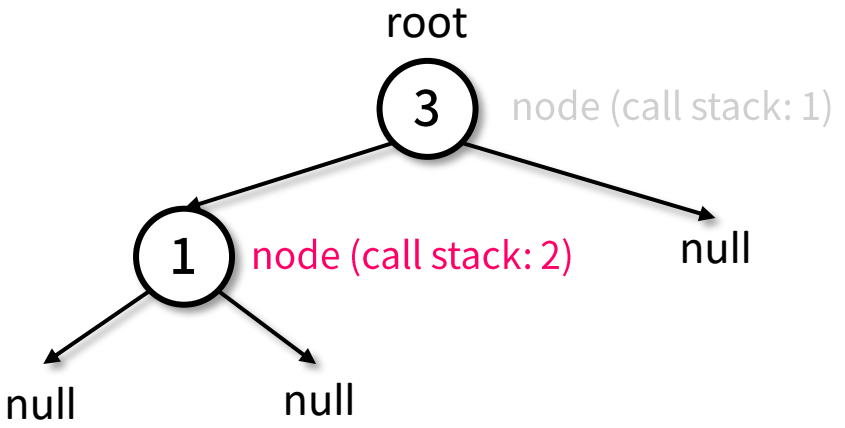
```
def __setitem__(self, key: ComparableT, value: Any) -> None:
  if self.n == 0:
    self.root = BinarySearchTreeNode(key, value)
    self.n += 1
    return
  else:
    self.root = self.insert(self.root, key, value)
    return
def insert(
    self,
    node: BinarySearchTreeNode,
    key: ComparableT,
    value: Any) -> BinarySearchTreeNode:
  if node is None:
    self.n += 1
    return BinarySearchTreeNode(key, value)

  if key < node.key:
    node.left = self.insert(node.left, key, value)
  elif key > node.key:
    node.right = self.insert(node.right, key, value)
  else:
    node.value = value
  return node
```

Command

```
bst[2] = '2'
```

Visualization

n = 2



Binary Search Tree

Set Item (10)

Set Item

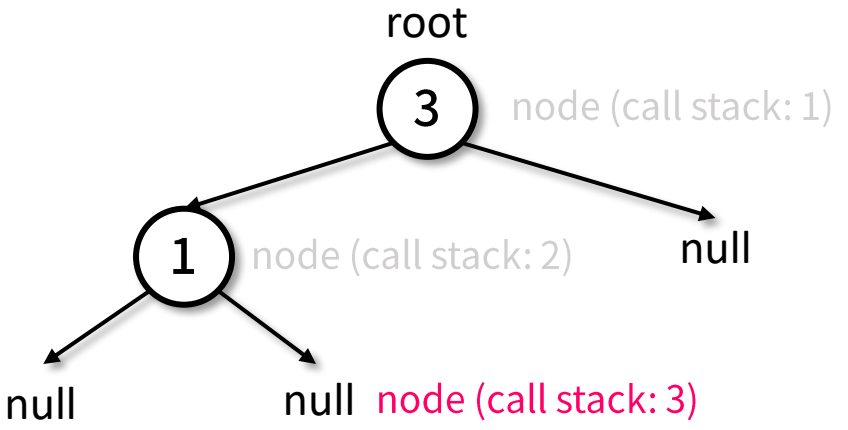
```
def __setitem__(self, key: ComparableT, value: Any) -> None:
  if self.n == 0:
    self.root = BinarySearchTreeNode(key, value)
    self.n += 1
    return
  else:
    self.root = self.insert(self.root, key, value)
    return
def insert(
    self,
    node: BinarySearchTreeNode,
    key: ComparableT,
    value: Any) -> BinarySearchTreeNode:
  if node is None:
    self.n += 1
    return BinarySearchTreeNode(key, value)

  if key < node.key:
    node.left = self.insert(node.left, key, value)
  elif key > node.key:
    node.right = self.insert(node.right, key, value)
  else:
    node.value = value
  return node
```

Command

```
bst[2] = '2'
```

Visualization

n = 3



Binary Search Tree

Set Item (11)

Set Item

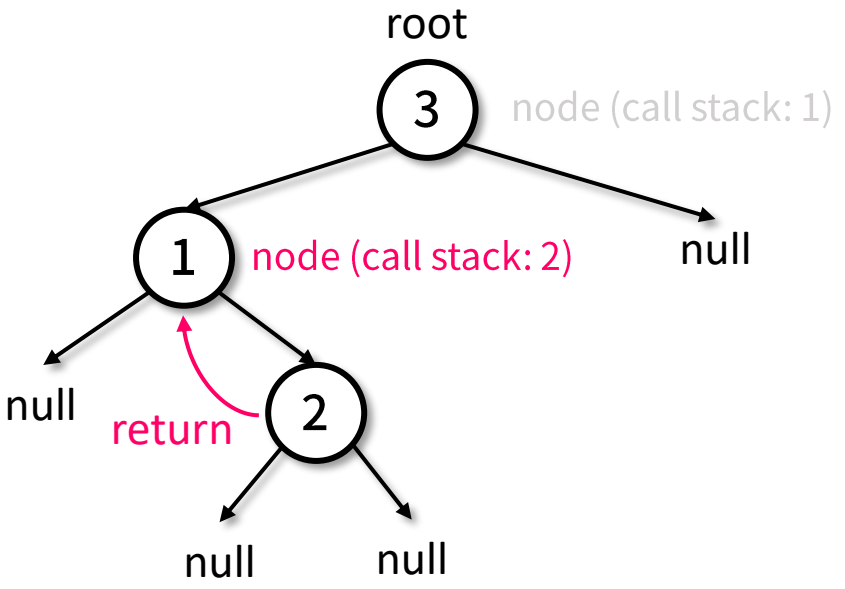
```python
def __setitem__(self, key: ComparableT, value: Any) -> None:
    if self.n == 0:
        self.root = BinarySearchTreeNode(key, value)
        self.n += 1
        return
    else:
        self.root = self.insert(self.root, key, value)
        return
def insert(
        self,
        node: BinarySearchTreeNode,
        key: ComparableT,
        value: Any) -> BinarySearchTreeNode:
    if node is None:
        self.n += 1
        return BinarySearchTreeNode(key, value)

    if key < node.key:
        node.left = self.insert(node.left, key, value)
    elif key > node.key:
        node.right = self.insert(node.right, key, value)
    else:
        node.value = value
    return node
```

Command

```
bst[2] = '2'
```

Visualization

n = 3



Binary Search Tree

Set Item (12)

Set Item

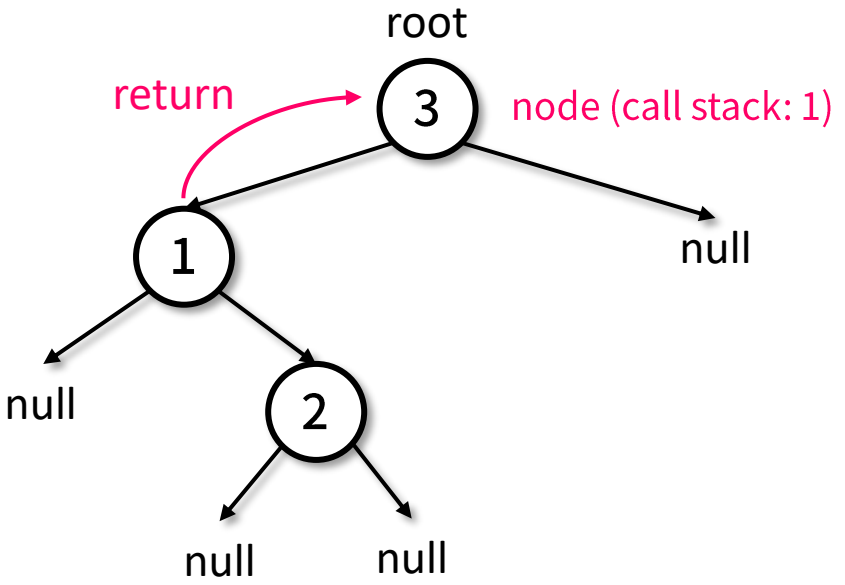
```
def __setitem__(self, key: ComparableT, value: Any) -> None:
  if self.n == 0:
    self.root = BinarySearchTreeNode(key, value)
    self.n += 1
    return
  else:
    self.root = self.insert(self.root, key, value)
    return
def insert(
    self,
    node: BinarySearchTreeNode,
    key: ComparableT,
    value: Any) -> BinarySearchTreeNode:
  if node is None:
    self.n += 1
    return BinarySearchTreeNode(key, value)

  if key < node.key:
    node.left = self.insert(node.left, key, value)
  elif key > node.key:
    node.right = self.insert(node.right, key, value)
  else:
    node.value = value
  return node
```

Command

```
bst[2] = '2'
```

Visualization

n = 3



Binary Search Tree

Set Item (13)

Set Item

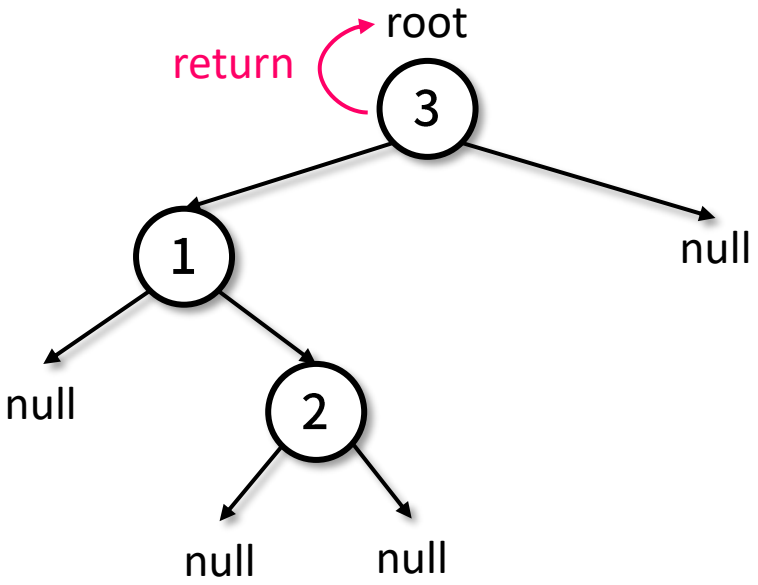
```
def __setitem__(self, key: ComparableT, value: Any) -> None:
  if self.n == 0:
    self.root = BinarySearchTreeNode(key, value)
    self.n += 1
    return
  else:
    self.root = self.insert(self.root, key, value)
    return
def insert(
    self,
    node: BinarySearchTreeNode,
    key: ComparableT,
    value: Any) -> BinarySearchTreeNode:
  if node is None:
    self.n += 1
    return BinarySearchTreeNode(key, value)

  if key < node.key:
    node.left = self.insert(node.left, key, value)
  elif key > node.key:
    node.right = self.insert(node.right, key, value)
  else:
    node.value = value
  return node
```

Command

```
bst[2] = '2'
```

Visualization

n = 3



Binary Search Tree

Set Item (14)

Set Item

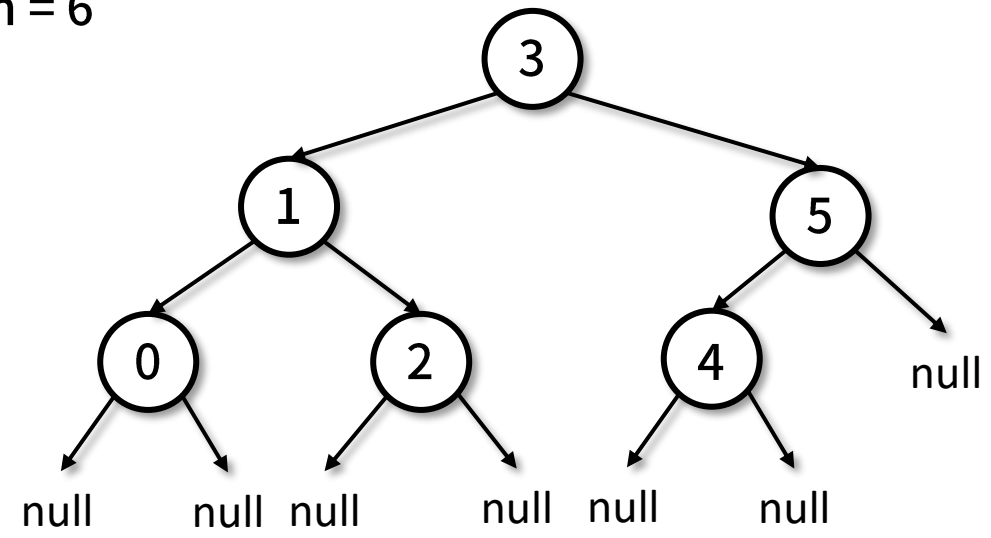
```
def __setitem__(self, key: ComparableT, value: Any) -> None:
  if self.n == 0:
    self.root = BinarySearchTreeNode(key, value)
    self.n += 1
    return
  else:
    self.root = self.insert(self.root, key, value)
    return
def insert(
    self,
    node: BinarySearchTreeNode,
    key: ComparableT,
    value: Any) -> BinarySearchTreeNode:
  if node is None:
    self.n += 1
    return BinarySearchTreeNode(key, value)

  if key < node.key:
    node.left = self.insert(node.left, key, value)
  elif key > node.key:
    node.right = self.insert(node.right, key, value)
  else:
    node.value = value
  return node
```

Command

```
bst[0] = '0'
bst[5] = '5'
bst[4] = '4'
```

Visualization

n = 6



Binary Search Tree

Delete Item (1)

Delete Item

```
def __delitem__(self, key: ComparableT) -> None:
    self.delete(self.root, key)
    self.n -= 1
    return
```

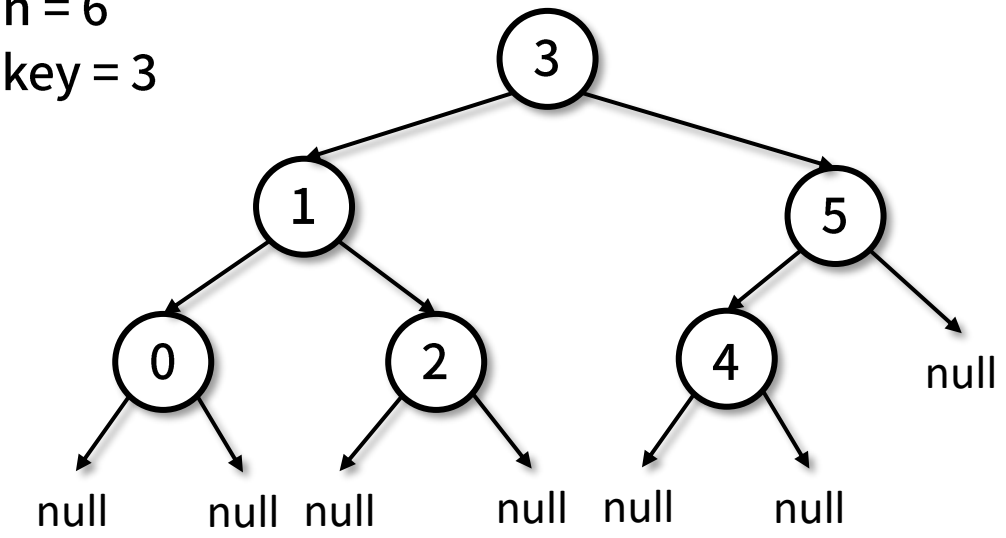
Command

```
del bst[3]
```

Visualization

n = 6

key = 3



Binary Search Tree

Delete Item (2)

Delete Item

```
def delete(
    self,
    node: BinarySearchTreeNode,
    key: ComparableT) -> BinarySearchTreeNode:
  if node is None:
    raise KeyError(f'{key}')

  if key < node.key:
    node.left = self.delete(node.left, key)
  elif key > node.key:
    node.right = self.delete(node.right, key)
  else:
    if node.left is None:
      temp = node.right
      node = None
      return temp
    elif node.right is None:
      node = node.left
      node = None
      return temp
    else:
      temp = self.min_of_subtree(node.right)
      node.key = temp.key
      node.value = temp.value
      node.right = self.delete(node.right, temp.key)

  return node
```

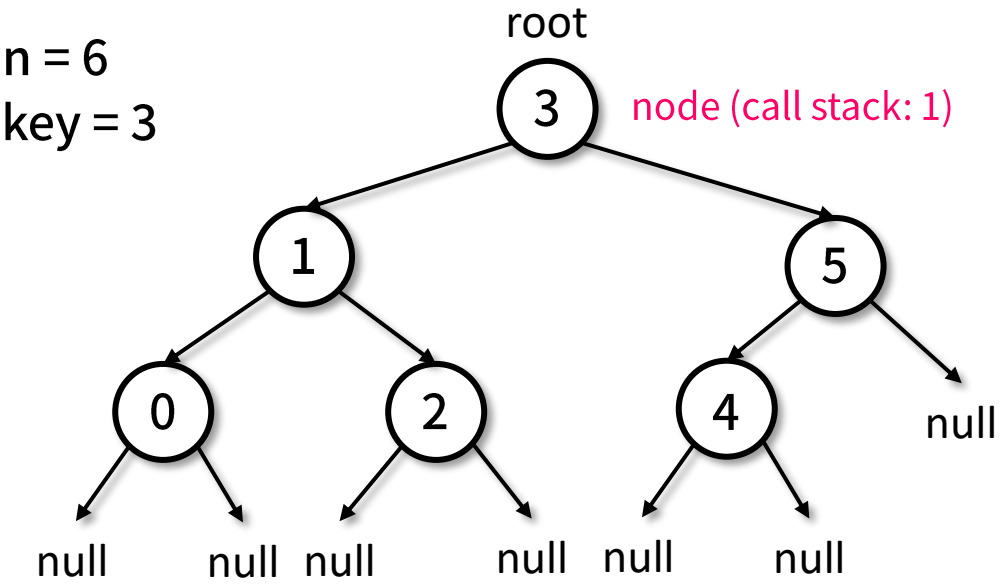
Command

```
del bst[3]
```

Visualization

n = 6

key = 3



Binary Search Tree

Delete Item (3)

Delete Item

```
def delete(
    self,
    node: BinarySearchTreeNode,
    key: ComparableT) -> BinarySearchTreeNode:
  if node is None:
    raise KeyError(f'{key}')

  if key < node.key:
    node.left = self.delete(node.left, key)
  elif key > node.key:
    node.right = self.delete(node.right, key)
  else:
    if node.left is None:
      temp = node.right
      node = None
      return temp
    elif node.right is None:
      node = node.left
      node = None
      return temp
    else:
      temp = self.min_of_subtree(node.right)
      node.key = temp.key
      node.value = temp.value
      node.right = self.delete(node.right, temp.key)

  return node
```

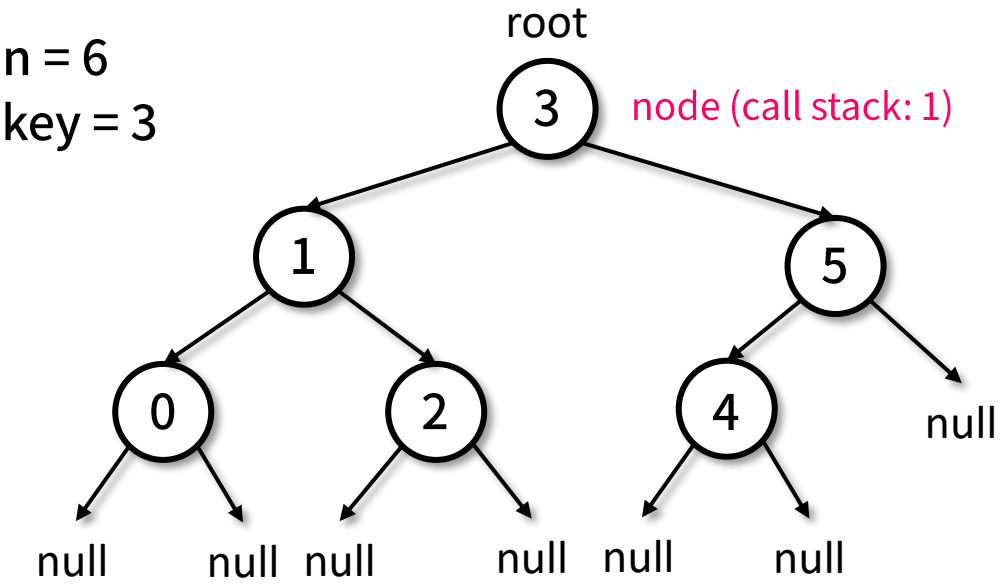
Command

```
del bst[3]
```

Visualization

n = 6

key = 3



Binary Search Tree

Delete Item (4)

Delete Item

```
def min_of_subtree(
    self,
    node: BinarySearchTreeNode) -> BinarySearchTreeNode:
  while node.left is not None:
    node = node.left
  return node
```

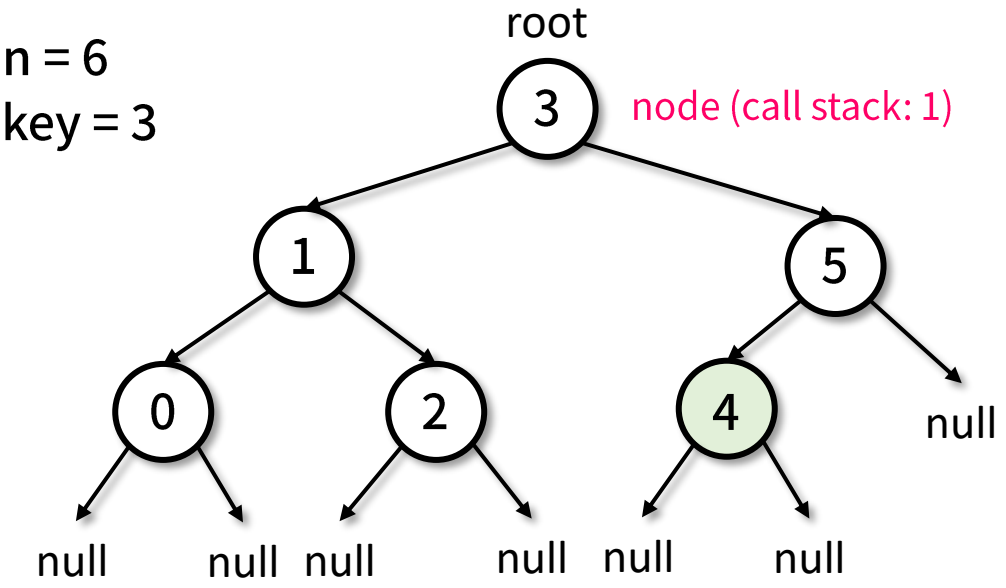
Command

```
del bst[3]
```

Visualization

n = 6

key = 3



Binary Search Tree

Delete Item (5)

Delete Item

```
def delete(
    self,
    node: BinarySearchTreeNode,
    key: ComparableT) -> BinarySearchTreeNode:
  if node is None:
    raise KeyError(f'{key}')

  if key < node.key:
    node.left = self.delete(node.left, key)
  elif key > node.key:
    node.right = self.delete(node.right, key)
  else:
    if node.left is None:
      temp = node.right
      node = None
      return temp
    elif node.right is None:
      node = node.left
      node = None
      return temp
    else:
      temp = self.min_of_subtree(node.right)
      node.key = temp.key
      node.value = temp.value
      node.right = self.delete(node.right, temp.key)

  return node
```

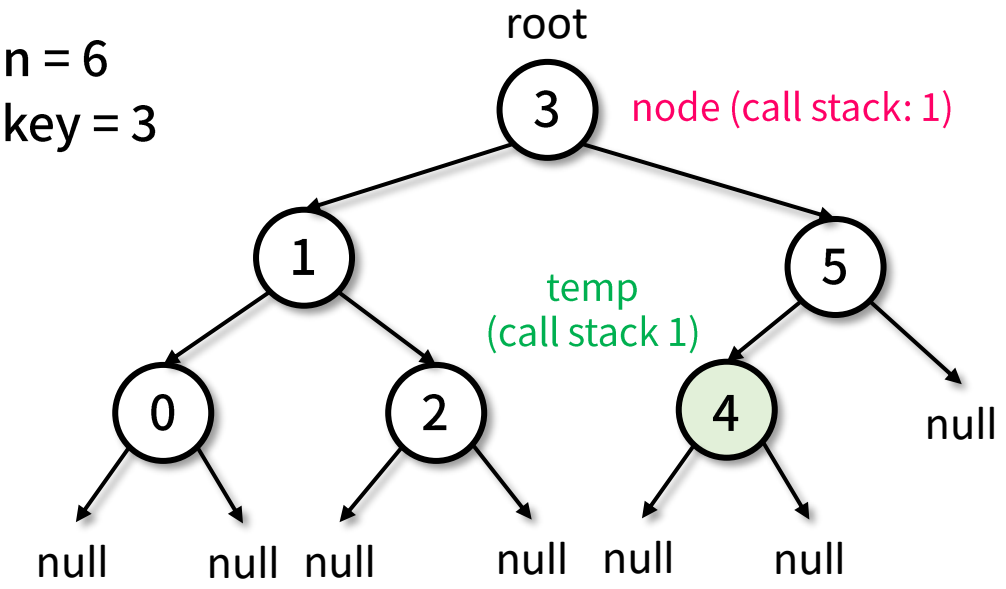
Command

```
del bst[3]
```

Visualization

n = 6

key = 3



Binary Search Tree

Delete Item (6)

Delete Item

```
def delete(
    self,
    node: BinarySearchTreeNode,
    key: ComparableT) -> BinarySearchTreeNode:
  if node is None:
    raise KeyError(f'{key}')

  if key < node.key:
    node.left = self.delete(node.left, key)
  elif key > node.key:
    node.right = self.delete(node.right, key)
  else:
    if node.left is None:
      temp = node.right
      node = None
      return temp
    elif node.right is None:
      node = node.left
      node = None
      return temp
    else:
      temp = self.min_of_subtree(node.right)
      node.key = temp.key
      node.value = temp.value
      node.right = self.delete(node.right, temp.key)

  return node
```

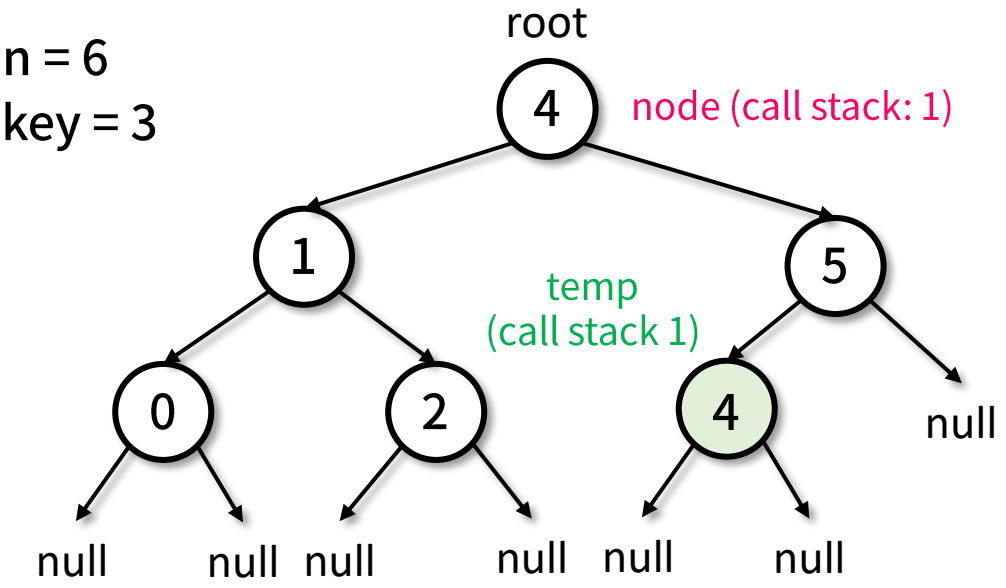
Command

```
del bst[3]
```

Visualization

n = 6

key = 3



Binary Search Tree

Delete Item (7)

Delete Item

```
def delete(
    self,
    node: BinarySearchTreeNode,
    key: ComparableT) -> BinarySearchTreeNode:
  if node is None:
    raise KeyError(f'{key}')

  if key < node.key:
    node.left = self.delete(node.left, key)
  elif key > node.key:
    node.right = self.delete(node.right, key)
  else:
    if node.left is None:
      temp = node.right
      node = None
      return temp
    elif node.right is None:
      node = node.left
      node = None
      return temp
    else:
      temp = self.min_of_subtree(node.right)
      node.key = temp.key
      node.value = temp.value
      node.right = self.delete(node.right, temp.key)

  return node
```

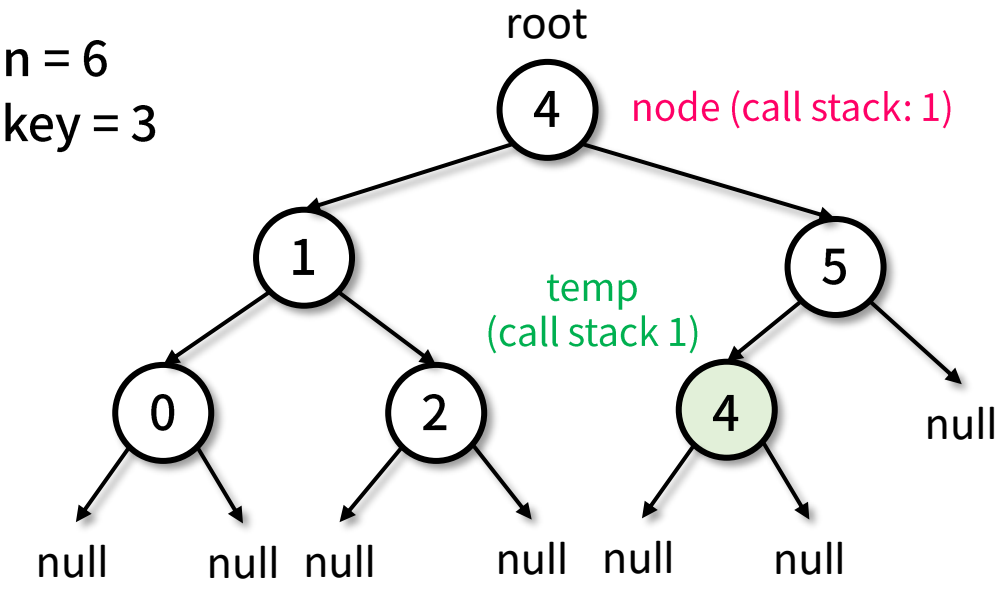
Command

```
del bst[3]
```

Visualization

n = 6

key = 3



Binary Search Tree

Delete Item (8)

Delete Item

```
def delete(
    self,
    node: BinarySearchTreeNode,
    key: ComparableT) -> BinarySearchTreeNode:
  if node is None:
    raise KeyError(f'{key}')

  if key < node.key:
    node.left = self.delete(node.left, key)
  elif key > node.key:
    node.right = self.delete(node.right, key)
  else:
    if node.left is None:
      temp = node.right
      node = None
      return temp
    elif node.right is None:
      node = node.left
      node = None
      return temp
    else:
      temp = self.min_of_subtree(node.right)
      node.key = temp.key
      node.value = temp.value
      node.right = self.delete(node.right, temp.key)

  return node
```

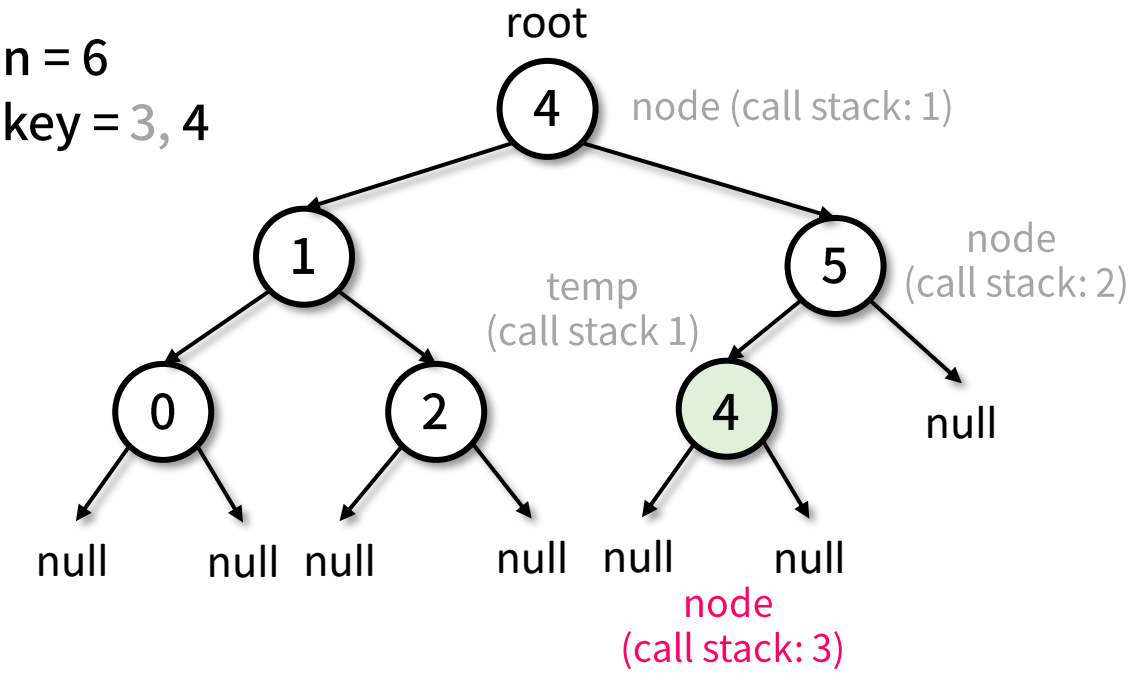
Command

```
del bst[3]
```

Visualization

n = 6

key = 3, 4



Binary Search Tree

Delete Item (9)

Delete Item

```
def delete(
    self,
    node: BinarySearchTreeNode,
    key: ComparableT) -> BinarySearchTreeNode:
  if node is None:
    raise KeyError(f'{key}')

  if key < node.key:
    node.left = self.delete(node.left, key)
  elif key > node.key:
    node.right = self.delete(node.right, key)
  else:
    if node.left is None:
      temp = node.right
      node = None
      return temp
    elif node.right is None:
      node = node.left
      node = None
      return temp
    else:
      temp = self.min_of_subtree(node.right)
      node.key = temp.key
      node.value = temp.value
      node.right = self.delete(node.right, temp.key)

  return node
```

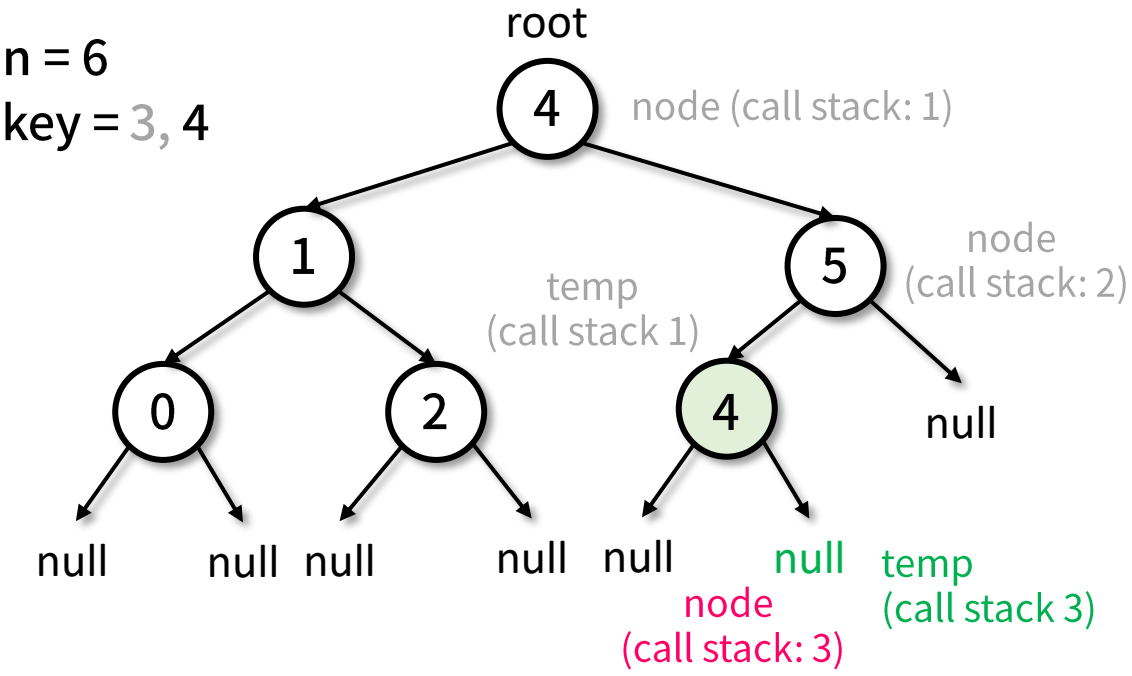
Command

```
del bst[3]
```

Visualization

n = 6

key = 3, 4



Binary Search Tree

Delete Item (10)

Delete Item

```
def delete(
    self,
    node: BinarySearchTreeNode,
    key: ComparableT) -> BinarySearchTreeNode:
  if node is None:
    raise KeyError(f'{key}')

  if key < node.key:
    node.left = self.delete(node.left, key)
  elif key > node.key:
    node.right = self.delete(node.right, key)
  else:
    if node.left is None:
      temp = node.right
      node = None
      return temp
    elif node.right is None:
      node = node.left
      node = None
      return temp
    else:
      temp = self.min_of_subtree(node.right)
      node.key = temp.key
      node.value = temp.value
      node.right = self.delete(node.right, temp.key)

  return node
```

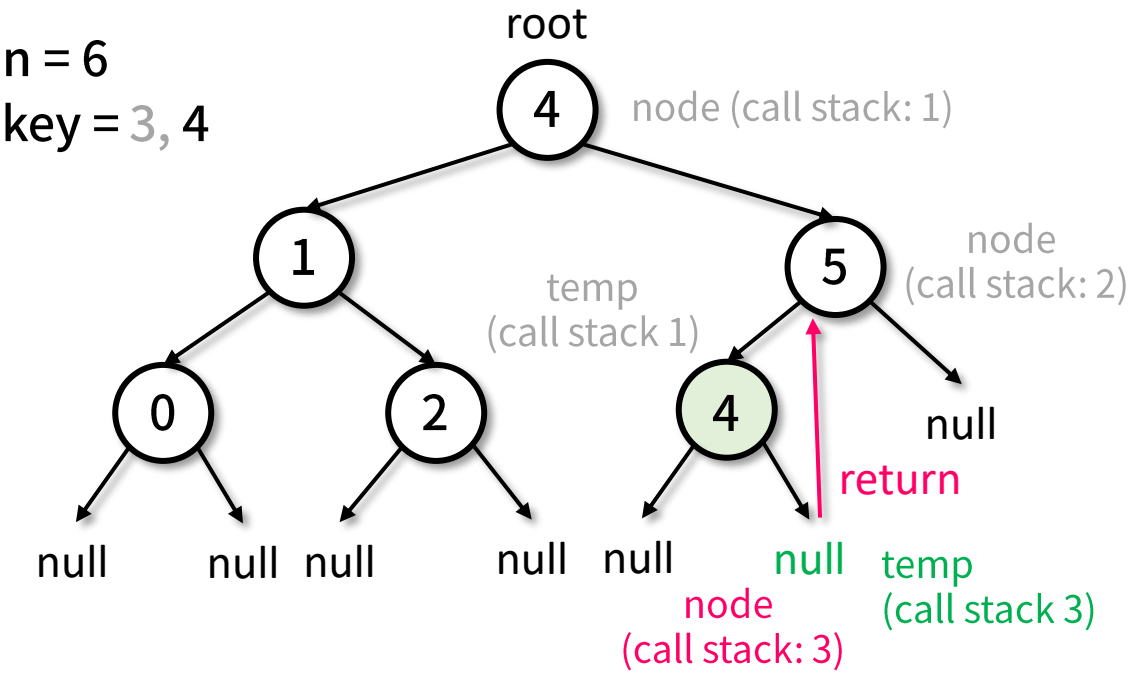
Command

```
del bst[3]
```

Visualization

n = 6

key = 3, 4



Binary Search Tree

Delete Item (11)

Delete Item

```python
def delete(
    self,
    node: BinarySearchTreeNode,
    key: ComparableT) -> BinarySearchTreeNode:
  if node is None:
    raise KeyError(f'{key}')

  if key < node.key:
    node.left = self.delete(node.left, key)
  elif key > node.key:
    node.right = self.delete(node.right, key)
  else:
    if node.left is None:
      temp = node.right
      node = None
      return temp
    elif node.right is None:
      node = node.left
      node = None
      return temp
    else:
      temp = self.min_of_subtree(node.right)
      node.key = temp.key
      node.value = temp.value
      node.right = self.delete(node.right, temp.key)

  return node
```

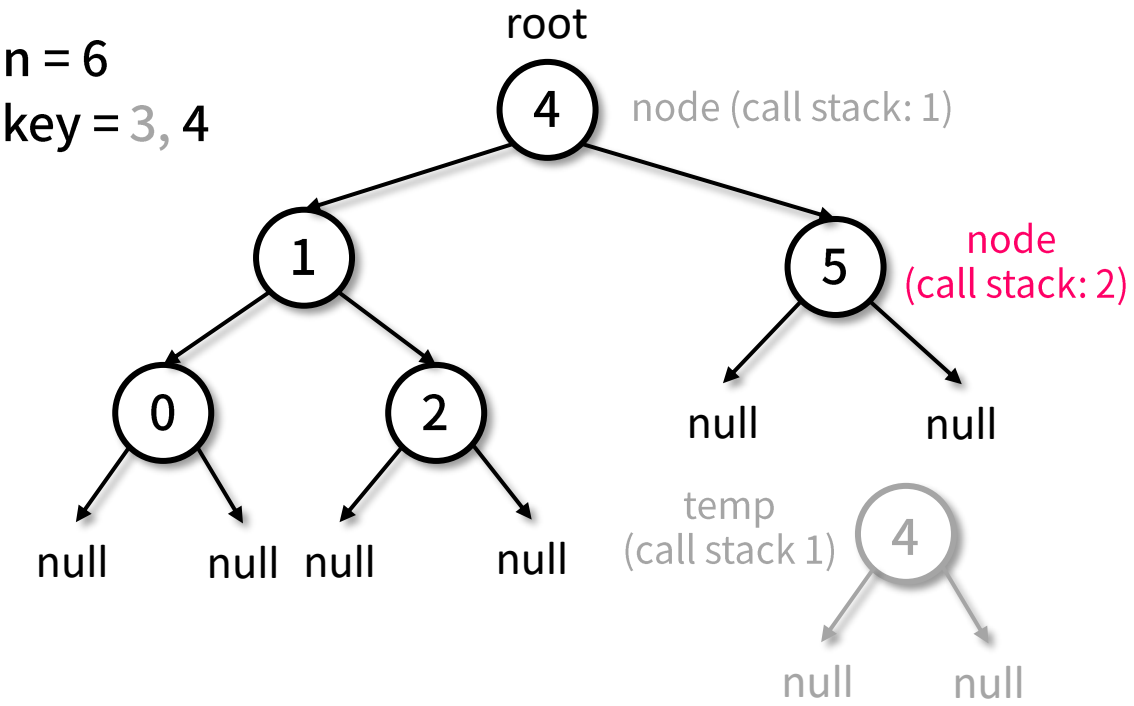
Command

```
del bst[3]
```

Visualization

n = 6

key = 3, 4



Binary Search Tree

Delete Item (12)

Delete Item

```
def delete(
    self,
    node: BinarySearchTreeNode,
    key: ComparableT) -> BinarySearchTreeNode:
  if node is None:
    raise KeyError(f'{key}')

  if key < node.key:
    node.left = self.delete(node.left, key)
  elif key > node.key:
    node.right = self.delete(node.right, key)
  else:
    if node.left is None:
      temp = node.right
      node = None
      return temp
    elif node.right is None:
      node = node.left
      node = None
      return temp
    else:
      temp = self.min_of_subtree(node.right)
      node.key = temp.key
      node.value = temp.value
      node.right = self.delete(node.right, temp.key)

  return node
```

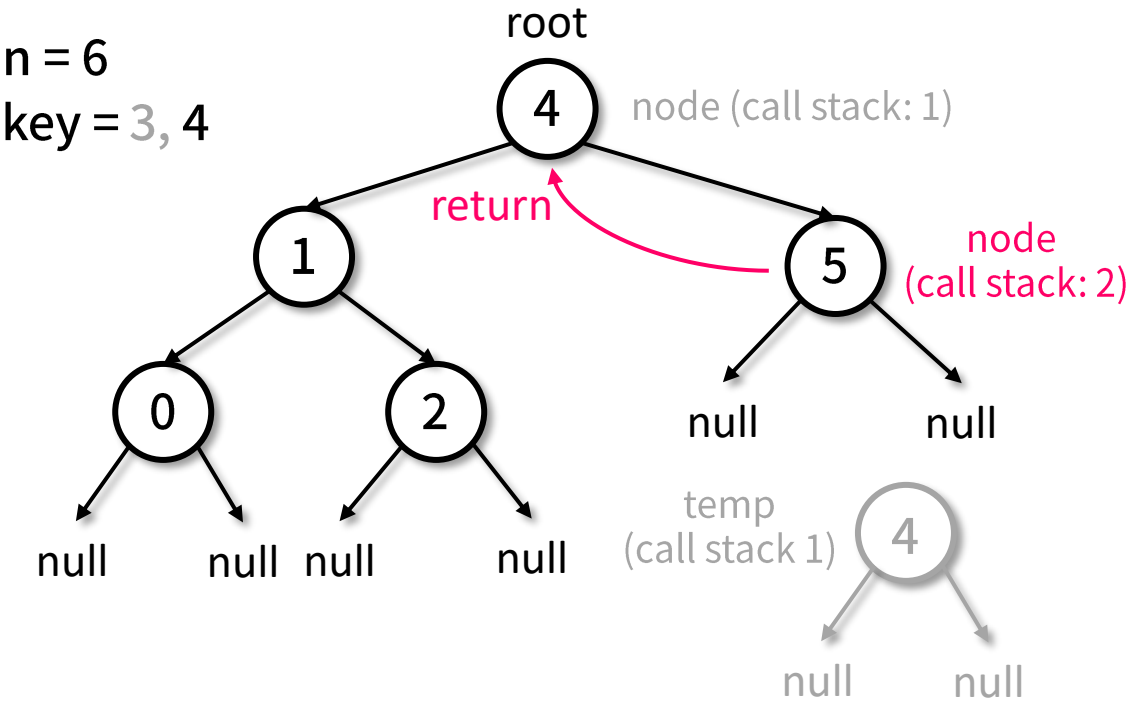
Command

```
del bst[3]
```

Visualization

n = 6

key = 3, 4



Binary Search Tree

Delete Item (13)

Delete Item

```
def delete(
    self,
    node: BinarySearchTreeNode,
    key: ComparableT) -> BinarySearchTreeNode:
  if node is None:
    raise KeyError(f'{key}')

  if key < node.key:
    node.left = self.delete(node.left, key)
  elif key > node.key:
    node.right = self.delete(node.right, key)
  else:
    if node.left is None:
      temp = node.right
      node = None
      return temp
    elif node.right is None:
      node = node.left
      node = None
      return temp
    else:
      temp = self.min_of_subtree(node.right)
      node.key = temp.key
      node.value = temp.value
      node.right = self.delete(node.right, temp.key)

  return node
```

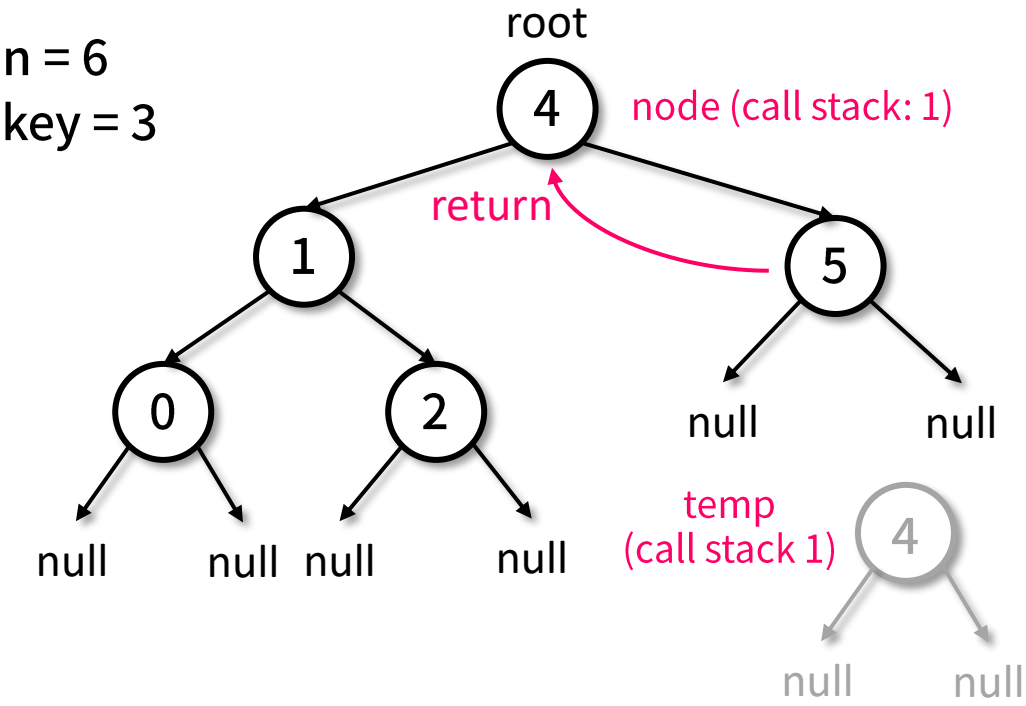
Command

```
del bst[3]
```

Visualization

n = 6

key = 3



Binary Search Tree

Delete Item (14)

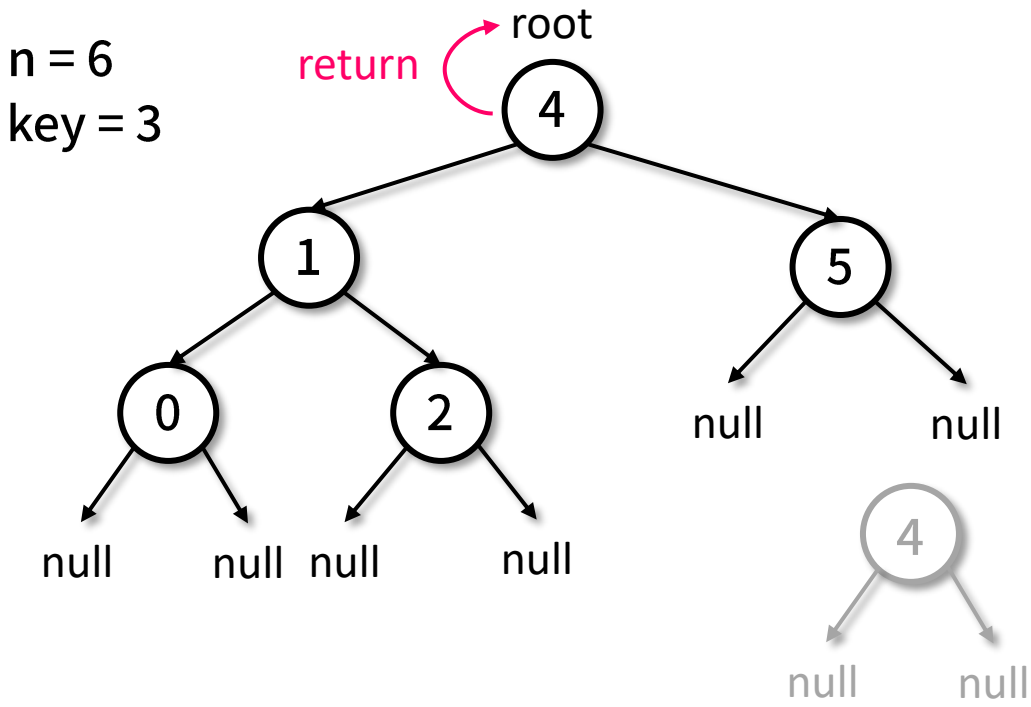
Delete Item

```
def delete(
    self,
    node: BinarySearchTreeNode,
    key: ComparableT) -> BinarySearchTreeNode:
  if node is None:
    raise KeyError(f'{key}')

  if key < node.key:
    node.left = self.delete(node.left, key)
  elif key > node.key:
    node.right = self.delete(node.right, key)
  else:
    if node.left is None:
      temp = node.right
      node = None
      return temp
    elif node.right is None:
      node = node.left
      node = None
      return temp
    else:
      temp = self.min_of_subtree(node.right)
      node.key = temp.key
      node.value = temp.value
      node.right = self.delete(node.right, temp.key)

  return node
```

Command

```
del bst[3]
```

Visualization



Binary Search Tree

Delete Item (15)

Delete Item

```
def __delitem__(self, key: ComparableT) -> None:
    self.delete(self.root, key)
    self.n -= 1
    return
```

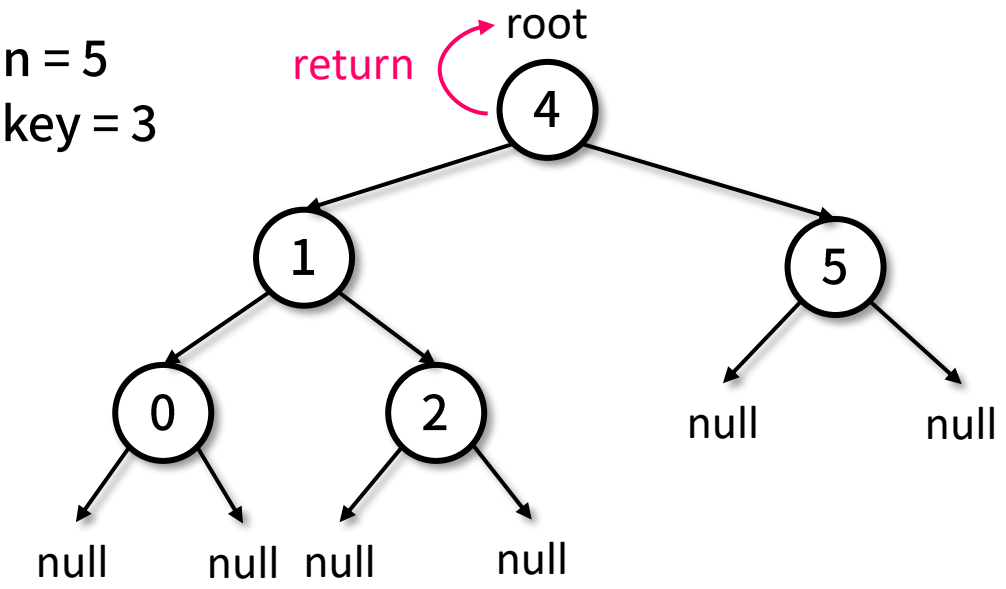
Command

```
del bst[3]
```

Visualization

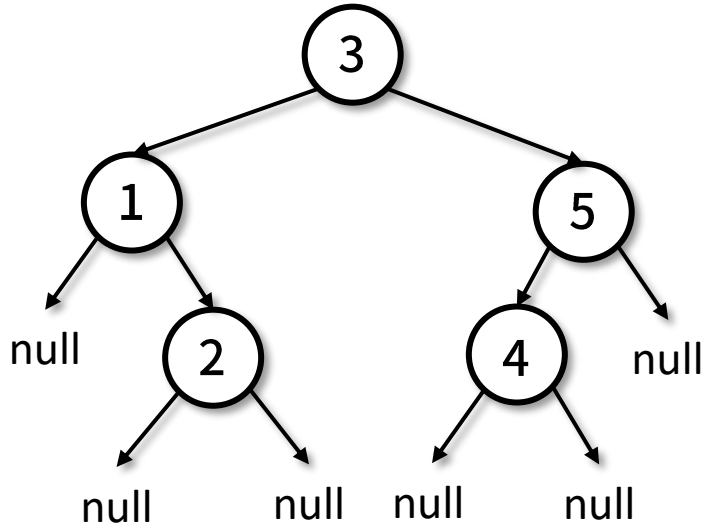
n = 5

key = 3



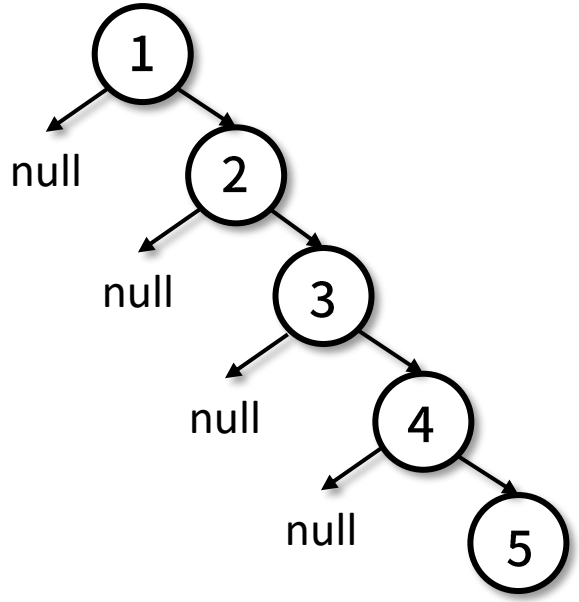
Problem of Binary Search Tree

- The tree structure depends on the order of insertion.



insertion: 3, 5, 4, 1, 2

average search time: 2.2T



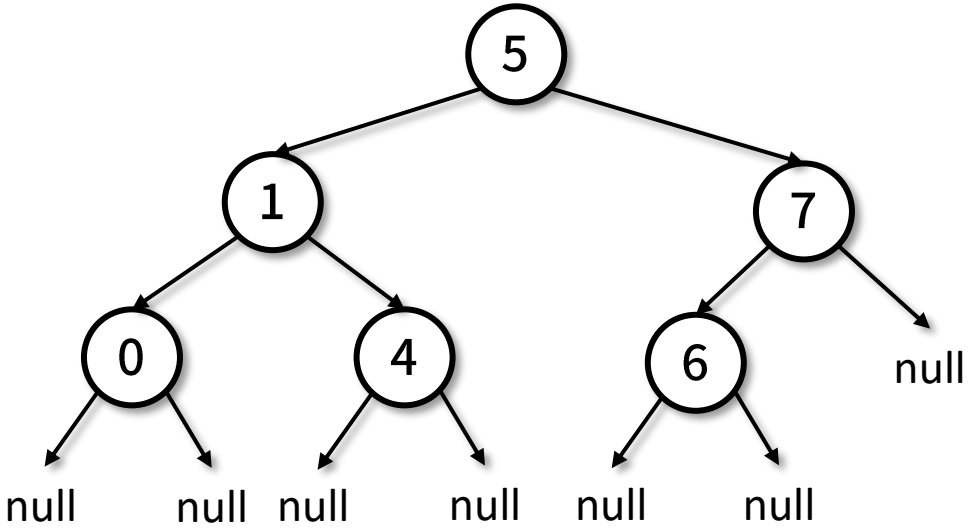
insertion: 1, 2, 3, 4, 5

average search time: 3T

Adelson-Velsky Landis Tree (AVL Tree)

AVL Tree (1)

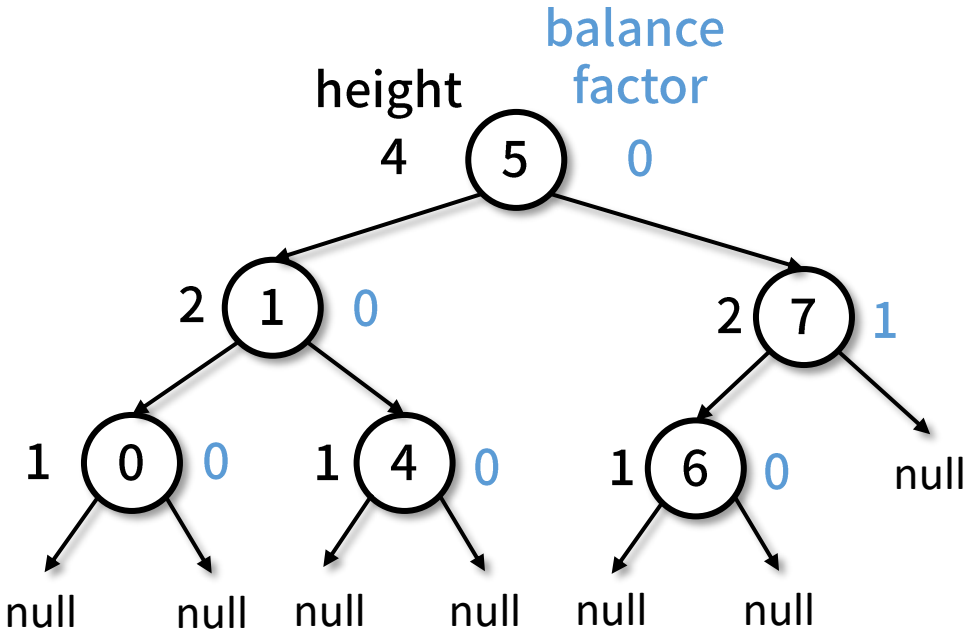
- The AVL tree is a balanced **binary search tree.**
- Any subtree in the AVL tree is also an AVL tree.
- Need additional operations for *insert* and *delete* to maintain the structure to ensure the rules are invariant.



AVL Tree (2)

- Define the balance factor as:
  - $height(left\ subtree) - height(right\ subtree)$

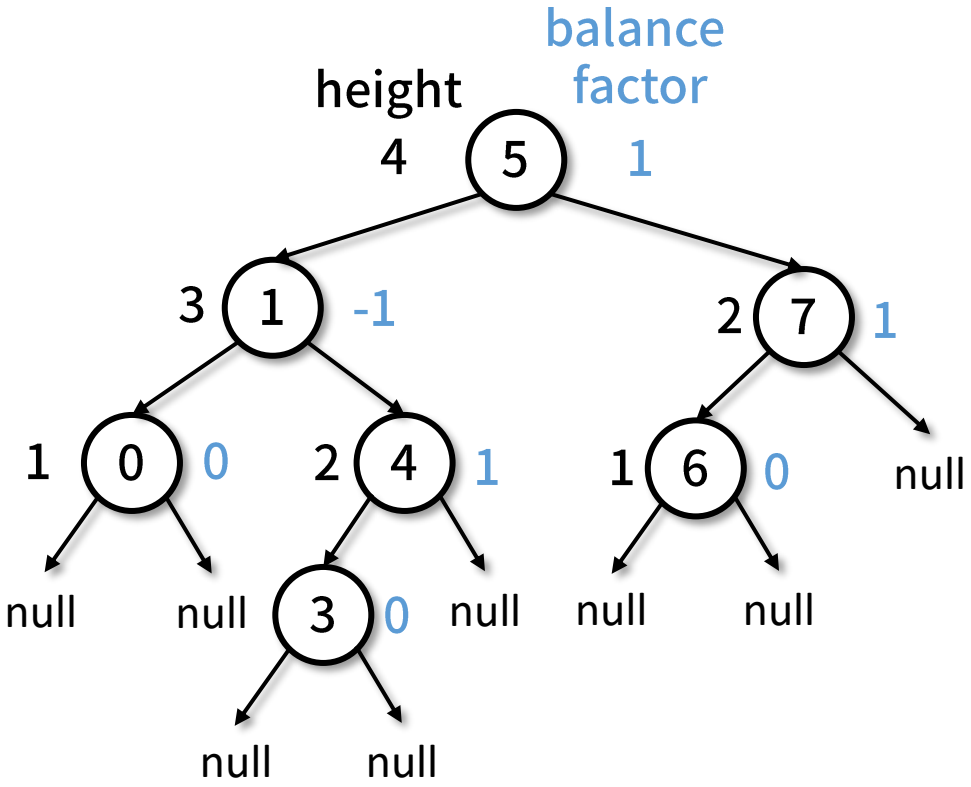
- Balance Tree: the balance factor of every node inside is either -1, 0 or 1.



AVL Tree

Insertion (1)

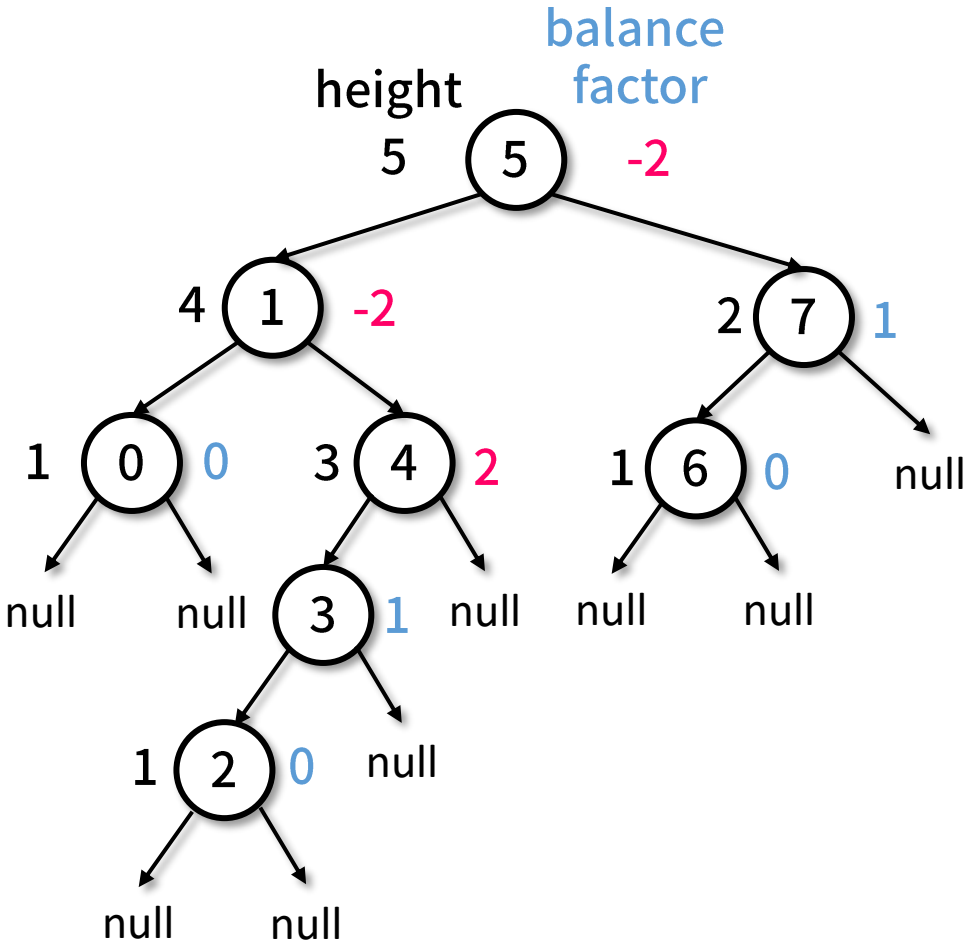
- Define the balance factor as:
  - $height(left\ subtree) - height(right\ subtree)$
- Balance Tree: the balance factor of every node inside is either -1, 0 or 1.
- Let us try insert a node with key 3.



AVL Tree

Insertion (2)

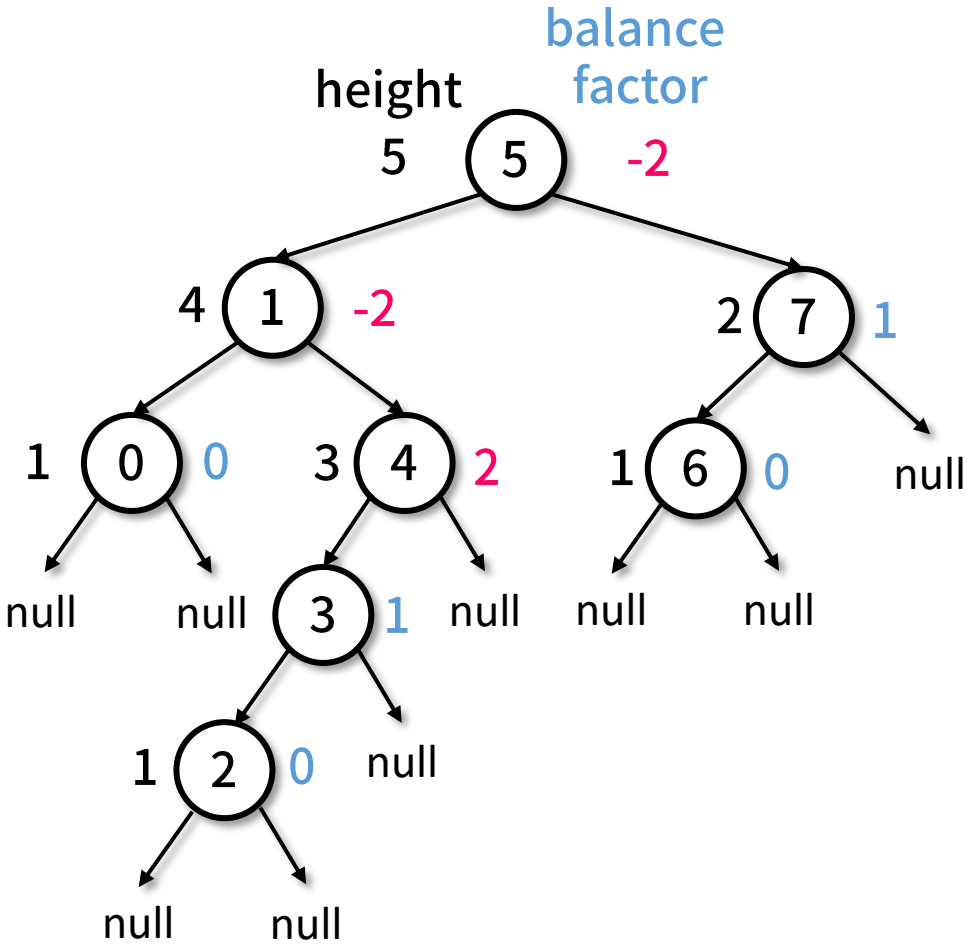
- Define the balance factor as:
  - $height(left\ subtree) - height(right\ subtree)$
- Balance Tree: the balance factor of every node inside is either -1, 0 or 1.
- Let us try insert a node with key 3.
- Let us try insert a node with key 2. The balance is broken.



AVL Tree

Insertion (3)

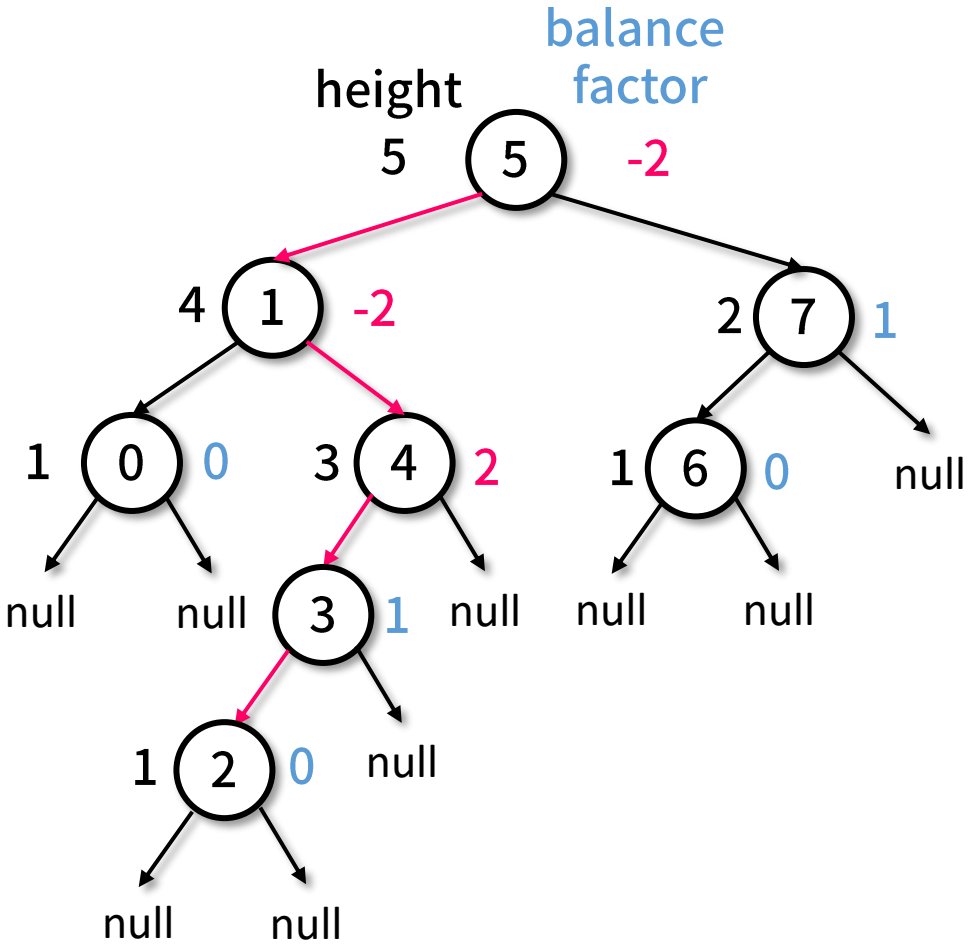
- Define the balance factor as:
  - $height(left\ subtree) - height(right\ subtree)$
- Balance Tree: the balance factor of every node inside is either -1, 0 or 1.
- In what situation the balance will be broken?



AVL Tree

Insertion (4)

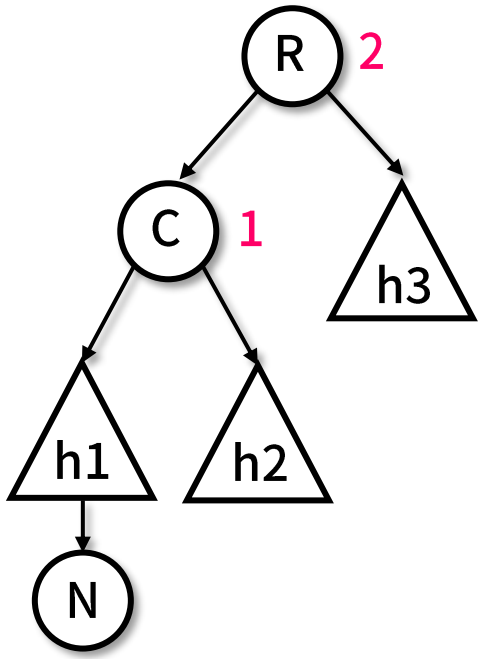
- Define the balance factor as:
  - $height(left\ subtree) - height(right\ subtree)$
- Balance Tree: the balance factor of every node inside is either -1, 0 or 1.
- In what situation the balance will be broken?
- What can we observe from the example?



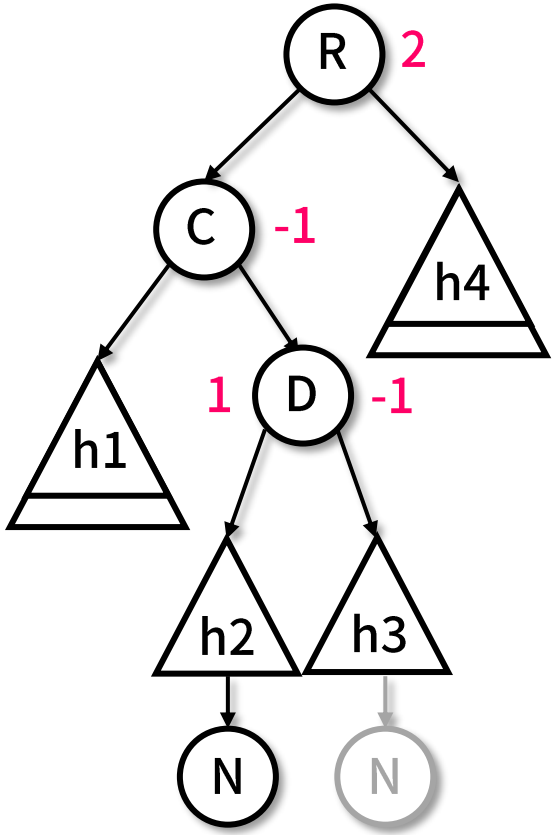
AVL Tree

Insertion (5)

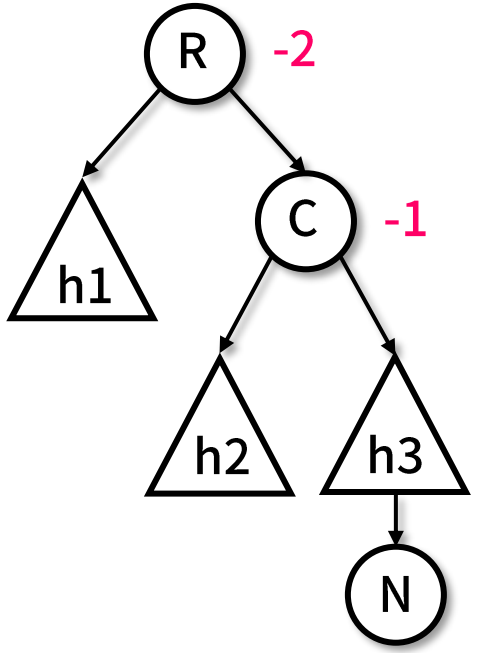
Left Left



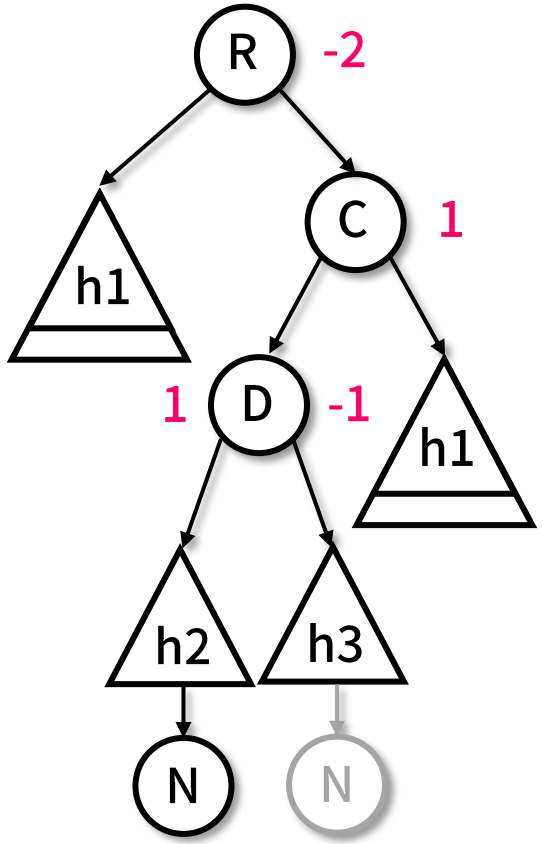
Left Right



Right Right



Right Left

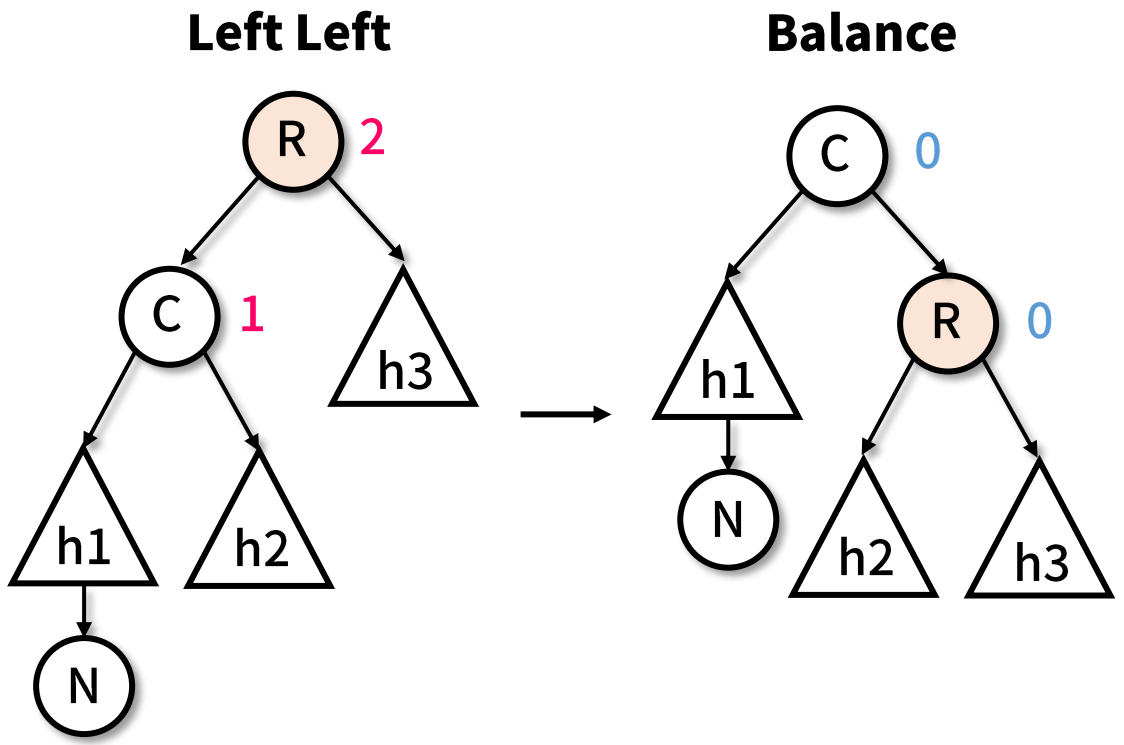


AVL Tree

Right Rotation

Right Rotation

```python
def right_rotate(self, node: AVLTreeNode) -> AVLTreeNode:

    x = node.left
    y = x.right

    x.right = node
    node.left = y

    node.height = 1 + max(
        self.get_node_height(node.left),
        self.get_node_height(node.right))
    x.height = 1 + max(
        self.get_node_height(x.left),
        self.get_node_height(x.right))

    return x
```



the height remains the same after correction

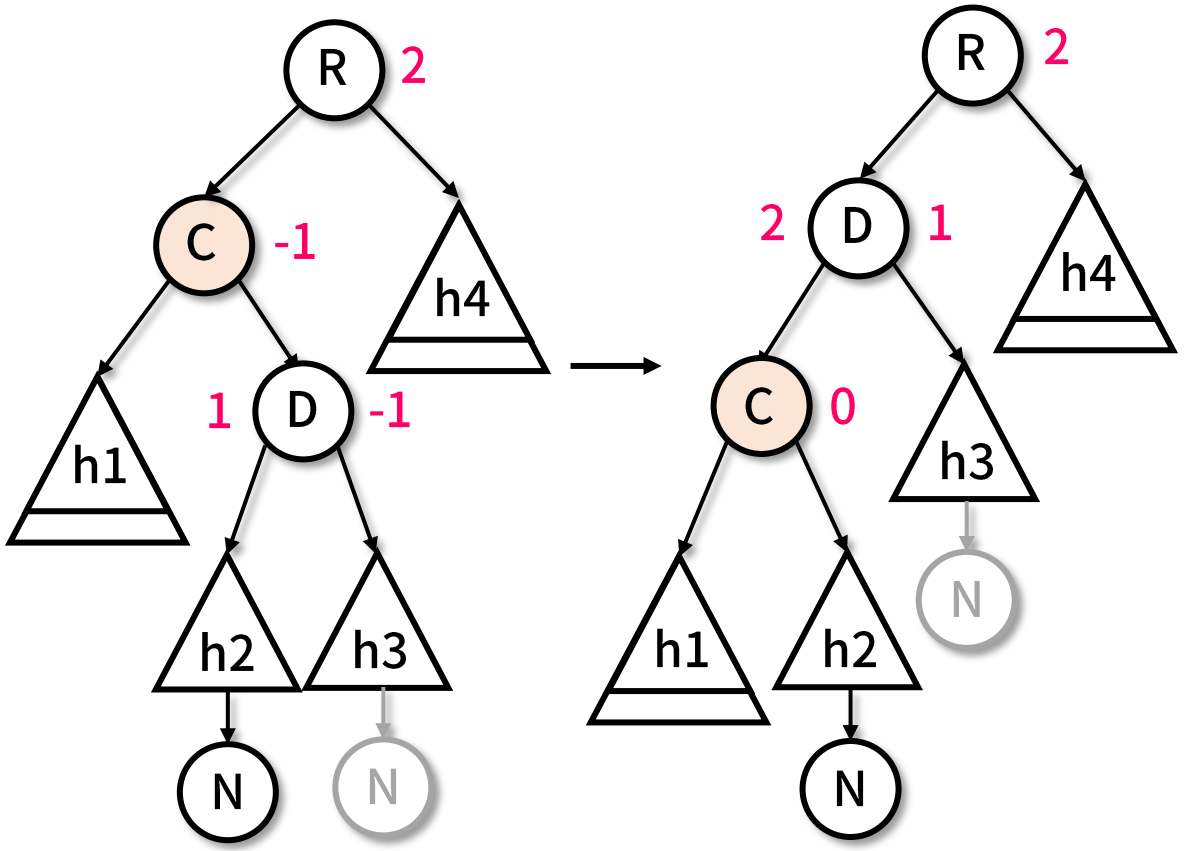
AVL Tree

Left Rotation

Left Rotation

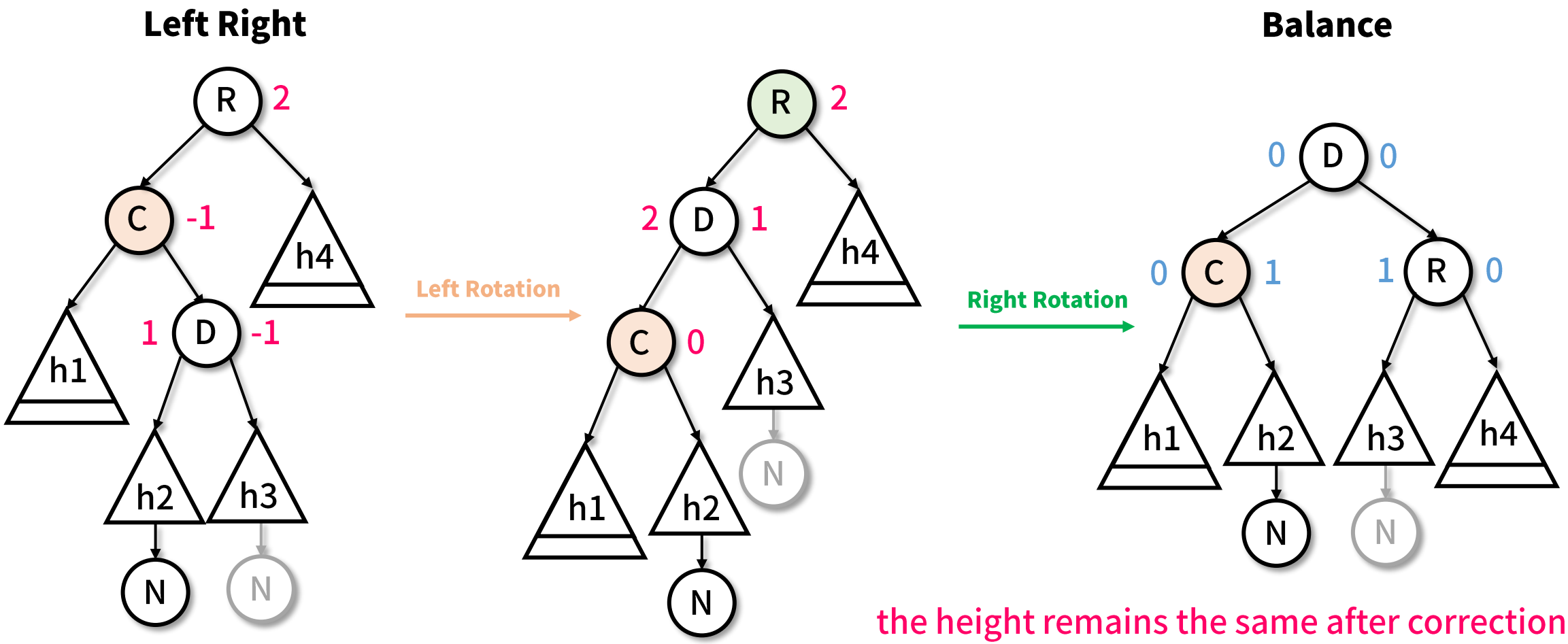
```python
def left_rotate(self, node: AVLTreeNode) -> AVLTreeNode:

    x = node.right
    y = x.left

    x.left = node
    node.right = y

    node.height = 1 + max(
        self.get_node_height(node.left),
        self.get_node_height(node.right))
    x.height = 1 + max(
        self.get_node_height(x.left),
        self.get_node_height(x.right))

    return x
```

Left Right



AVL Tree

Rotation



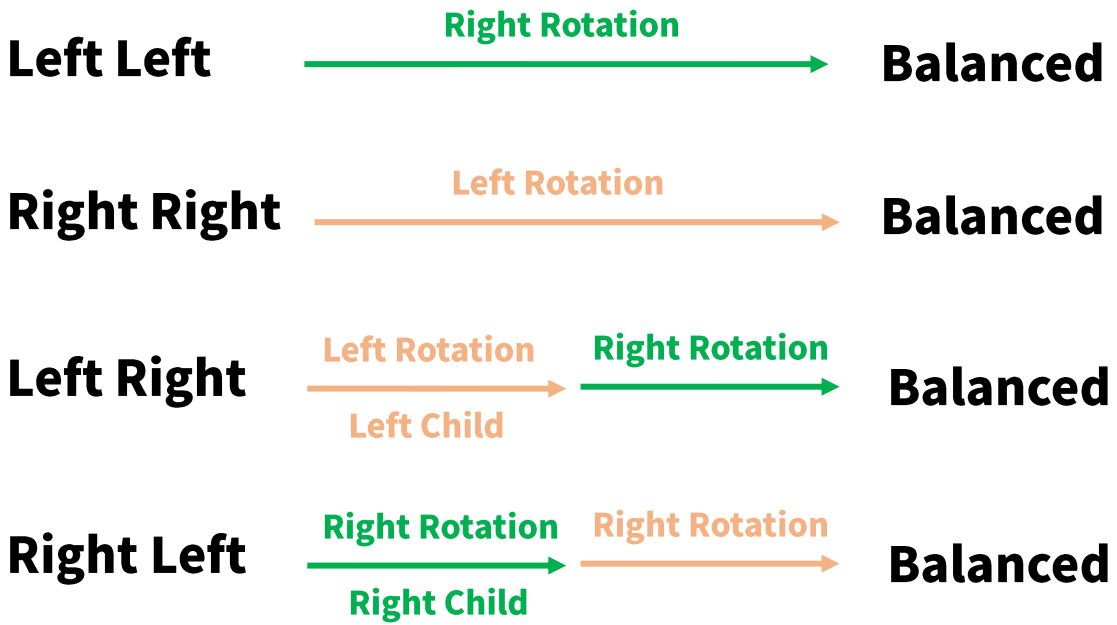
the height remains the same after correction

AVL Tree

Insertion Correction

Insertion Correction

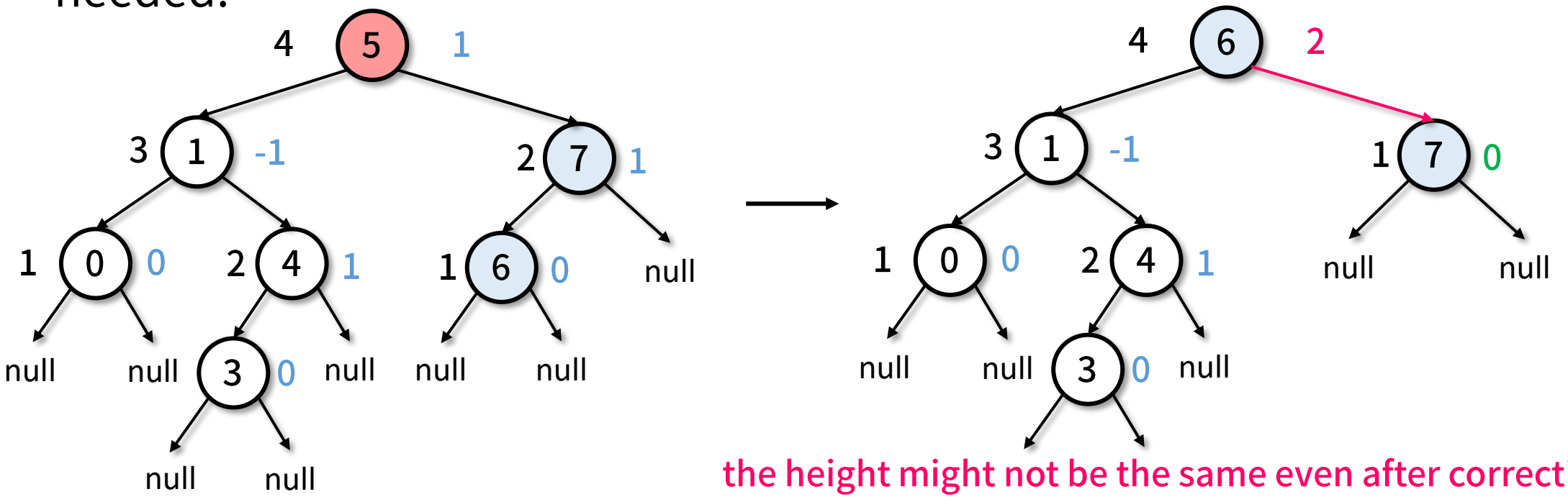
```
if balance_factor > 1 and key < node.left.key:
    return self.right_rotate(node)

if balance_factor < -1 and key > node.right.key:
    return self.left_rotate(node)

if balance_factor > 1 and key > node.left.key:
    node.left = self.left_rotate(node.left)
    return self.right_rotate(node)

if balance_factor < -1 and key < node.right.key:
    node.right = self.right_rotate(node.right)
    return self.left_rotate(node)
```



AVL Tree

Deletion (1)

- Perform the ordinary binary search tree deletion.
- Check all the nodes iterated in the process, correct the structure if needed.

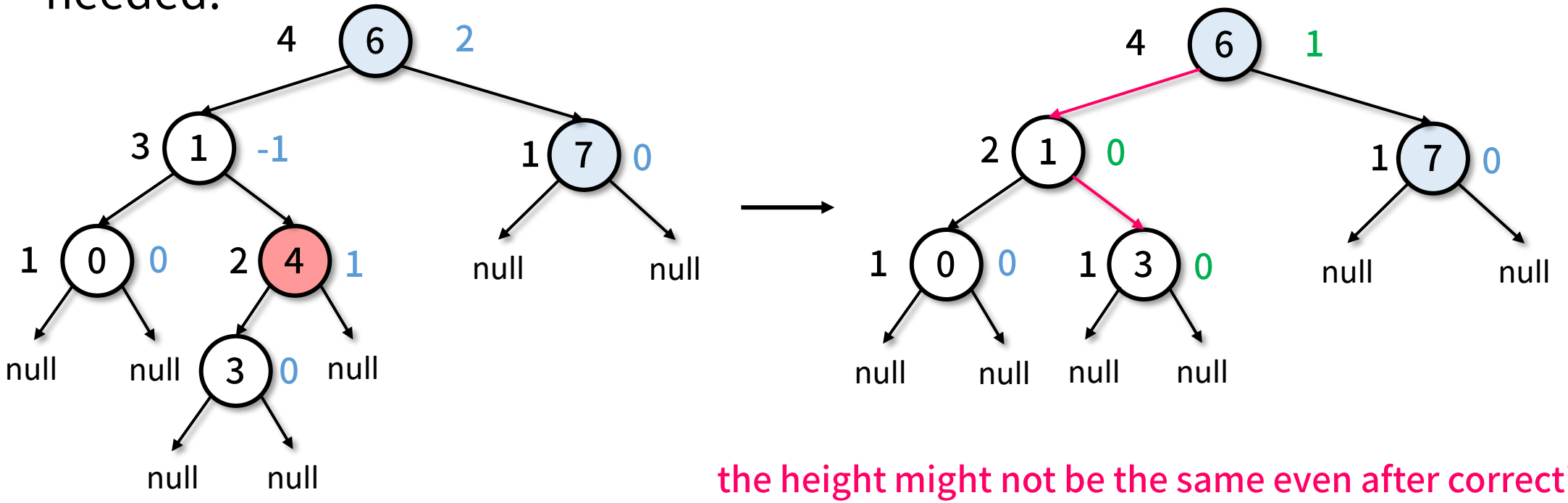


the height might not be the same even after correction, need to check through all node in the iteration.

AVL Tree

Deletion (2)

- Perform the ordinary binary search tree deletion.
- Check all the nodes iterated in the process, correct the structure if needed.



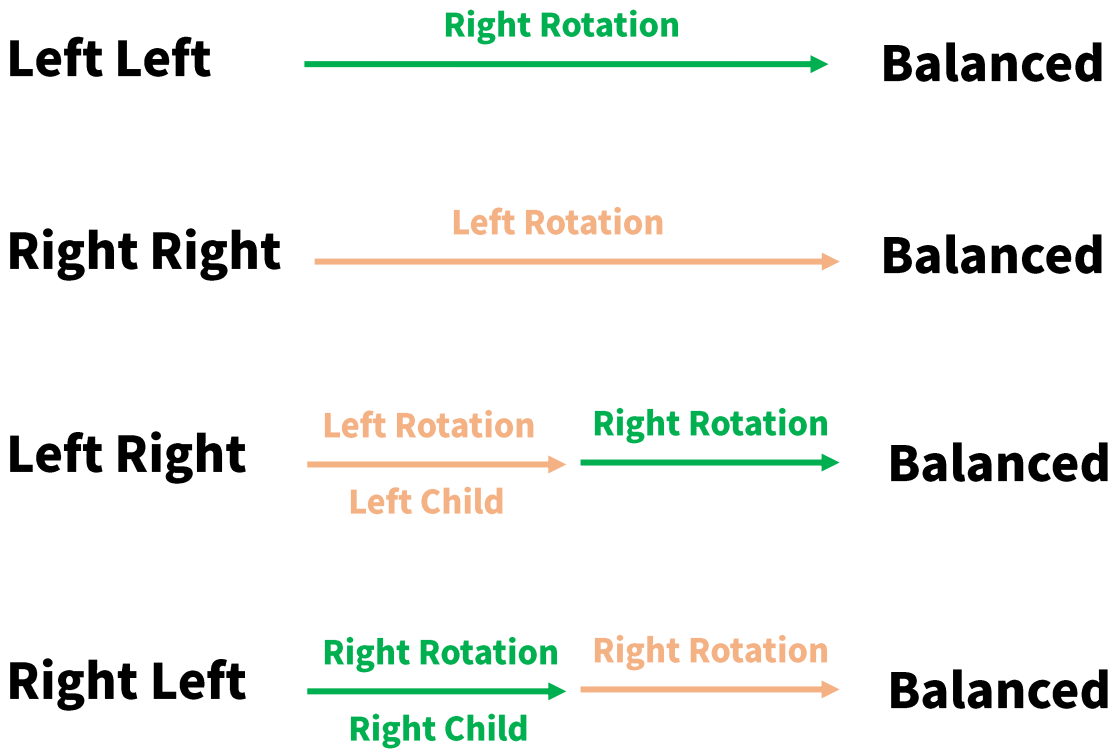
the height might not be the same even after correction, need to check through all node in the iteration.

AVL Tree

Deletion Correction

Deletion Correction

```
if ((balance_factor > 1) and
    (self.get_balance_factor(node.left) >= 0)):
    return self.right_rotate(node)

if ((balance_factor < -1) and
    (self.get_balance_factor(node.right) <= 0)):
    return self.left_rotate(node)

if ((balance_factor > 1) and
    (self.get_balance_factor(node.left) < 0)):
    node.left = self.left_rotate(node.left)
    return self.right_rotate(node)

if ((balance_factor < -1) and
    (self.get_balance_factor(node.right) > 0)):
    node.right = self.right_rotate(node.right)
    return self.left_rotate(node)
```

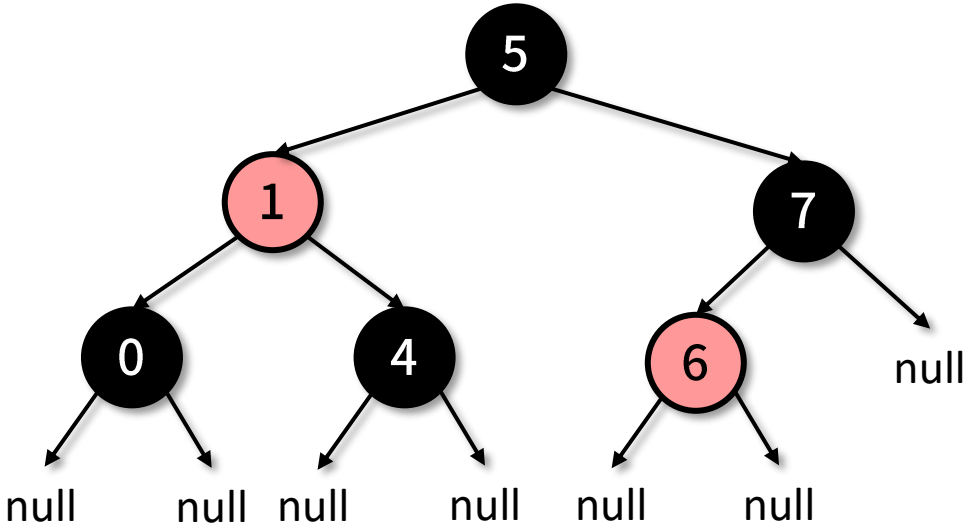


Red-Black Tree

Red-Black Tree (1)

- Red-black tree rules:
  - Node can be either black or red.
  - Root node is always black.
  - Null node is considered as black.
  - The number of black node from the root to the leaf is the same.
  - Red node can only have black children.
  - Black node can have black or red children.

- Need additional operations for *insert* and *delete* to maintain the structure to ensure the rules are invariant.

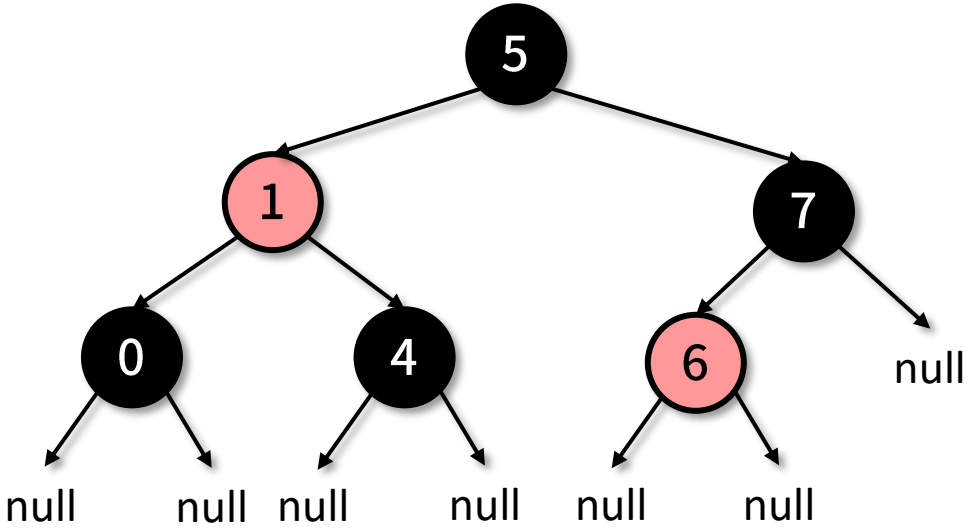


Red-Black Tree (2)

- Red-black tree properties:
  - Any subtree with a black root is also a red-black tree.
  - The maximum height difference between the two subtree is

$$2 \times \min(height_{left}, height_{right})$$

  - Not strictly balanced binary search tree.

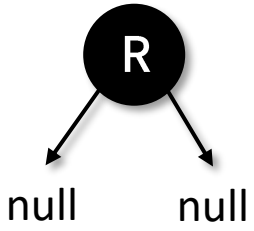


Red-Black Tree

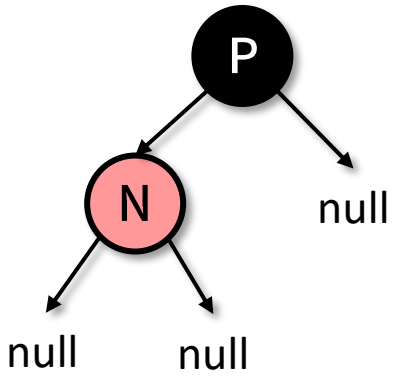
Insertion (1)

- We always add a red node using ordinary insertion of binary search tree, and then correct the tree if rules are broken.

case 1:
empty tree



case 2:
add to a black parent

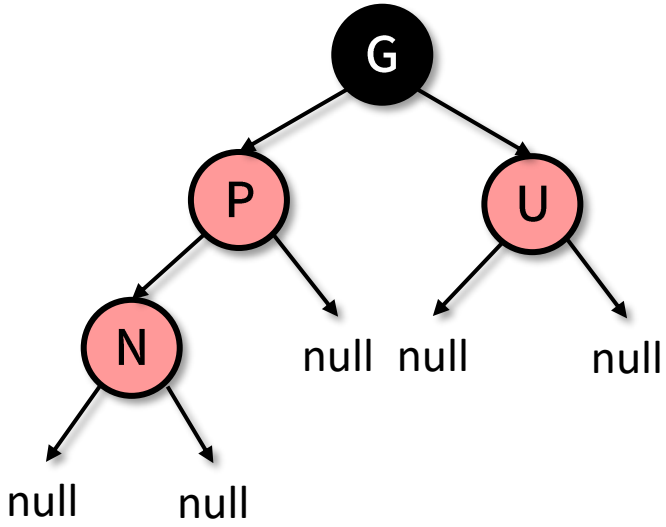


the number of black nodes to the leaf node remains the same

Red-Black Tree

Insertion (3)

case 3:
add to a red parent with a red uncle

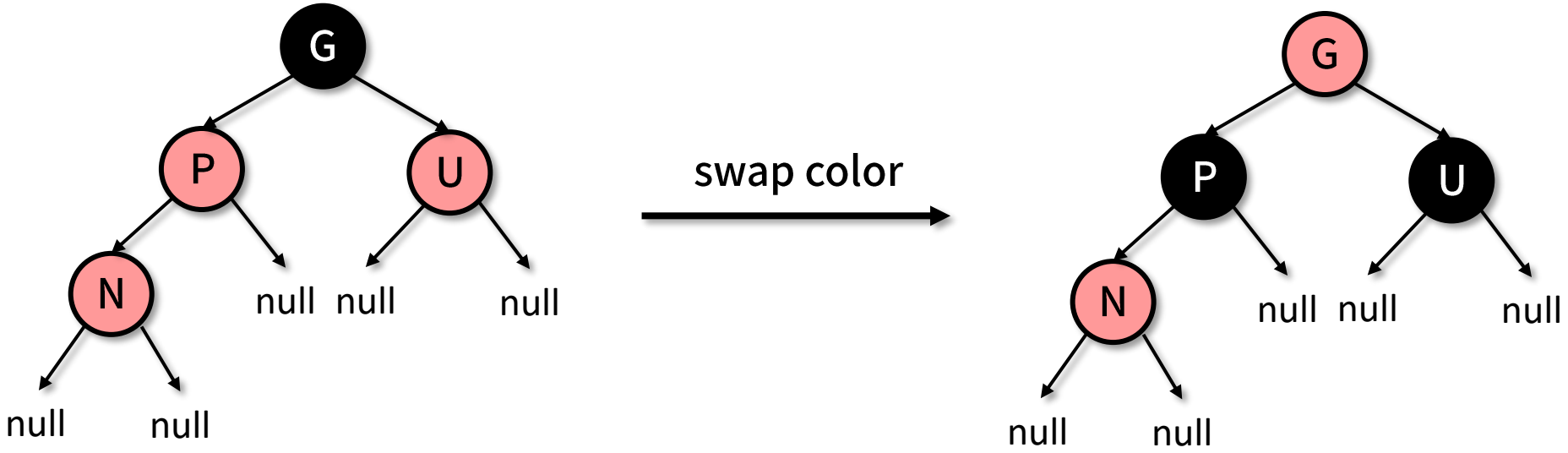


Red-Black Tree

Insertion (4)

case 3:
add to a red parent with a red uncle

iterate to parent of grandparent
to check the rules



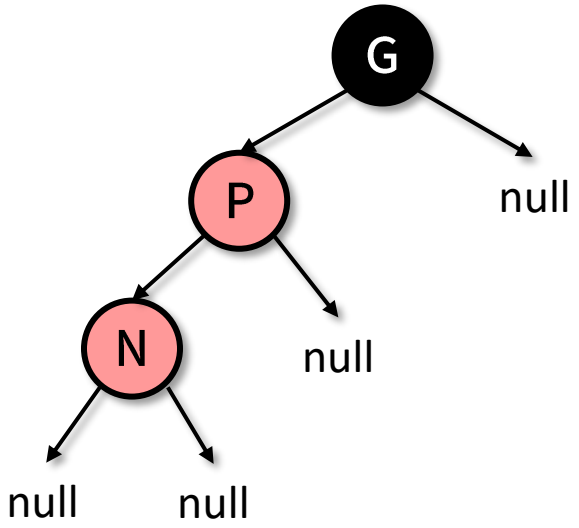
the number of black nodes to the leaf node remains the same

Red-Black Tree

Insertion (5)

case 4: left left

1. **add to a red parent without uncle**
2. **new node is a left child of the parent**
3. **parent is a left child of the grandparent**

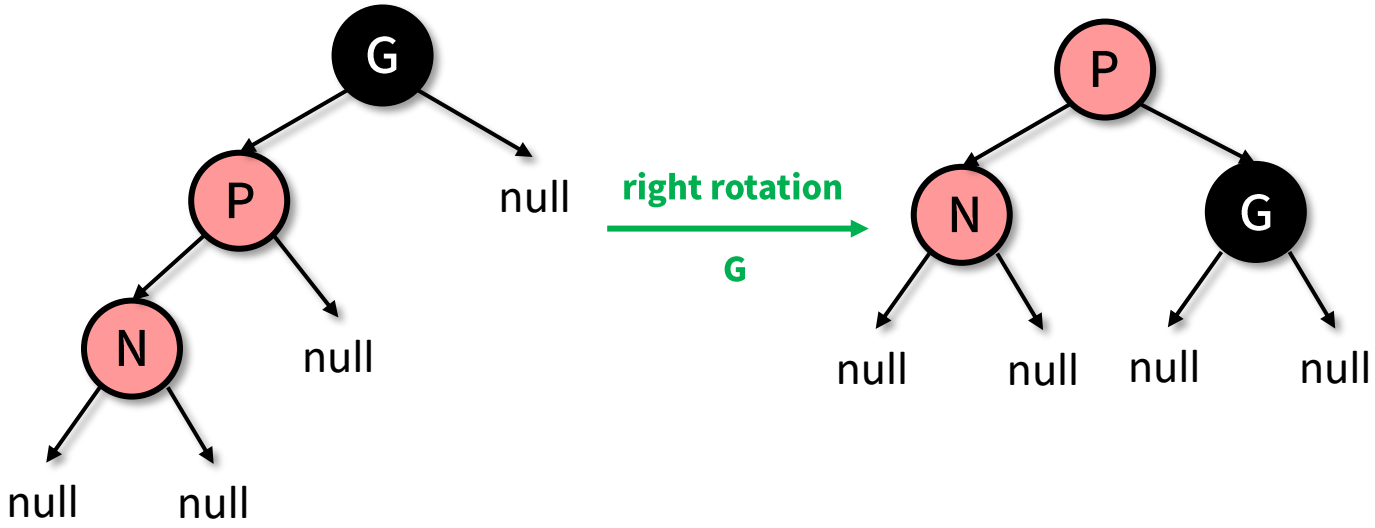


Red-Black Tree

Insertion (6)

case 4: left left

1. **add to a red parent without uncle**
2. **new node is a left child of the parent**
3. **parent is a left child of the grandparent**

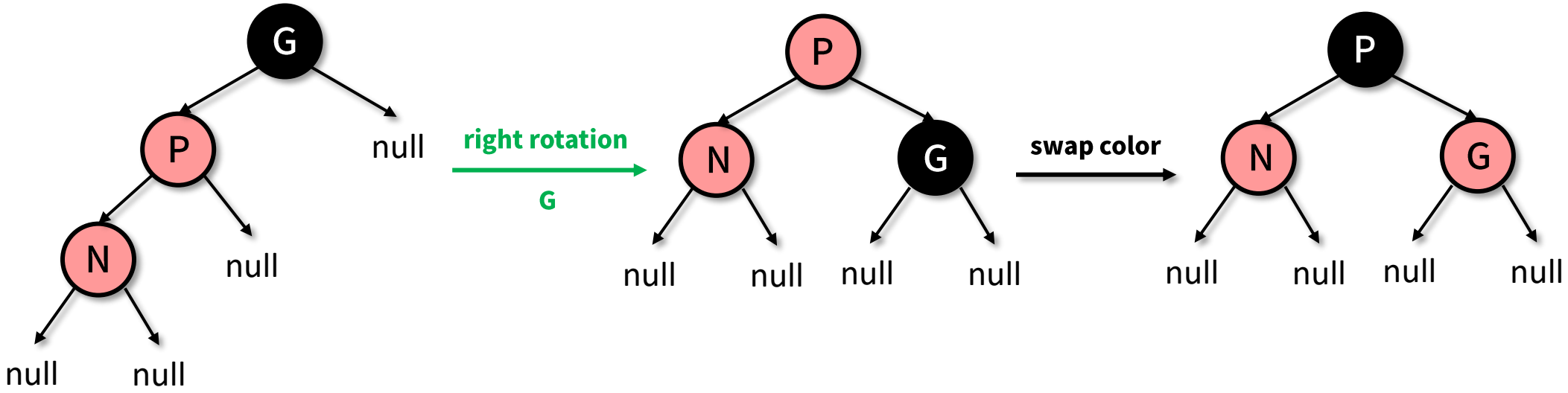


Red-Black Tree

Insertion (7)

case 4: left left

1. **add to a red parent without uncle**
2. **new node is a left child of the parent**
3. **parent is a left child of the grandparent**



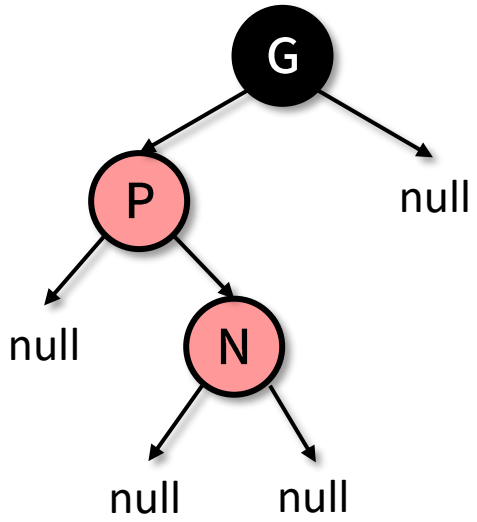
the number of black nodes to the leaf node remains the same

Red-Black Tree

Insertion (8)

case 5: left right

1. **add to a red parent without uncle**
2. **new node is a right child of the parent**
3. **parent is a left child of the grandparent**

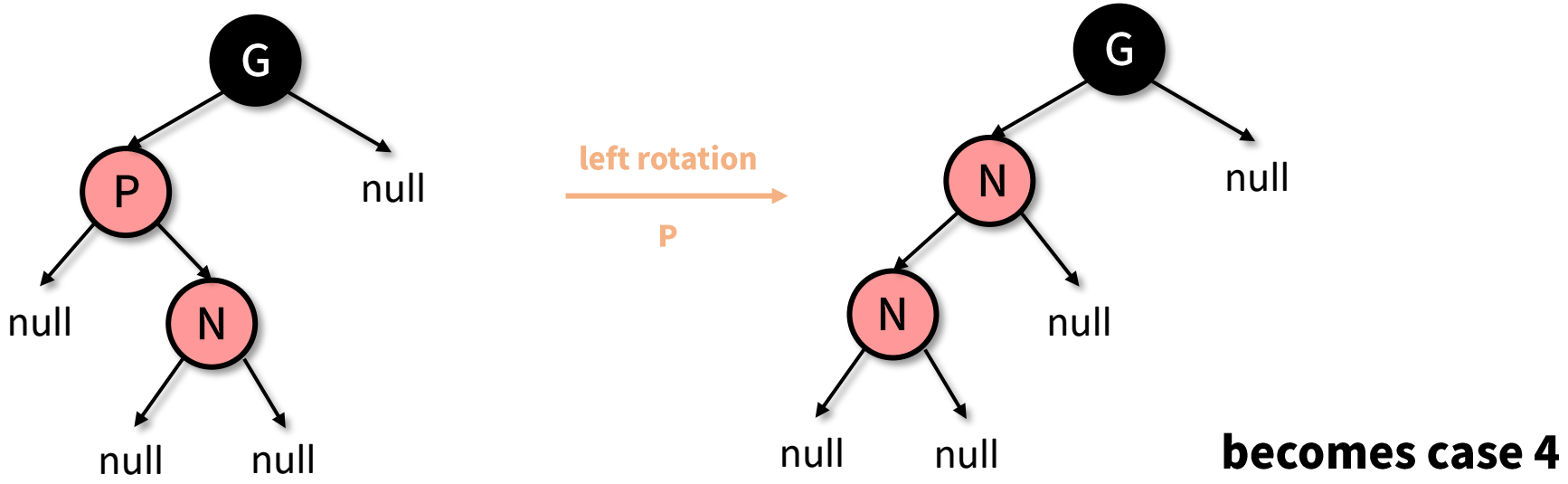


Red-Black Tree

Insertion (9)

case 5: left right

1. **add to a red parent without uncle**
2. **new node is a right child of the parent**
3. **parent is a left child of the grandparent**



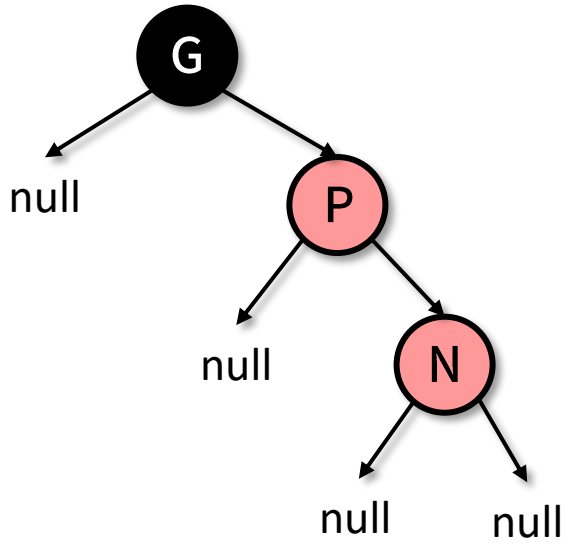
becomes case 4

Red-Black Tree

Insertion (10)

case 6: right right

1. **add to a red parent without uncle**
2. **new node is a right child of the parent**
3. **parent is a right child of the grandparent**

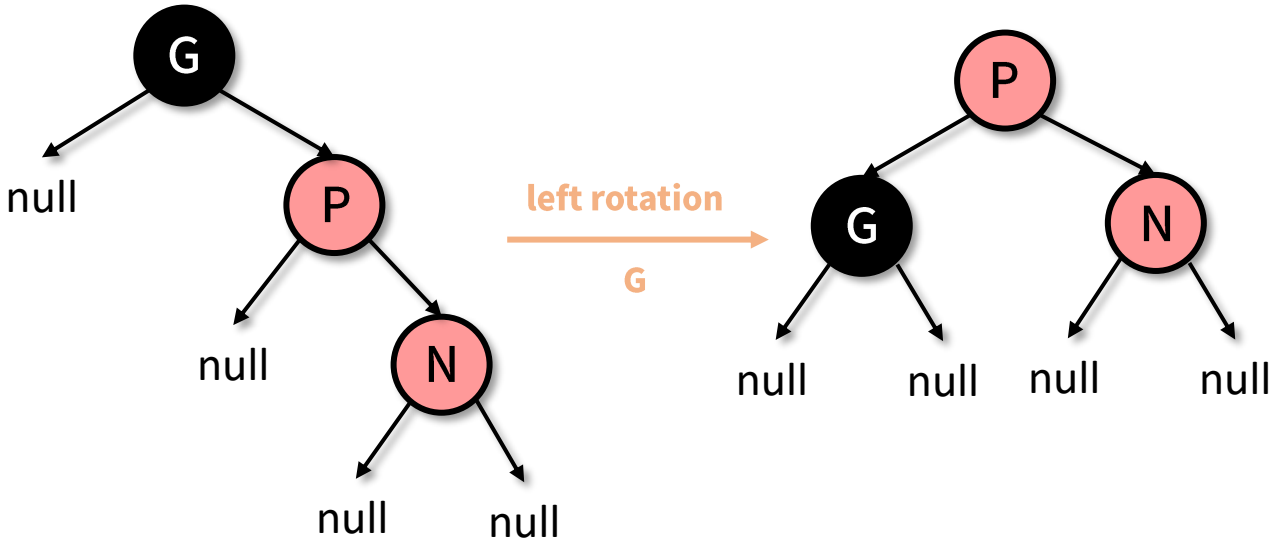


Red-Black Tree

Insertion (11)

case 6: right right

1. **add to a red parent without uncle**
2. **new node is a right child of the parent**
3. **parent is a right child of the grandparent**

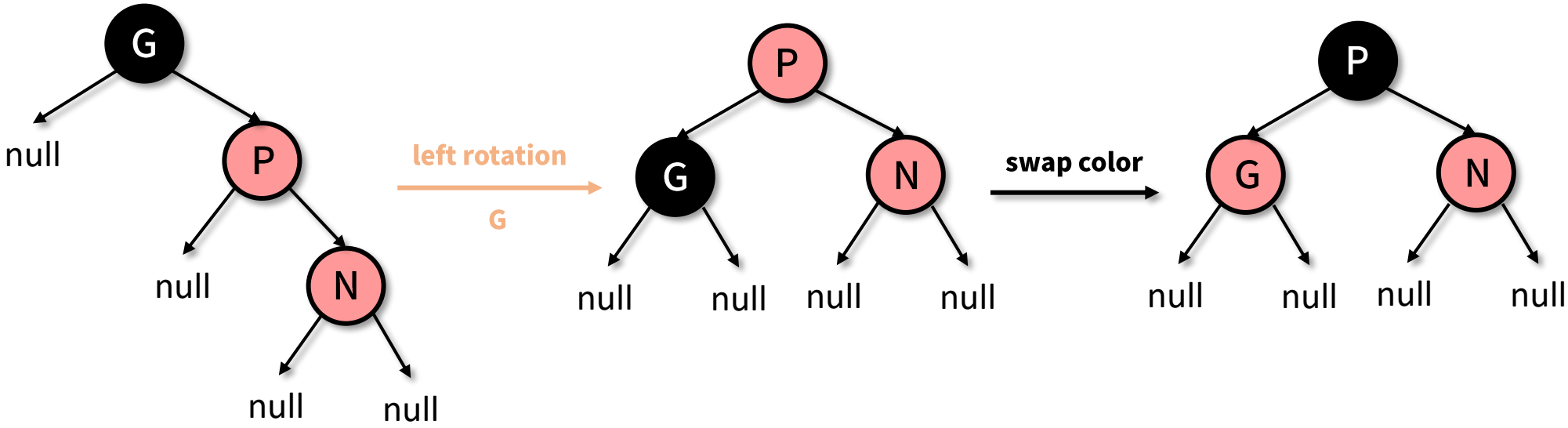


Red-Black Tree

Insertion (12)

case 6: right right

1. **add to a red parent without uncle**
2. **new node is a right child of the parent**
3. **parent is a right child of the grandparent**



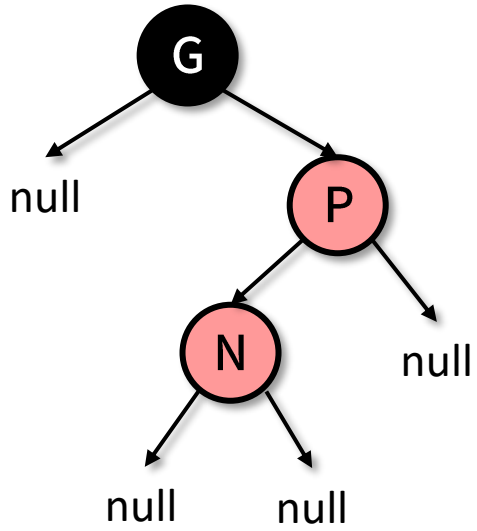
the number of black nodes to the leaf node remains the same

Red-Black Tree

Insertion (13)

case 7: right left

1. **add to a red parent without uncle**
2. **new node is a left child of the parent**
3. **parent is a right child of the grandparent**

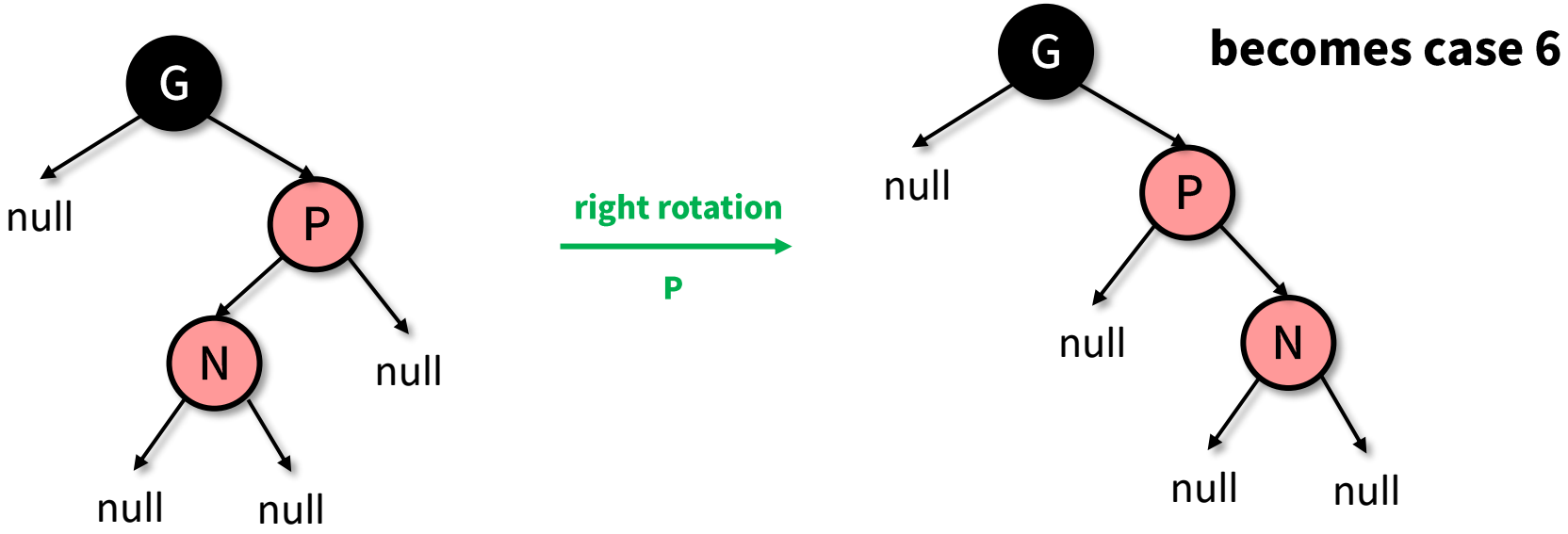


Red-Black Tree

Insertion (14)

case 7: right left

1. **add to a red parent without uncle**
2. **new node is a left child of the parent**
3. **parent is a right child of the grandparent**

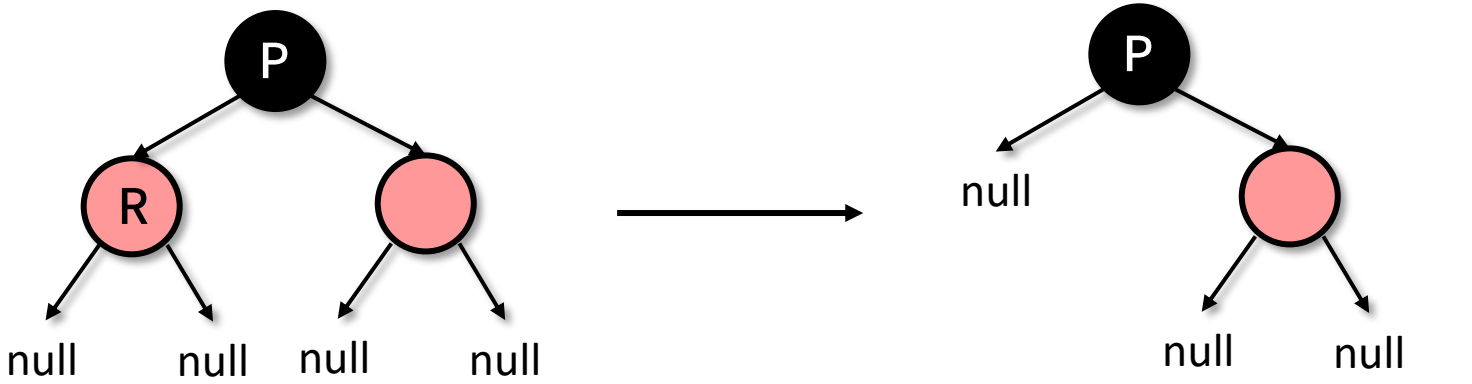


Red-Black Tree

Deletion (1)

- Need custom operations for different cases, not using the ordinary deletion in binary search tree.

case 1: remove any red leaf node



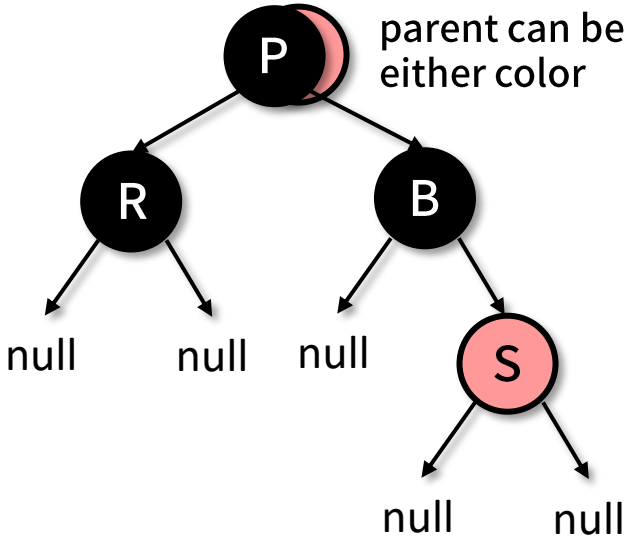
no need for correction

Red-Black Tree

Deletion (2)

case 2:

1. **remove black leaf node**
2. **the brother is a black node**
3. **the brother has only one red right child**

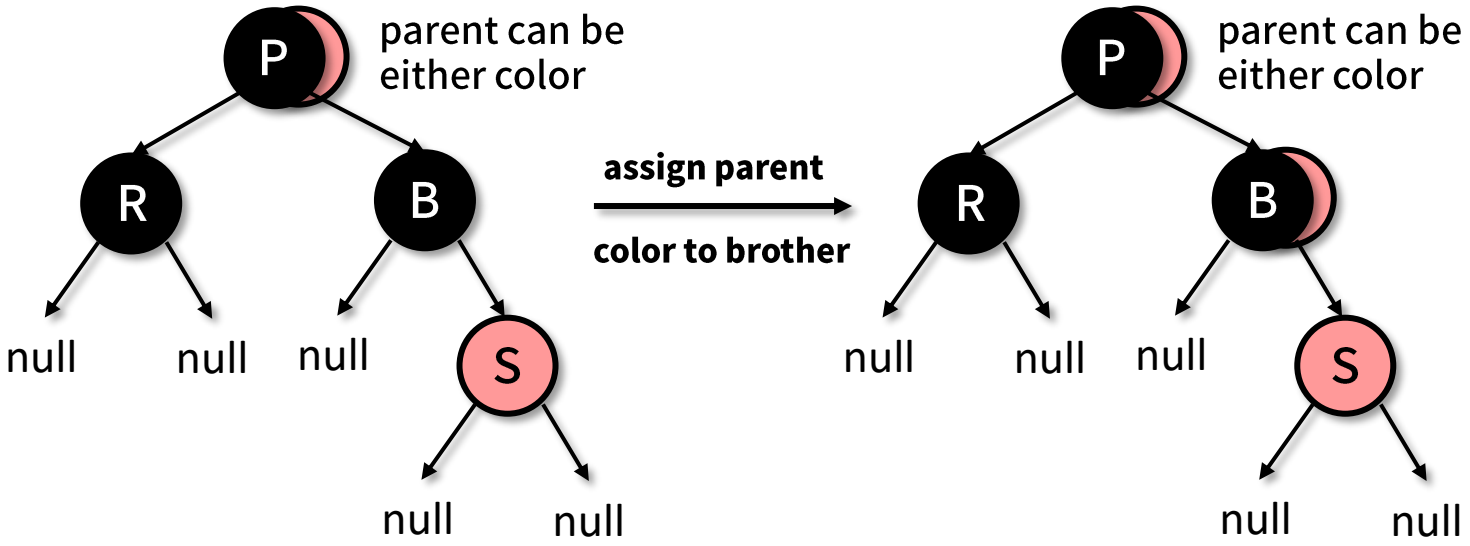


Red-Black Tree

Deletion (3)

case 2:

1. **remove black leaf node**
2. **the brother is a black node**
3. **the brother has only one red right child**

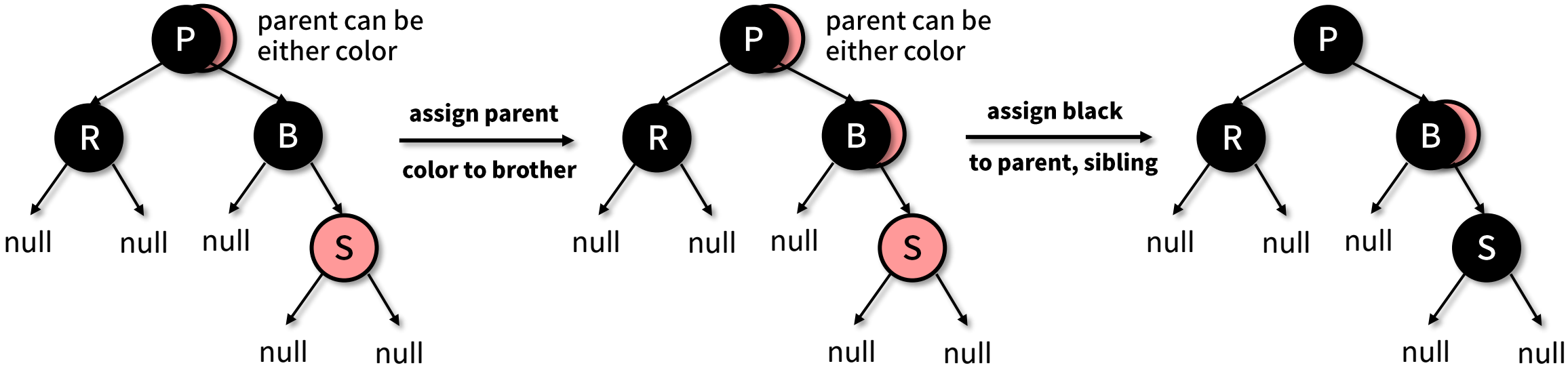


Red-Black Tree

Deletion (4)

case 2:

1. **remove black leaf node**
2. **the brother is a black node**
3. **the brother has only one red right child**

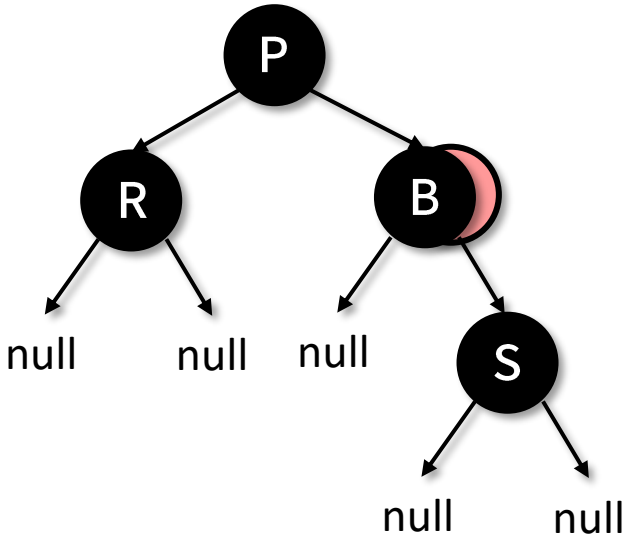


Red-Black Tree

Deletion (5)

case 2:

1. **remove black leaf node**
2. **the brother is a black node**
3. **the brother has only one red right child**

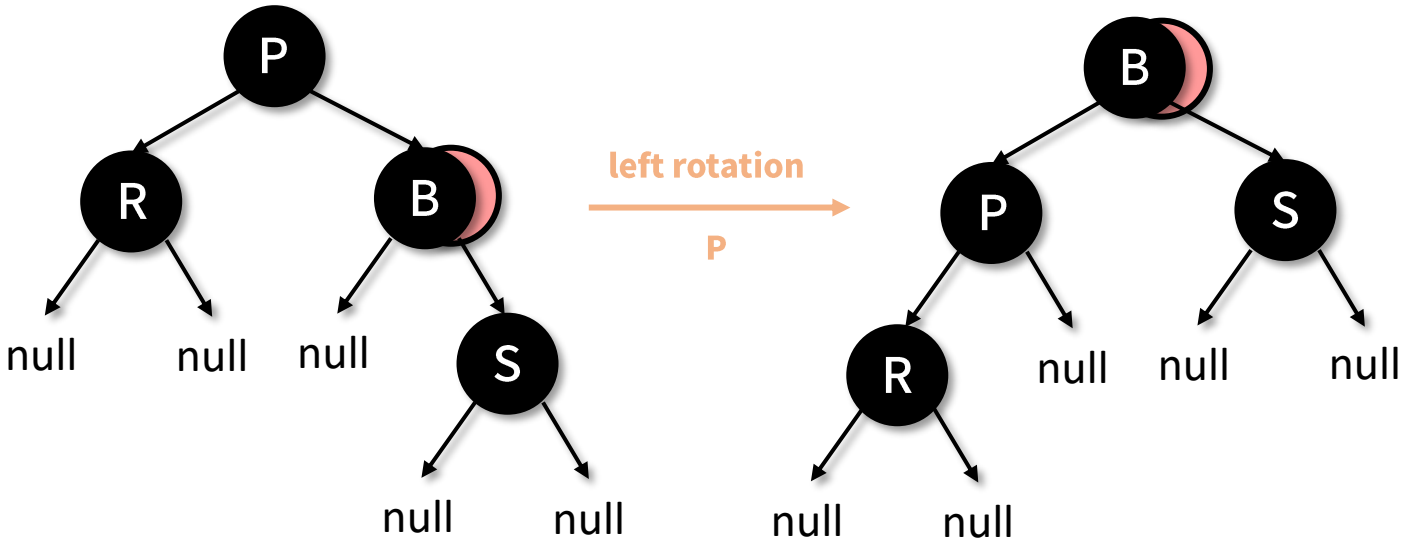


Red-Black Tree

Deletion (6)

case 2:

1. **remove black leaf node**
2. **the brother is a black node**
3. **the brother has only one red right child**

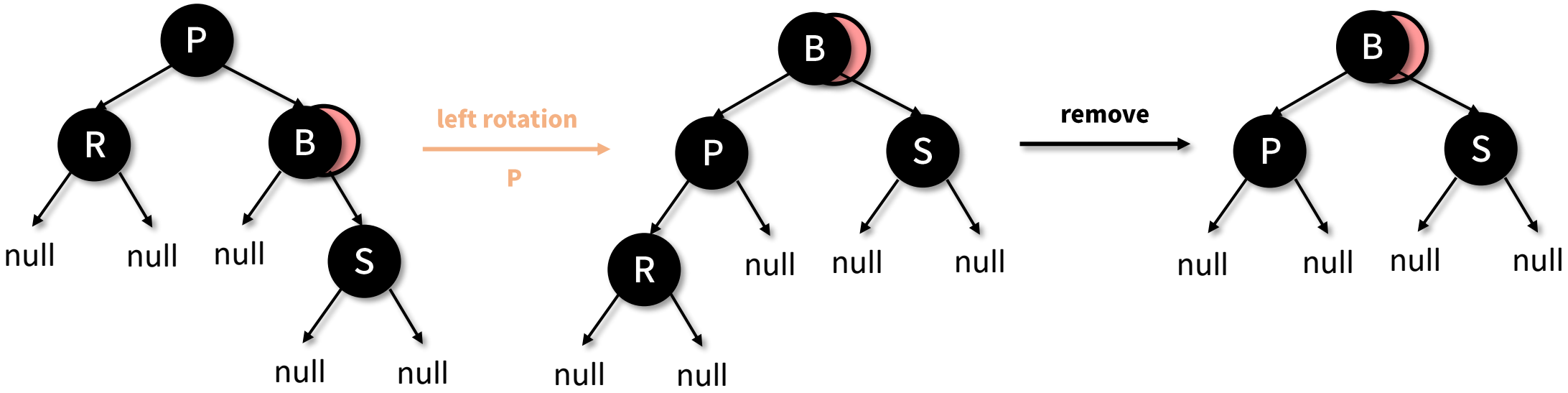


Red-Black Tree

Deletion (7)

case 2:

1. **remove black leaf node**
2. **the brother is a black node**
3. **the brother has only one red right child**



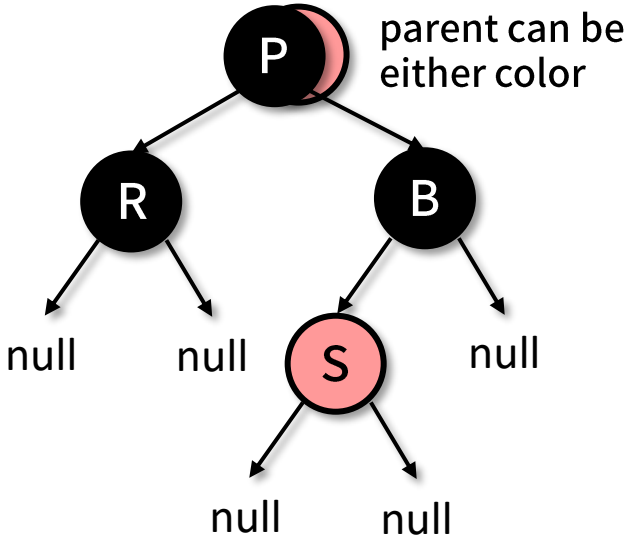
the number of black nodes to the leaf node remains the same

Red-Black Tree

Deletion (8)

case 3:

1. **remove black leaf node**
2. **the brother is a black node**
3. **the brother has only one red left child**

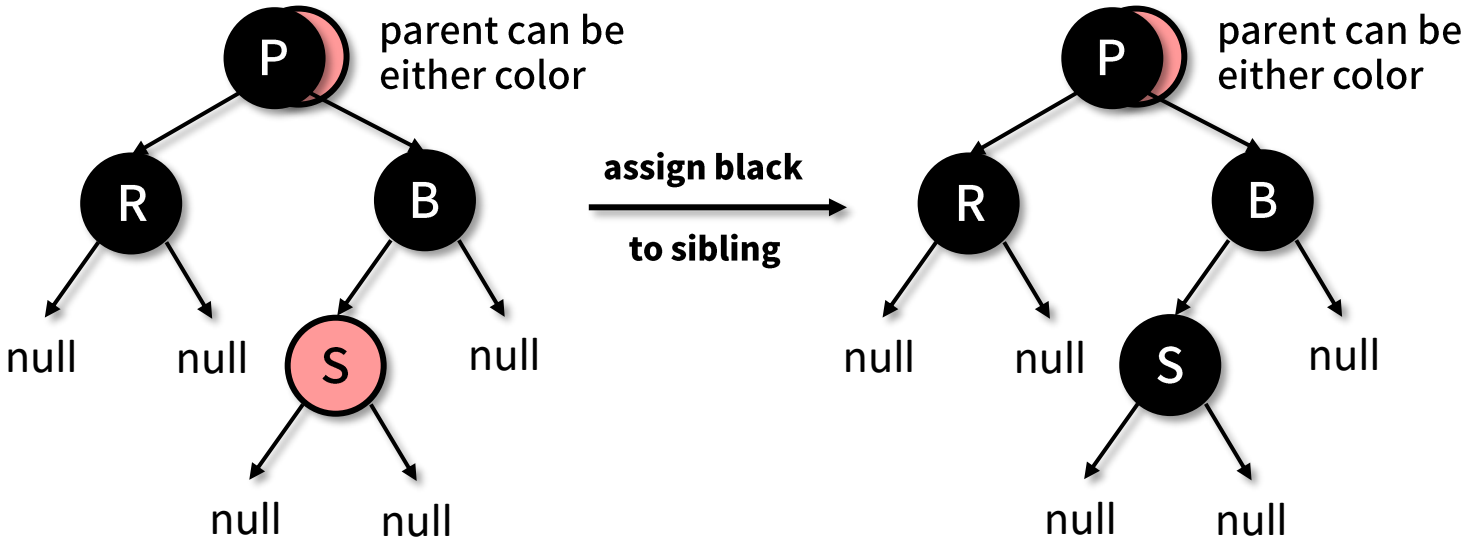


Red-Black Tree

Deletion (9)

case 3:

1. **remove black leaf node**
2. **the brother is a black node**
3. **the brother has only one red left child**

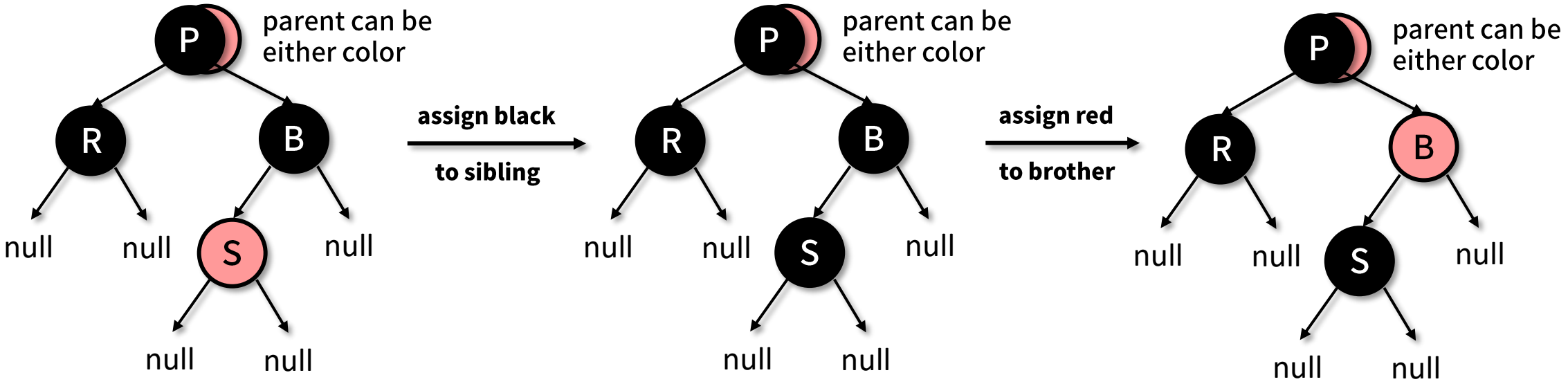


Red-Black Tree

Deletion (10)

case 3:

1. **remove black leaf node**
2. **the brother is a black node**
3. **the brother has only one red left child**

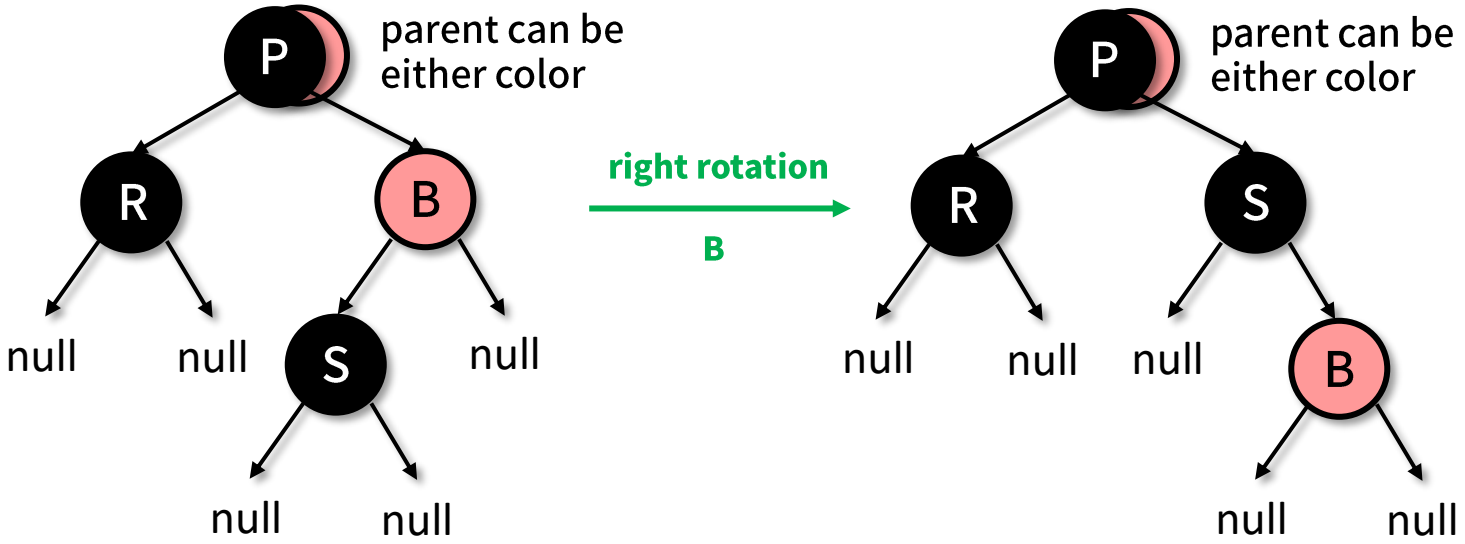


Red-Black Tree

Deletion (11)

case 3:

1. **remove black leaf node**
2. **the brother is a black node**
3. **the brother has only one red left child**



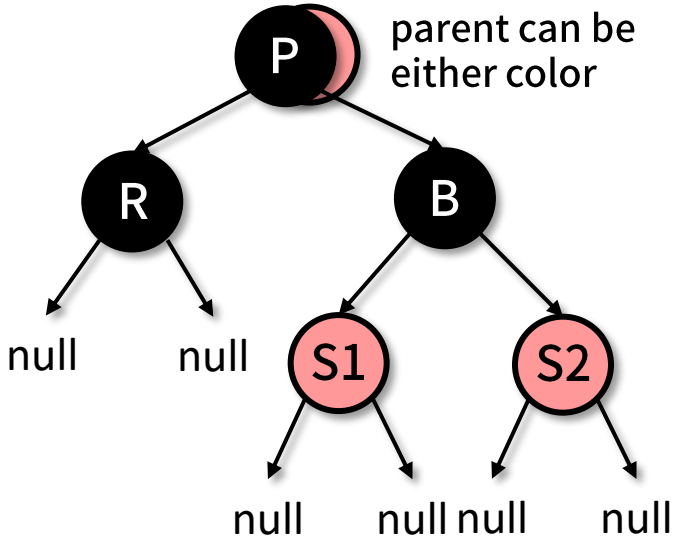
becomes case 2

Red-Black Tree

Deletion (12)

case 4:

1. **remove black leaf node**
2. **the brother is a black node**
3. **the brother has two red children**

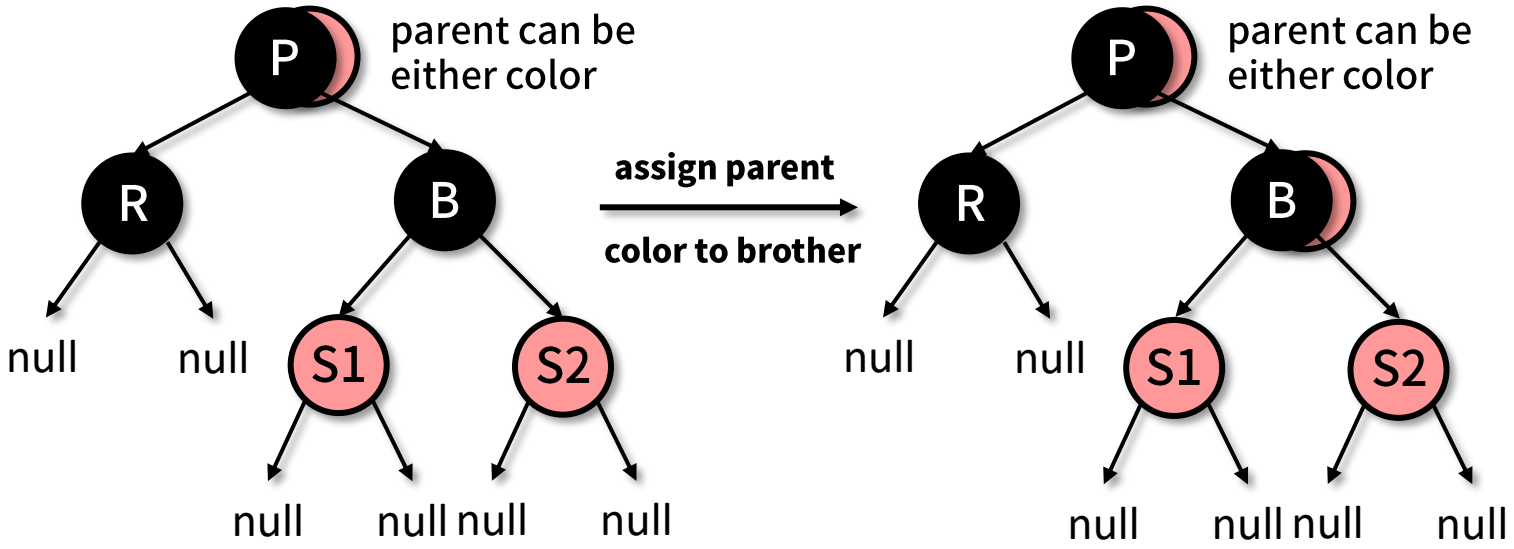


Red-Black Tree

Deletion (13)

case 4:

1. **remove black leaf node**
2. **the brother is a black node**
3. **the brother has two red children**

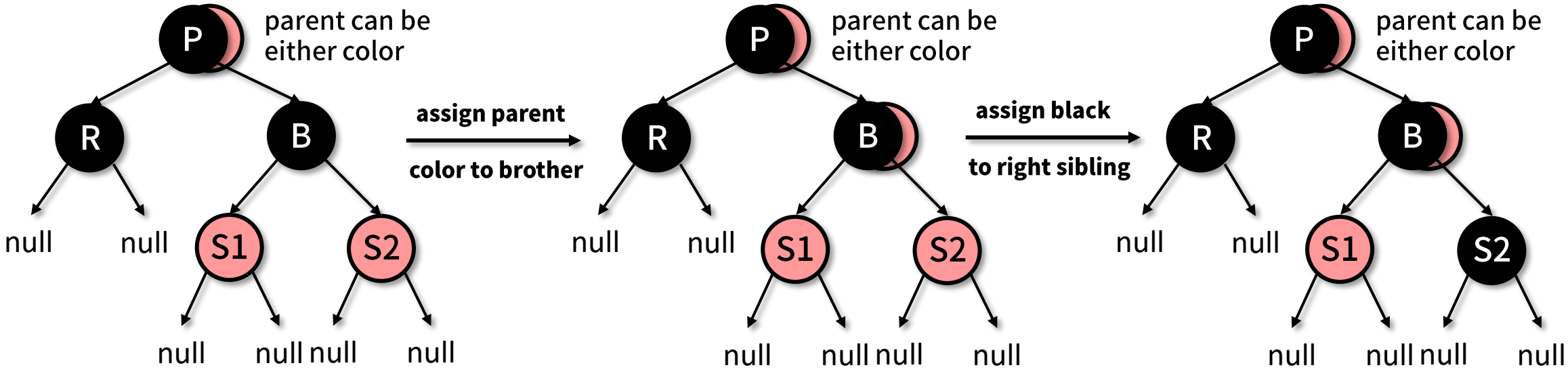


Red-Black Tree

Deletion (14)

case 4:

1. **remove black leaf node**
2. **the brother is a black node**
3. **the brother has two red children**

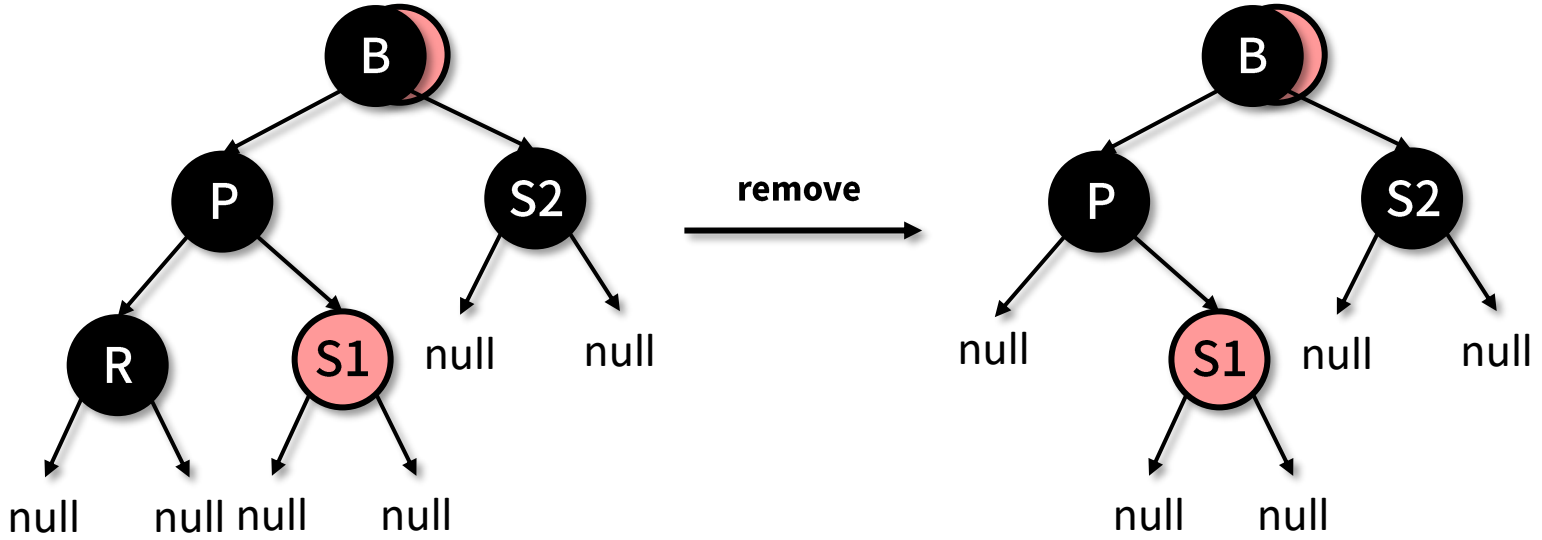


Red-Black Tree

Deletion (15)

case 4:

1. **remove black leaf node**
2. **the brother is a black node**
3. **the brother has two red children**



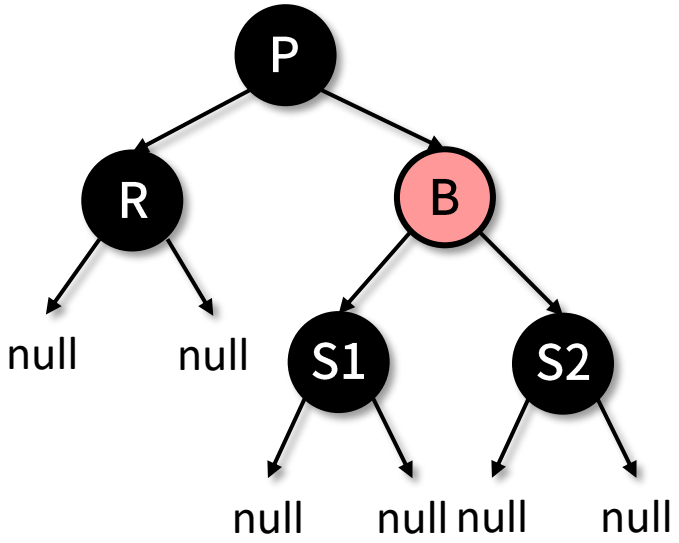
the number of black nodes to the leaf node remains the same

Red-Black Tree

Deletion (16)

case 5:

1. **remove black leaf node**
2. **the brother is a red node**
3. **the brother has two red children**

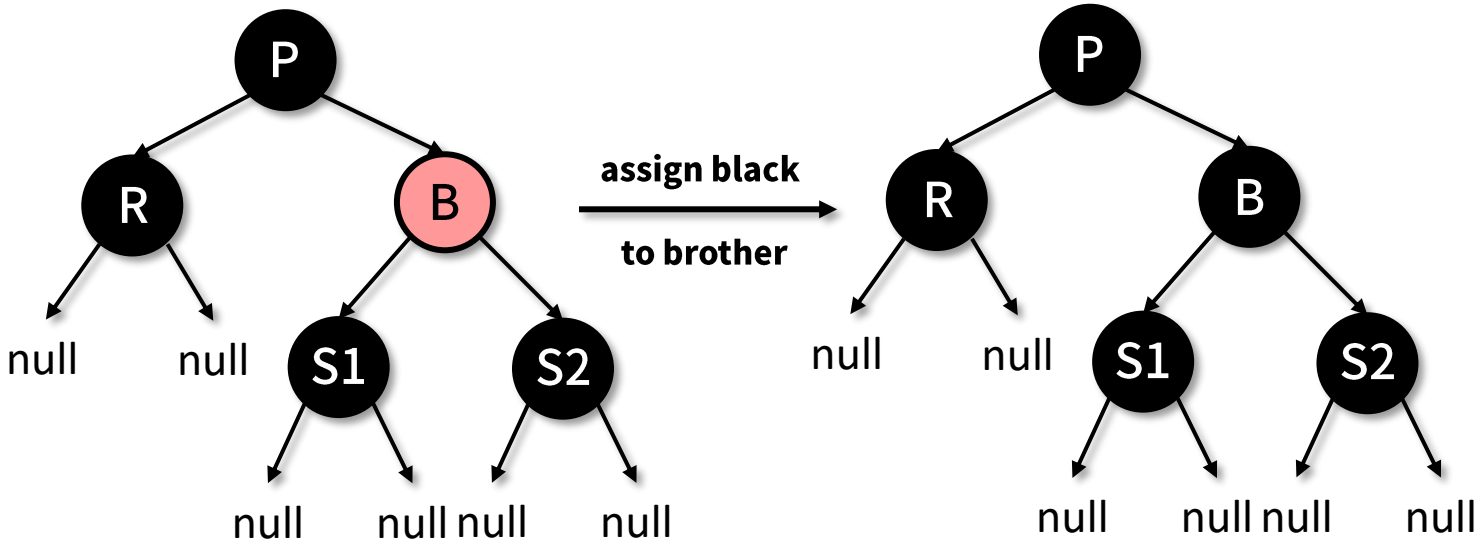


Red-Black Tree

Deletion (17)

case 5:

1. **remove black leaf node**
2. **the brother is a red node**
3. **the brother has two red children**

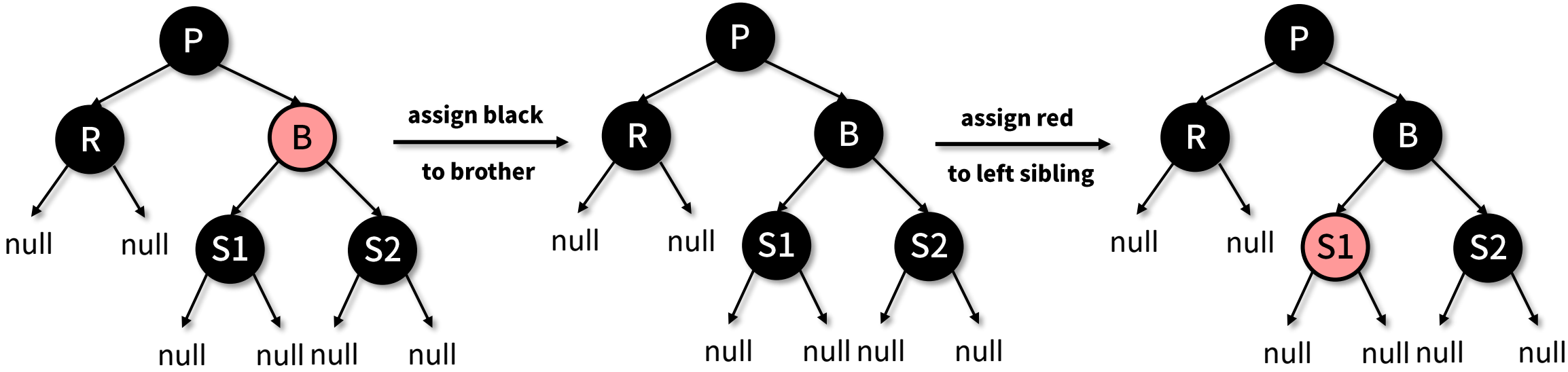


Red-Black Tree

Deletion (18)

case 5:

1. **remove black leaf node**
2. **the brother is a red node**
3. **the brother has two red children**

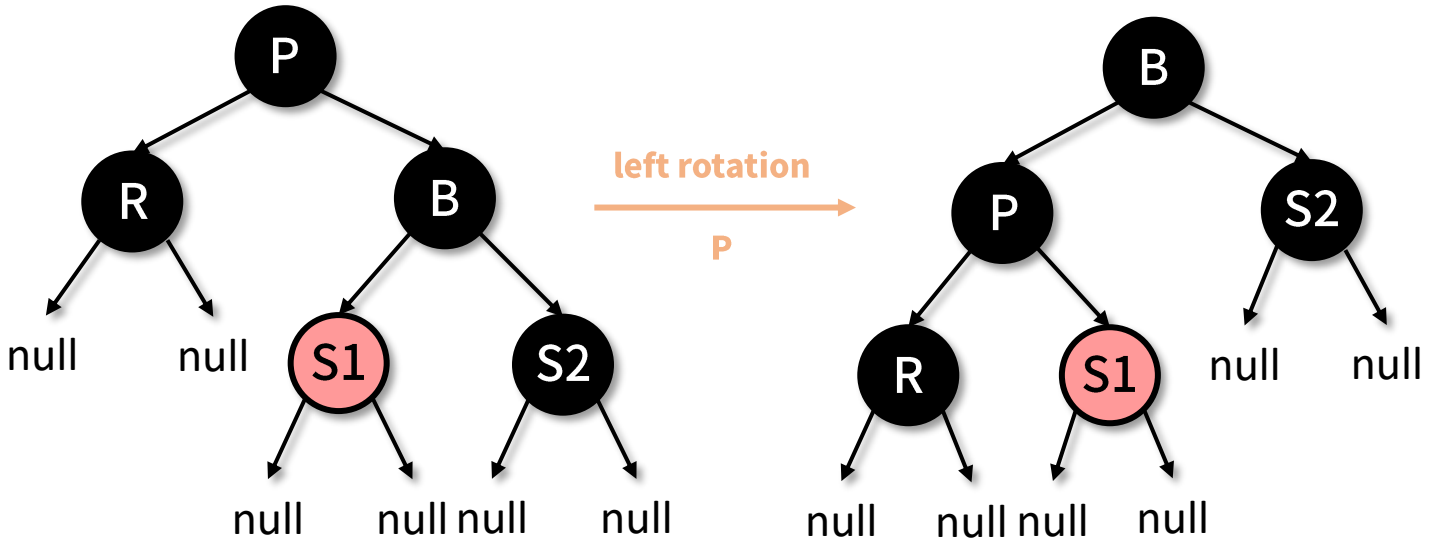


Red-Black Tree

Deletion (19)

case 5:

1. **remove black leaf node**
2. **the brother is a red node**
3. **the brother has two red children**

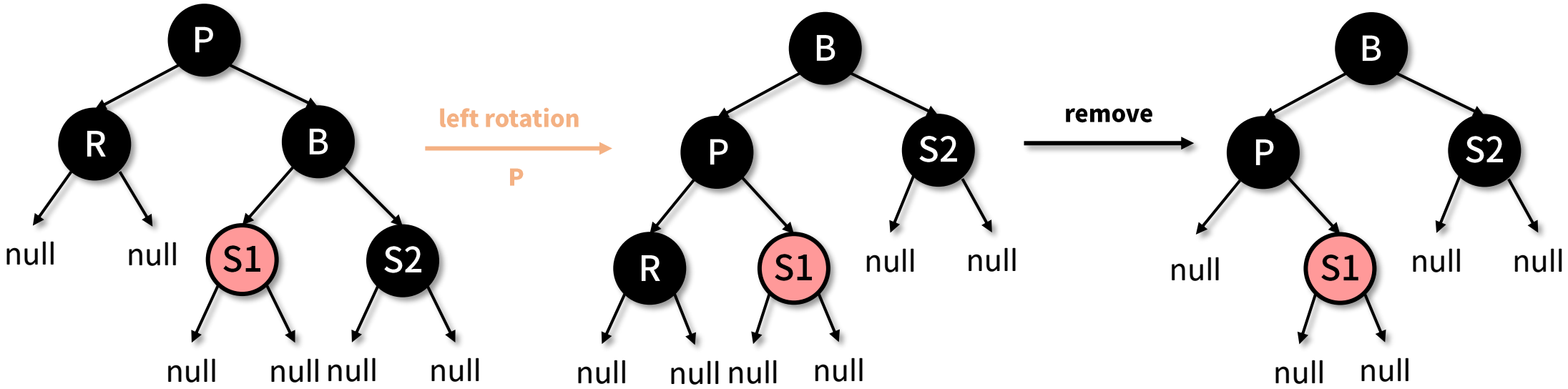


Red-Black Tree

Deletion (20)

case 5:

1. **remove black leaf node**
2. **the brother is a red node**
3. **the brother has two red children**



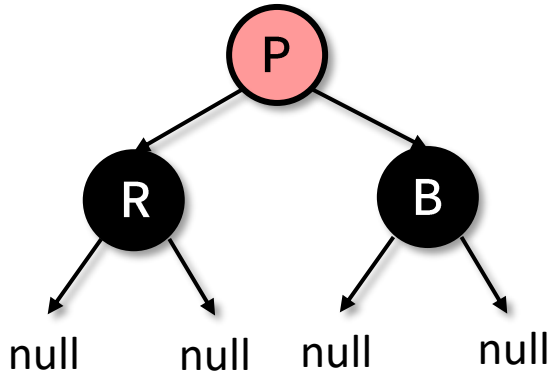
the number of black nodes to the leaf node remains the same

Red-Black Tree

Deletion (21)

case 6:

1. **remove black leaf node**
2. **the parent is red**
3. **the brother is a black node**
4. **the brother has no children**

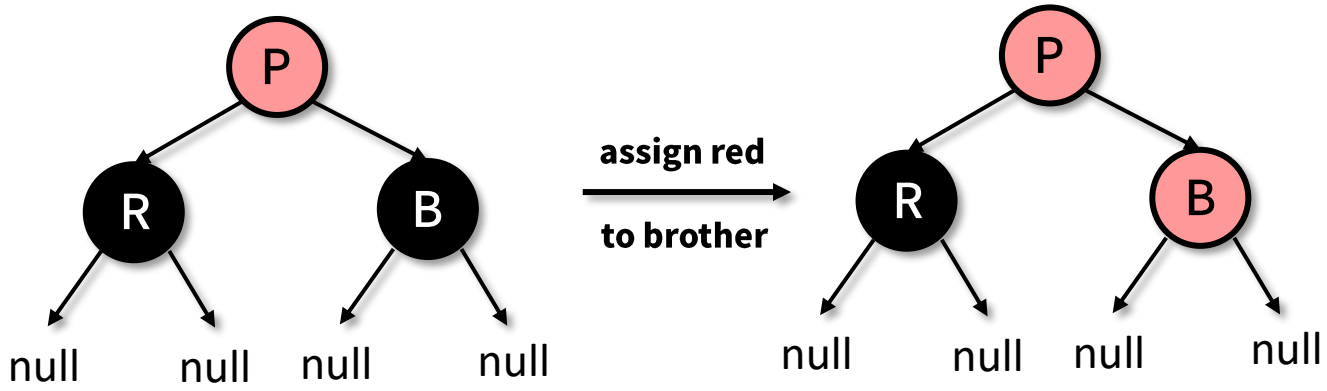


Red-Black Tree

Deletion (22)

case 6:

1. **remove black leaf node**
2. **the parent is red**
3. **the brother is a black node**
4. **the brother has no children**

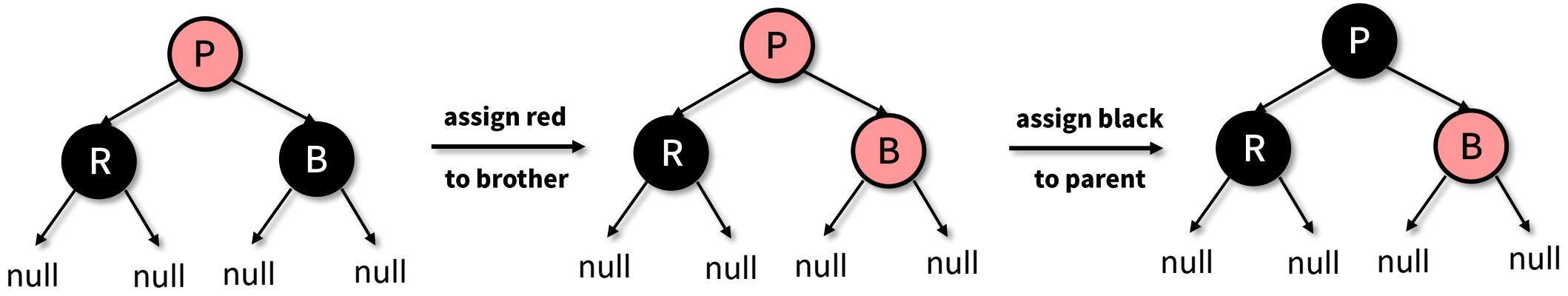


Red-Black Tree

Deletion (23)

case 6:

1. **remove black leaf node**
2. **the parent is red**
3. **the brother is a black node**
4. **the brother has no children**

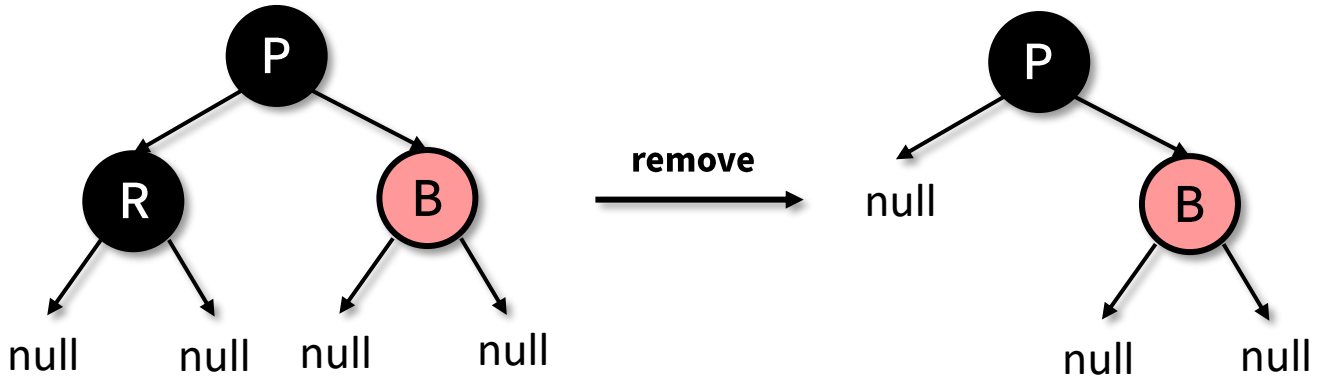


Red-Black Tree

Deletion (24)

case 6:

1. **remove black leaf node**
2. **the parent is red**
3. **the brother is a black node**
4. **the brother has no children**



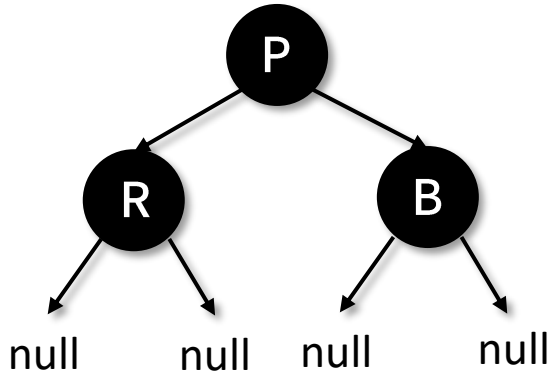
the number of black nodes to the leaf node remains the same

Red-Black Tree

Deletion (25)

case 7:

1. **remove black leaf node**
2. **the parent is black**
3. **the brother is a black node**
4. **the brother has no children**

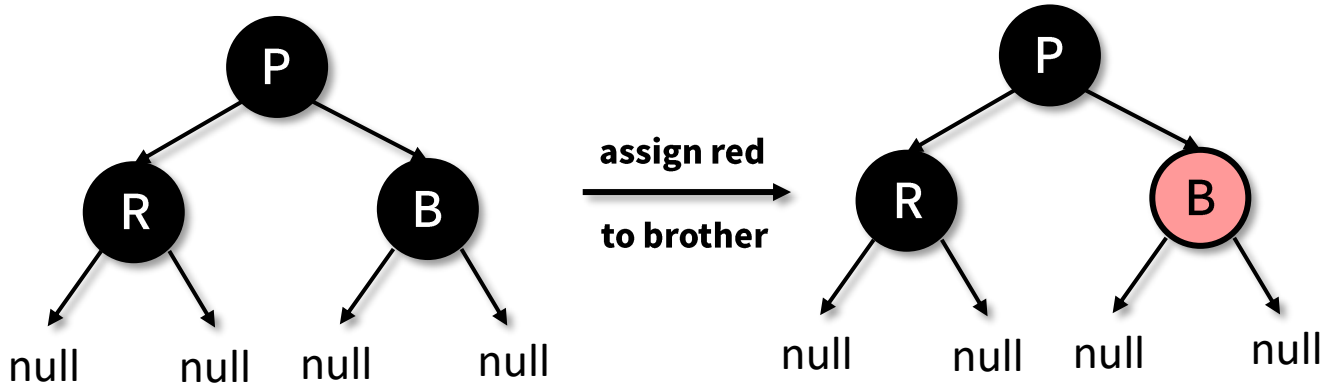


Red-Black Tree

Deletion (26)

case 7:

1. **remove black leaf node**
2. **the parent is black**
3. **the brother is a black node**
4. **the brother has no children**

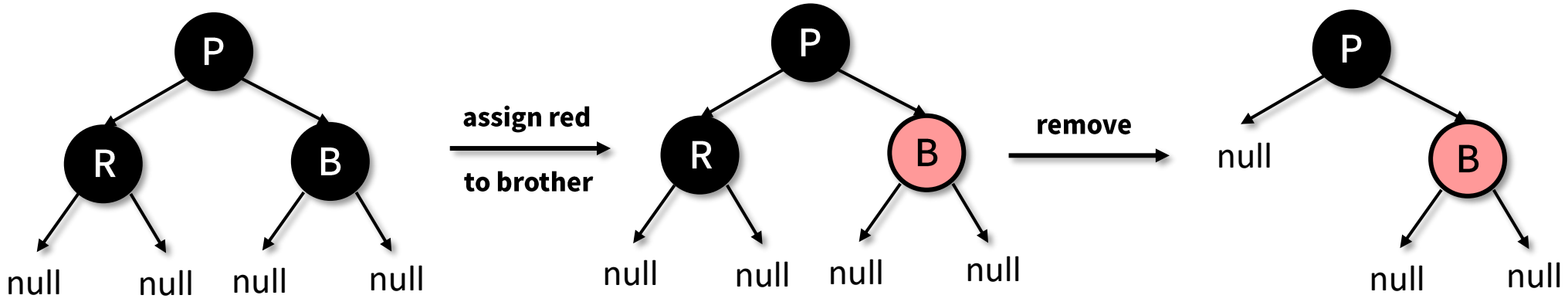


Red-Black Tree

Deletion (27)

case 7:

1. **remove black leaf node**
2. **the parent is black**
3. **the brother is a black node**
4. **the brother has no children**



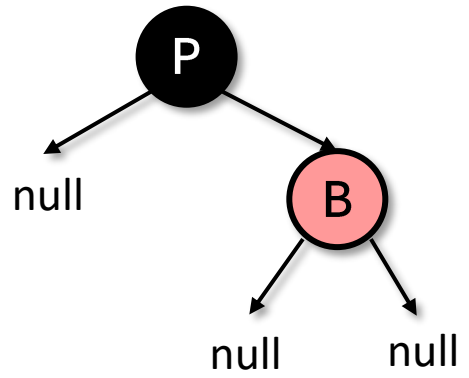
the number of black nodes to the leaf node decrease by one.
need to further correct for it.

Red-Black Tree

Deletion (28)

case 7.1:

1. **parent is root**



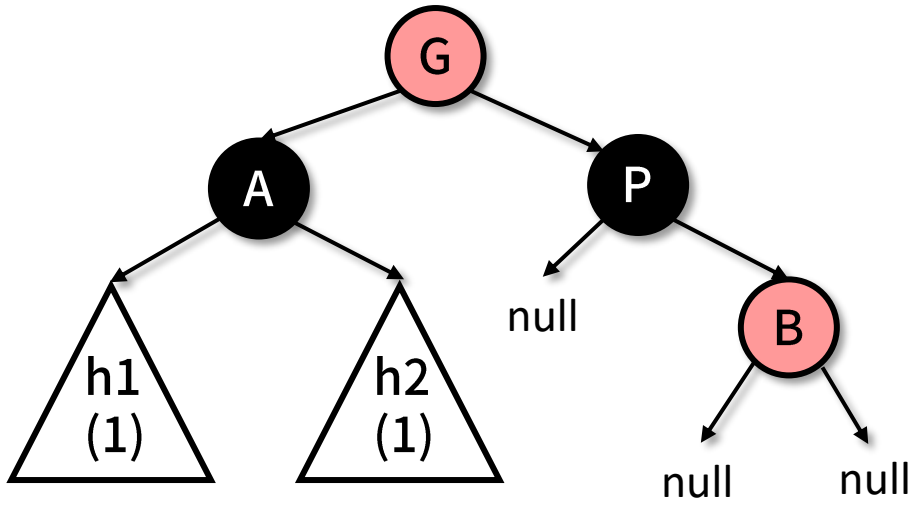
no need for correction

Red-Black Tree

Deletion (29)

case 7.2:

1. **grandparent is red (number of black: 2)**

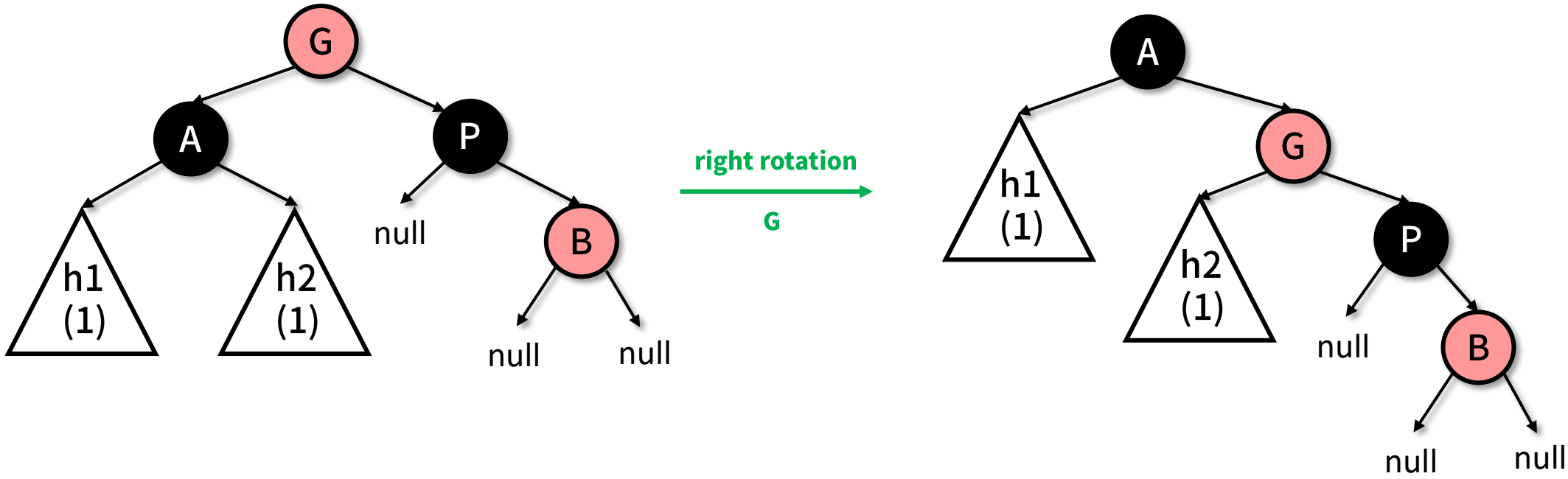


Red-Black Tree

Deletion (30)

case 7.2:

1. grandparent is red (number of black: 2)

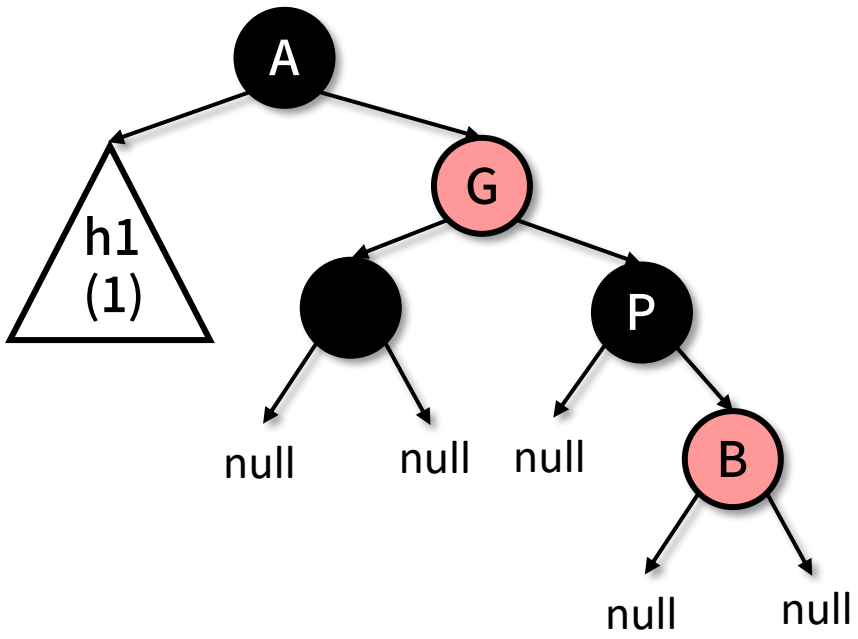


Red-Black Tree

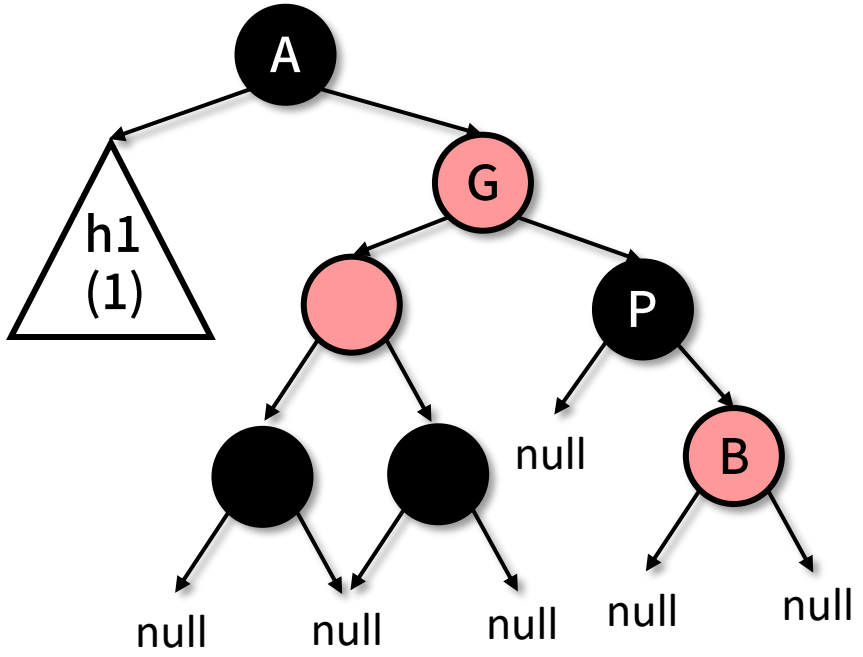
Deletion (31)

case 7.2:

1. grandparent is red (number of black: 2)



no need for correction



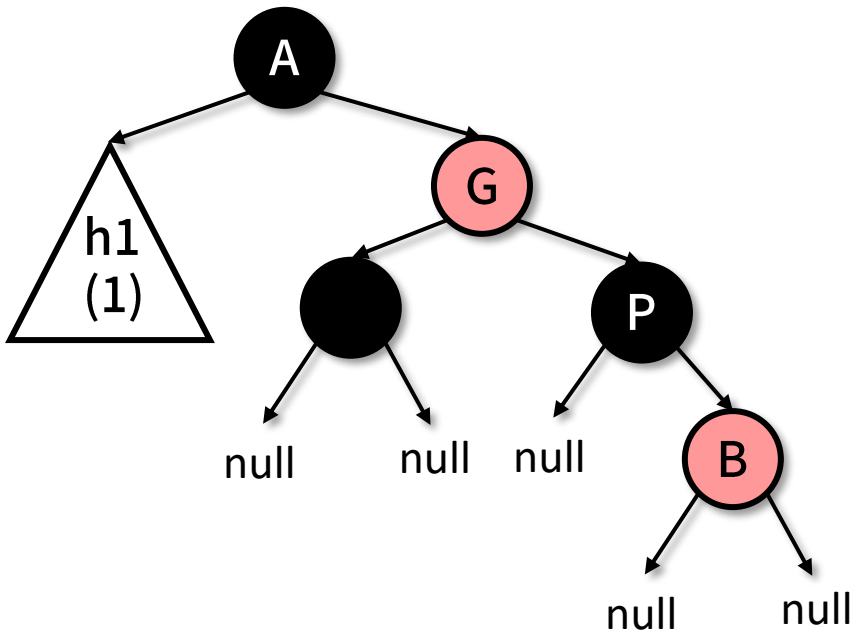
need to swap the color

Red-Black Tree

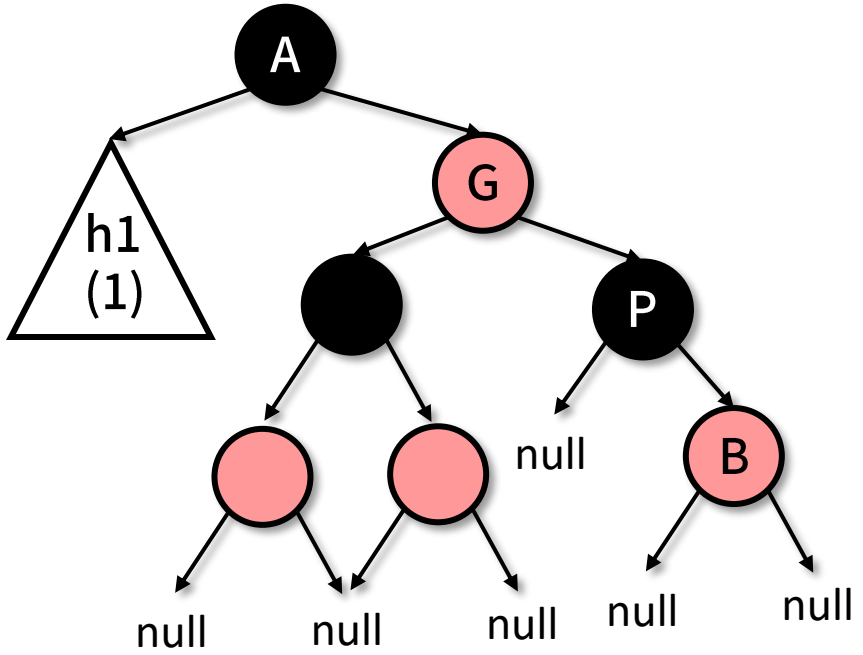
Deletion (32)

case 7.2:

1. grandparent is red (number of black: 2)



no need for correction



need to swap the color

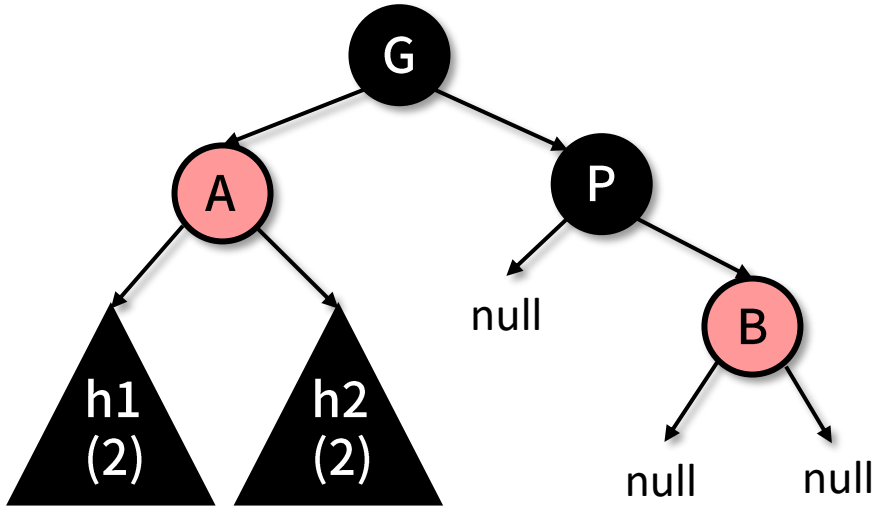
the number of black nodes to the leaf node remains the same

Red-Black Tree

Deletion (33)

case 7.3:

1. **grandparent is black (number of black: 3)**
2. **brother of parent is red**

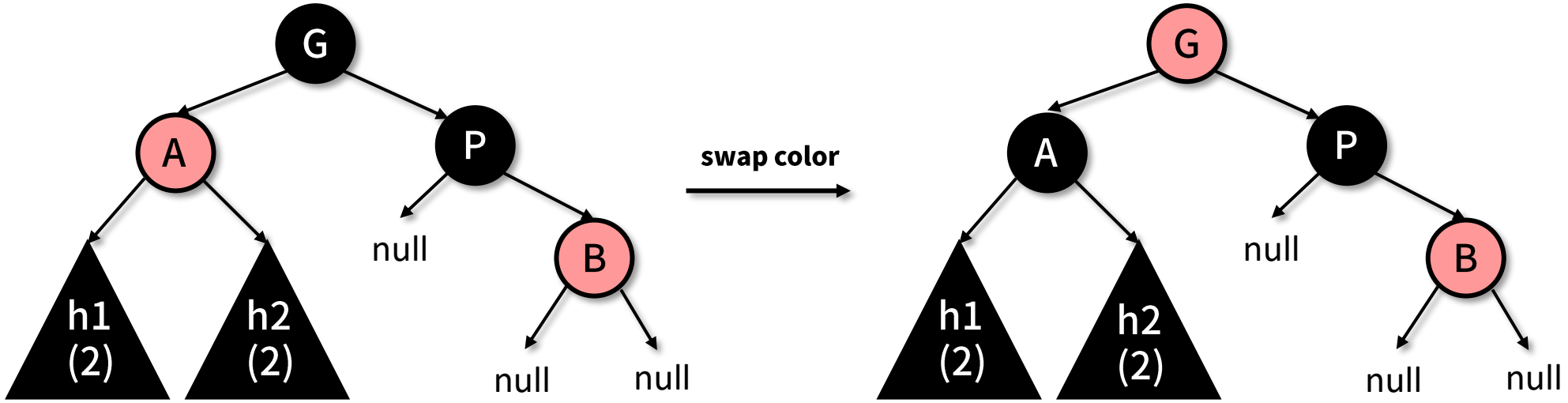


Red-Black Tree

Deletion (34)

case 7.3:

1. **grandparent is black (number of black: 3)**
2. **brother of parent is red**

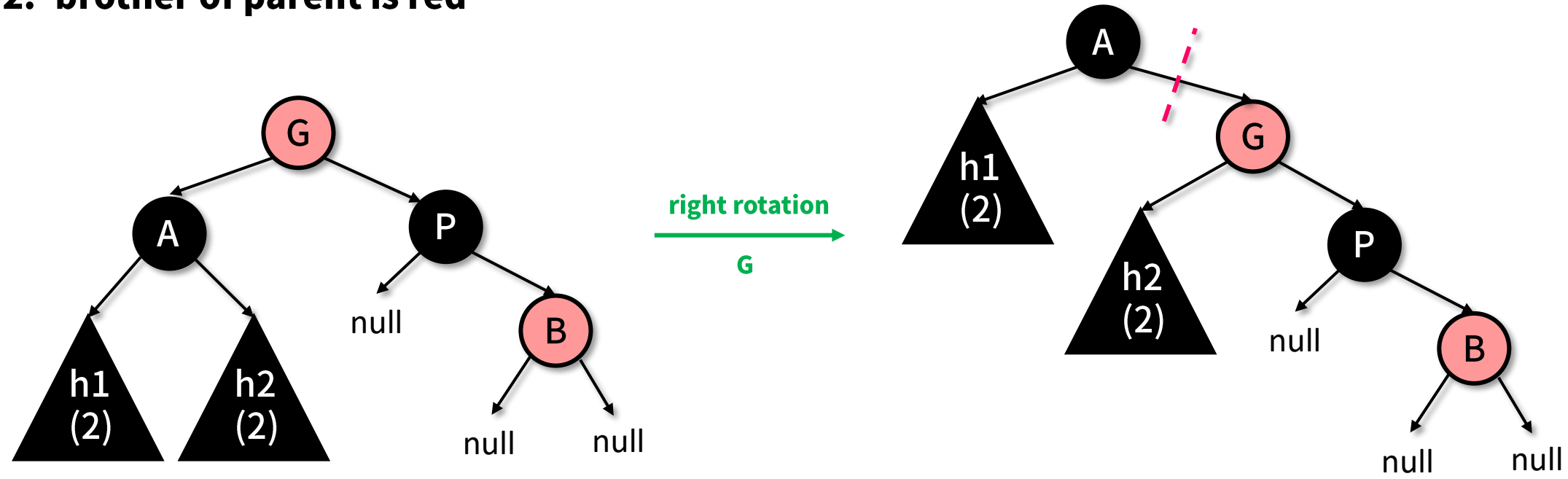


Red-Black Tree

Deletion (35)

case 7.3:

1. **grandparent is black (number of black: 3)**
2. **brother of parent is red**



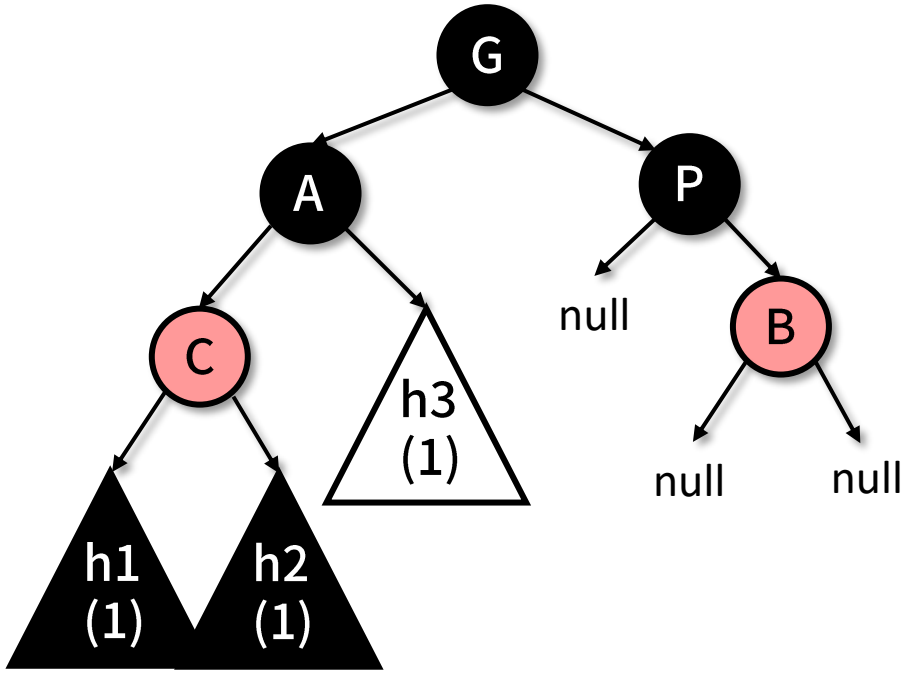
becomes case 7.2

Red-Black Tree

Deletion (36)

case 7.4:

1. **grandparent is black (number of black: 3)**
2. **brother of parent is black**
3. **left child of brother of parent is red.**

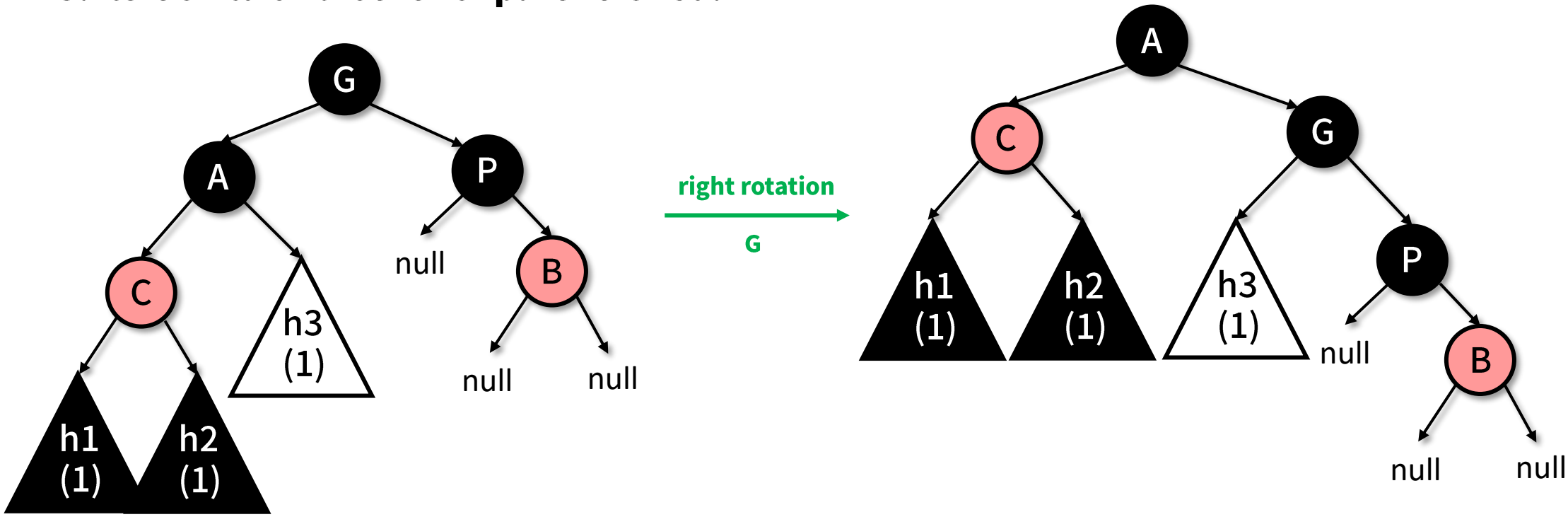


Red-Black Tree

Deletion (37)

case 7.4:

1. **grandparent is black (number of black: 3)**
2. **brother of parent is black**
3. **left child of brother of parent is red.**

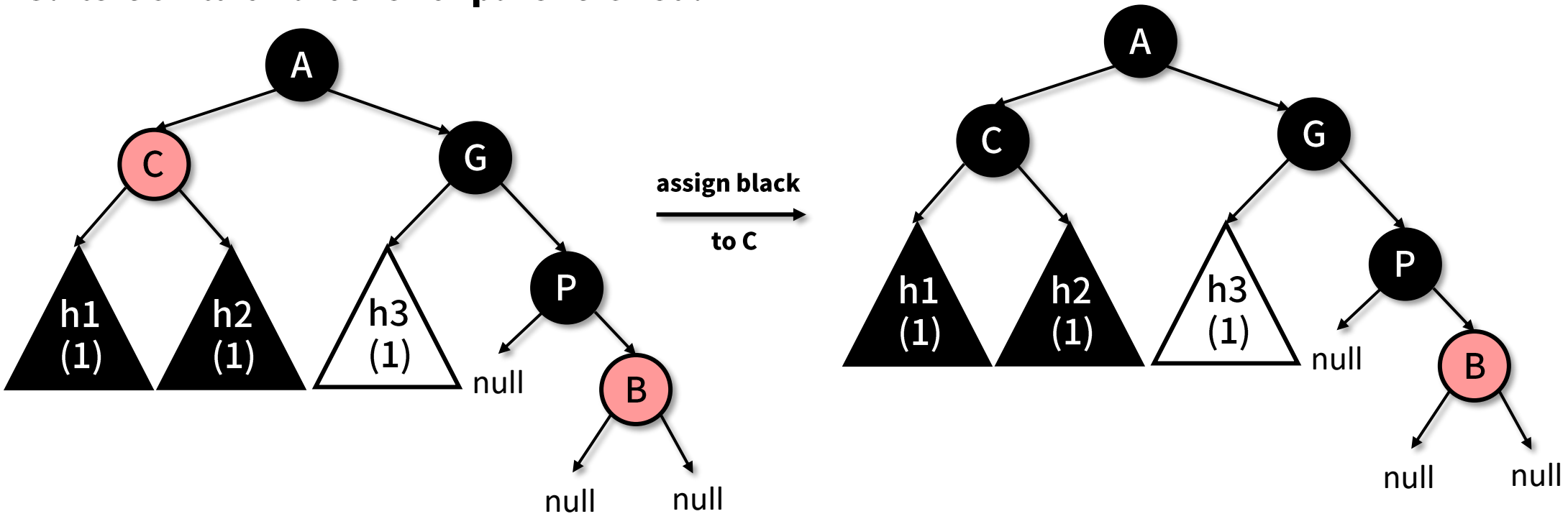


Red-Black Tree

Deletion (38)

case 7.4:

1. **grandparent is black (number of black: 3)**
2. **brother of parent is black**
3. **left child of brother of parent is red.**



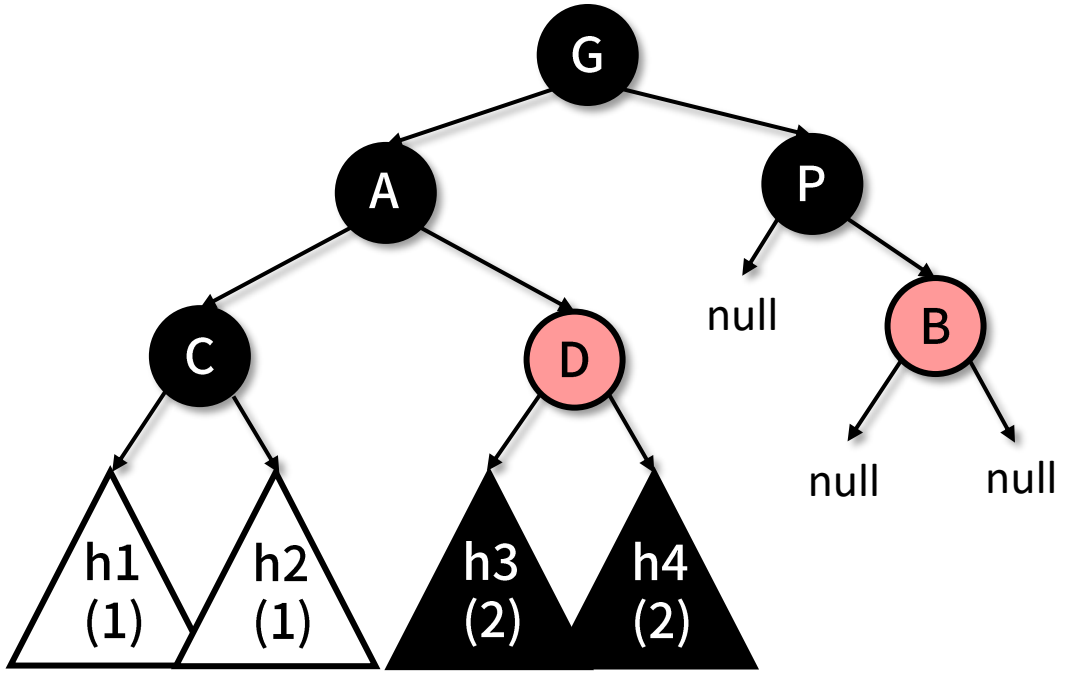
the number of black nodes to the leaf node remains the same

Red-Black Tree

Deletion (39)

case 7.5:

1. **grandparent is black (number of black: 3)**
2. **brother of parent is black**
3. **left child of brother of parent is black.**
4. **right child of brother of parent is red**

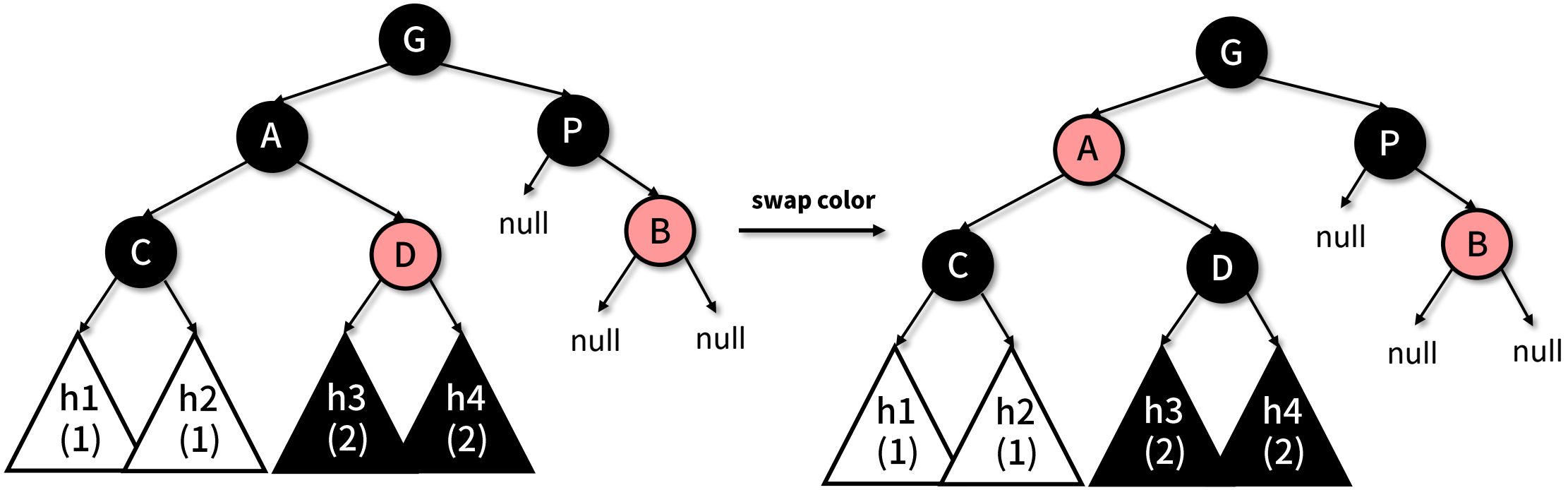


Red-Black Tree

Deletion (40)

case 7.5:

1. **grandparent is black (number of black: 3)**
2. **brother of parent is black**
3. **left child of brother of parent is black.**
4. **right child of brother of parent is red**

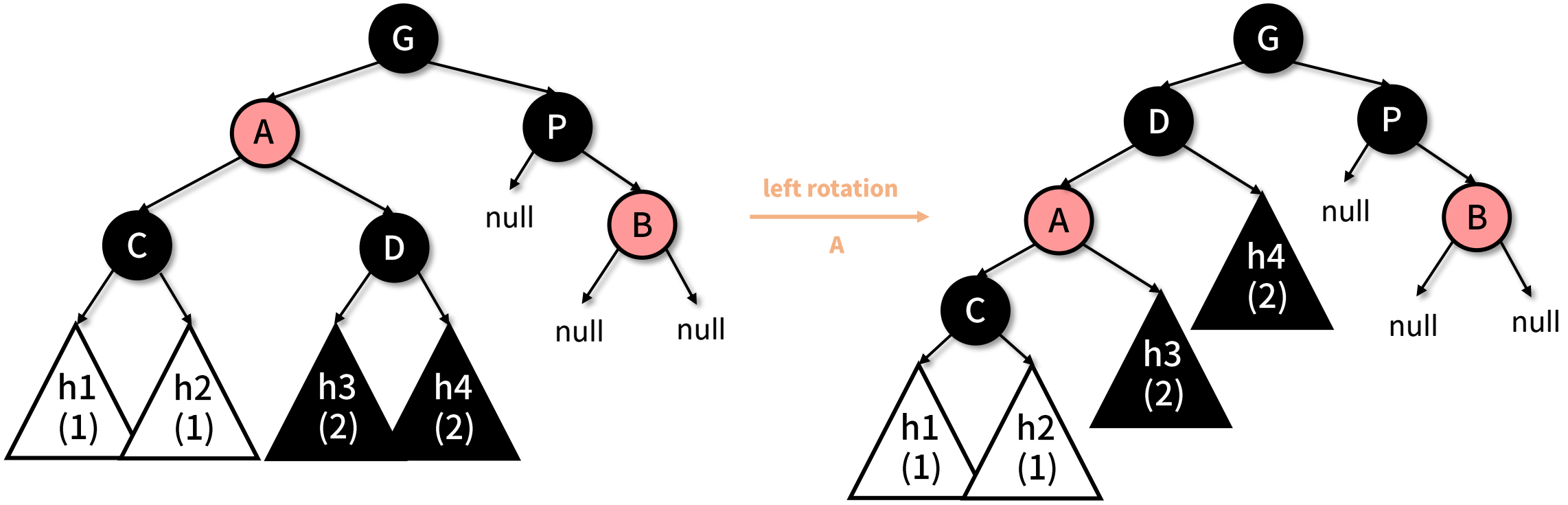


Red-Black Tree

Deletion (41)

case 7.5:

1. **grandparent is black (number of black: 3)**
2. **brother of parent is black**
3. **left child of brother of parent is black.**
4. **right child of brother of parent is red**

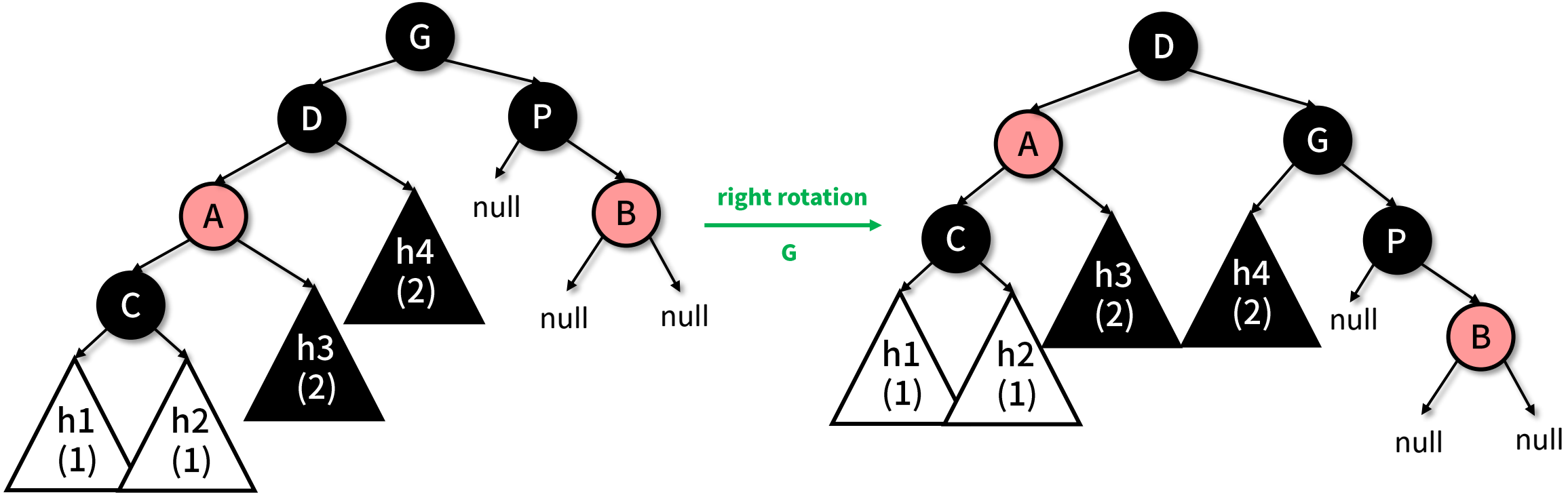


Red-Black Tree

Deletion (42)

case 7.5:

1. **grandparent is black (number of black: 3)**
2. **brother of parent is black**
3. **left child of brother of parent is black.**
4. **right child of brother of parent is red**

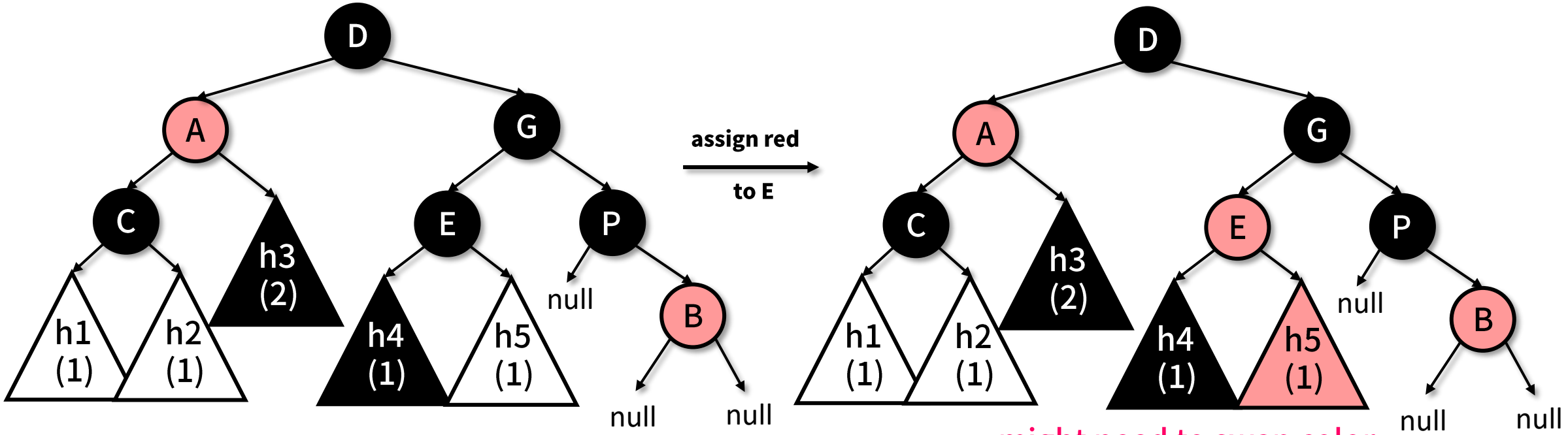


Red-Black Tree

Deletion (43)

case 7.5:

1. **grandparent is black (number of black: 3)**
2. **brother of parent is black**
3. **left child of brother of parent is black.**
4. **right child of brother of parent is red**



might need to swap color

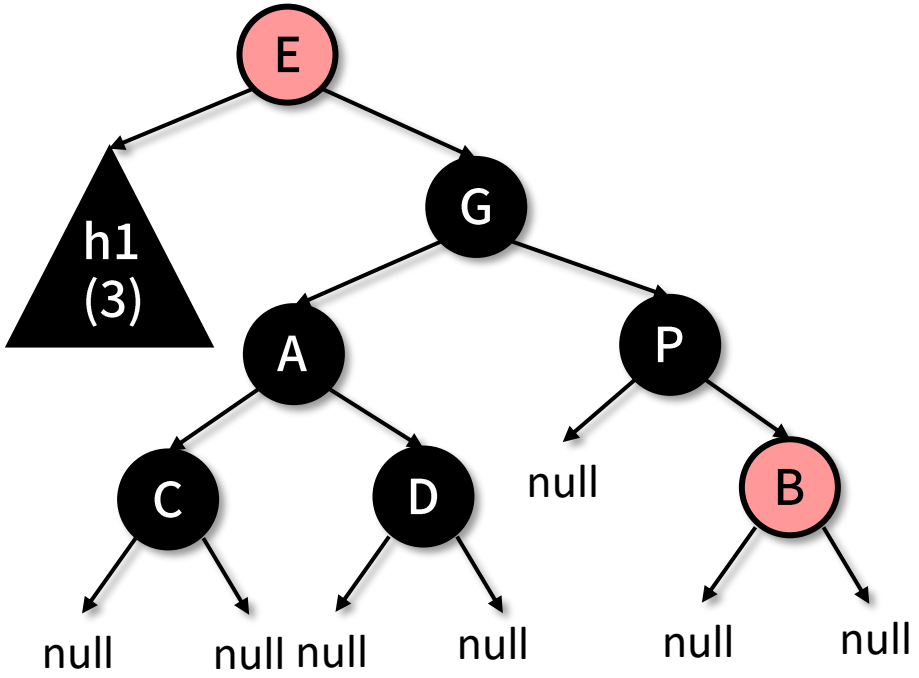
the number of black nodes to the leaf node remains the same

Red-Black Tree

Deletion (44)

case 7.6:

1. **grandparent is black (number of black: 3)**
2. **brother of parent is black**
3. **children of brother of parent is black**
4. **grand-grand parent is red**

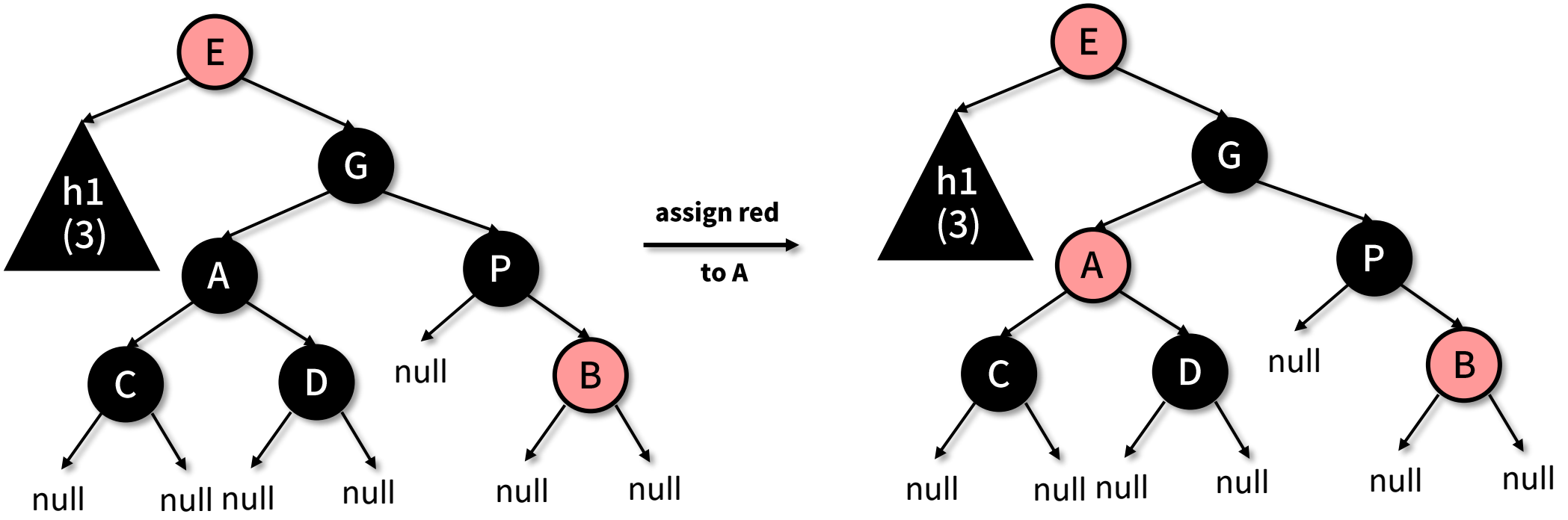


Red-Black Tree

Deletion (45)

case 7.6:

1. **grandparent is black (number of black: 3)**
2. **brother of parent is black**
3. **children of brother of parent is black**
4. **grand-grand parent is red**



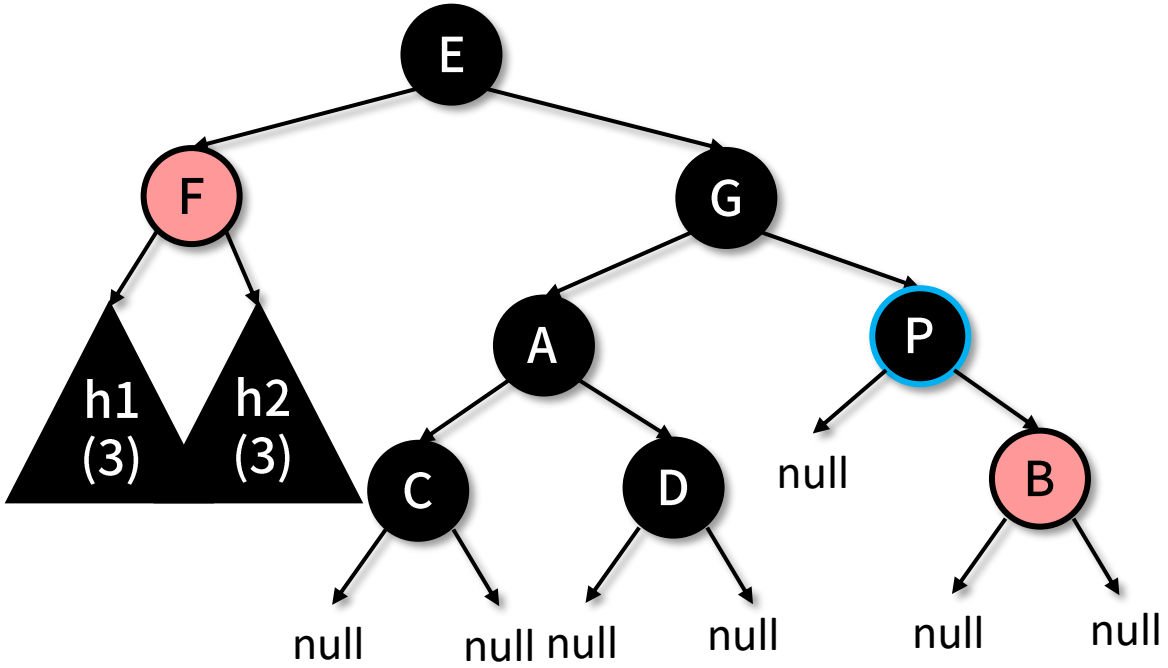
becomes case 7.2

Red-Black Tree

Deletion (46)

case 7.6:

1. **grandparent is black (number of black: 4)**
2. **brother of parent is black**
3. **children of brother of parent is black**
4. **grand-grand parent is black**
5. **left child of grand-grand parent is red**

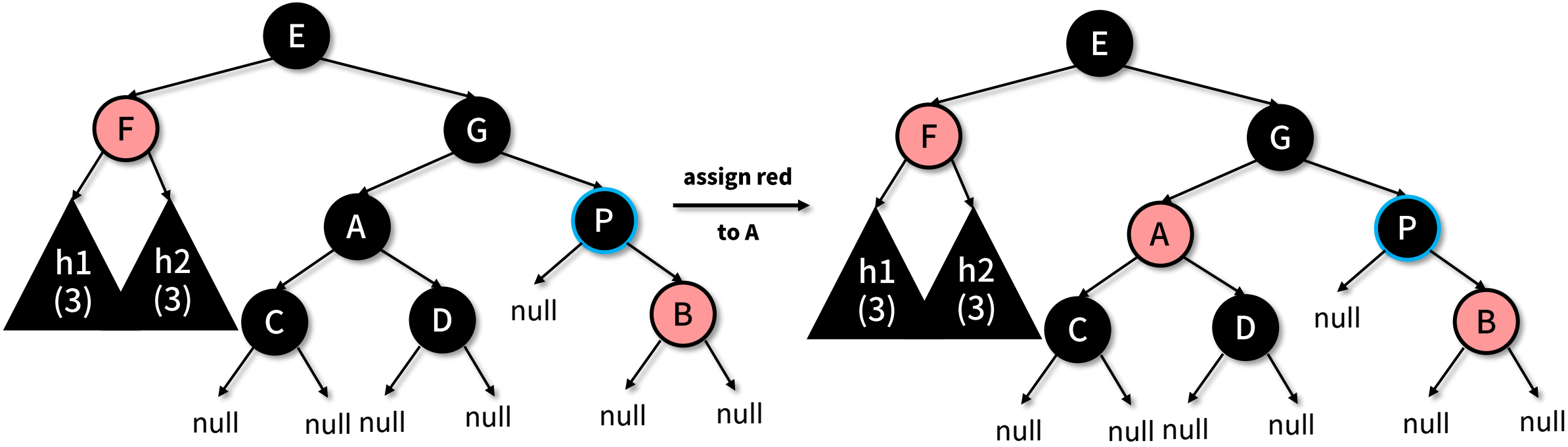


Red-Black Tree

Deletion (47)

case 7.6:

1. **grandparent is black (number of black: 4)**
2. **brother of parent is black**
3. **children of brother of parent is black**
4. **grand-grand parent is black**
5. **left child of grand-grand parent is red**



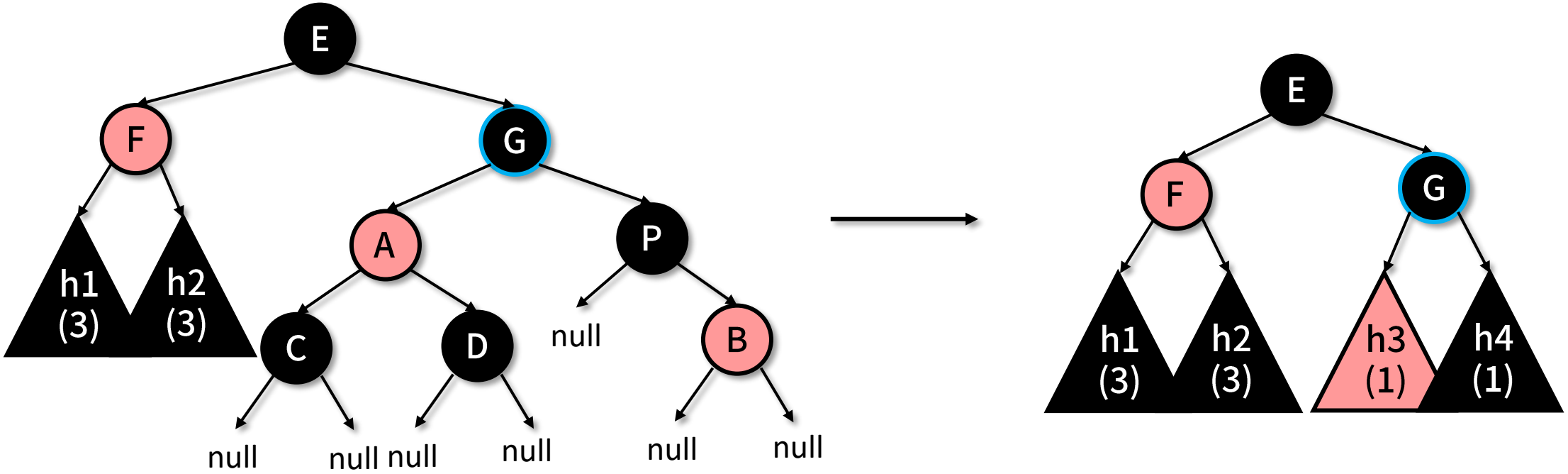
iterate to G

Red-Black Tree

Deletion (48)

case 7.6:

1. **grandparent is black (number of black: 4)**
2. **brother of parent is black**
3. **children of brother of parent is black**
4. **grand-grand parent is black**
5. **left child of grand-grand parent is red**

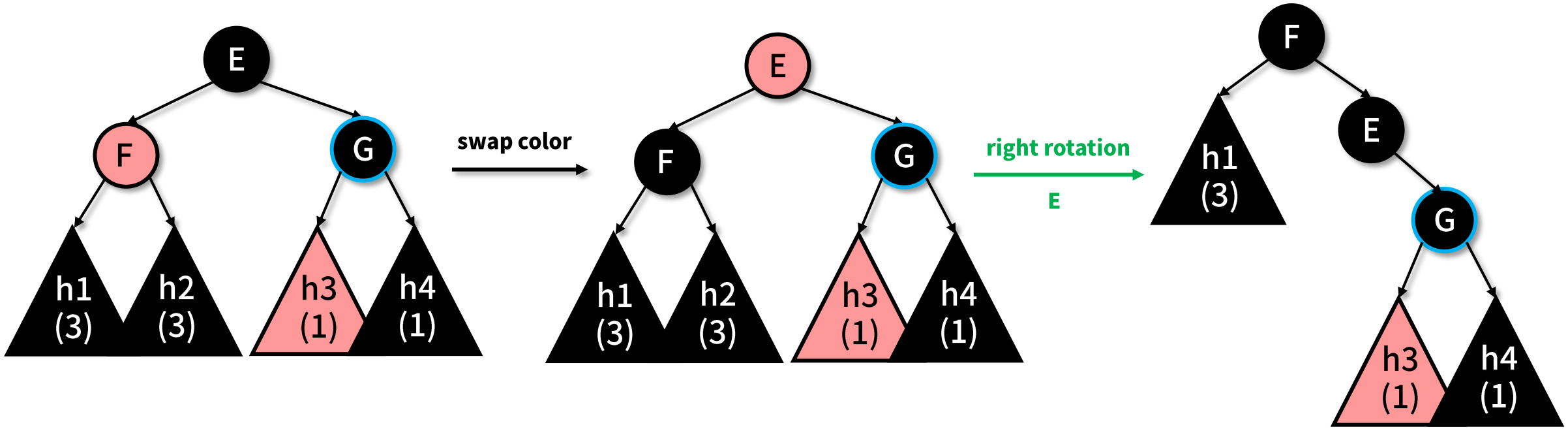


Red-Black Tree

Deletion (49)

case 7.6:

1. **grandparent is black (number of black: 4)**
2. **brother of parent is black**
3. **children of brother of parent is black**
4. **grand-grand parent is black**
5. **left child of grand-grand parent is red**



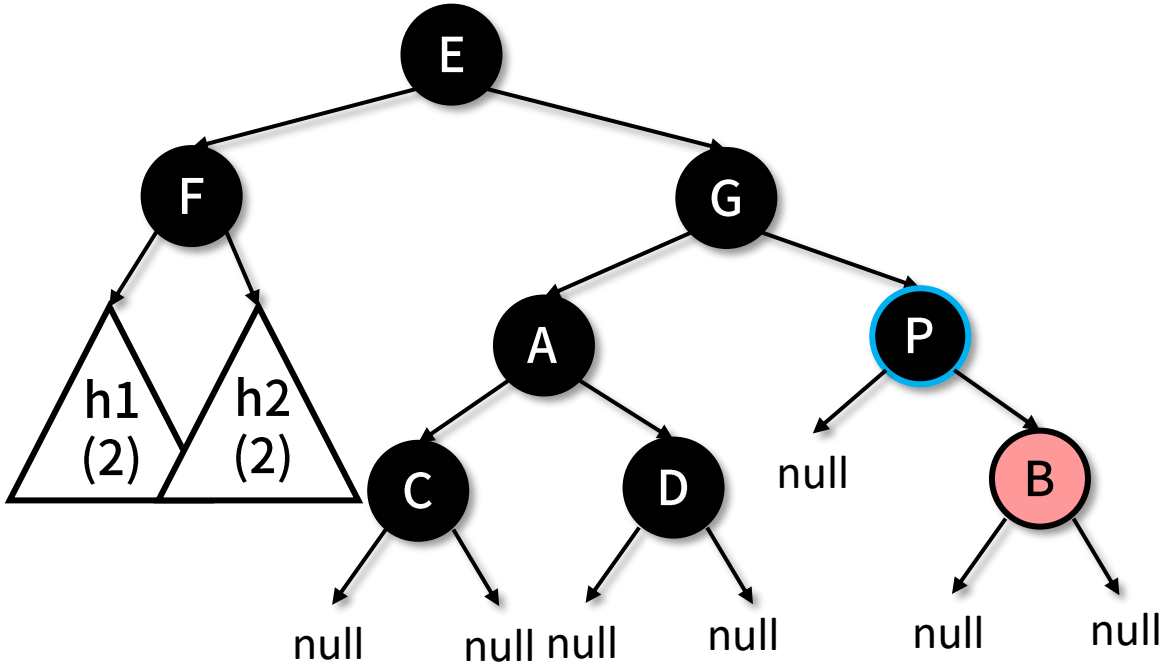
the number of black nodes to the leaf node remains the same

Red-Black Tree

Deletion (50)

case 7.7:

1. **grandparent is black (number of black: 4)**
2. **brother of parent is black**
3. **children of brother of parent is black**
4. **grand-grand parent is black**
5. **left child of grand-grand parent is black**

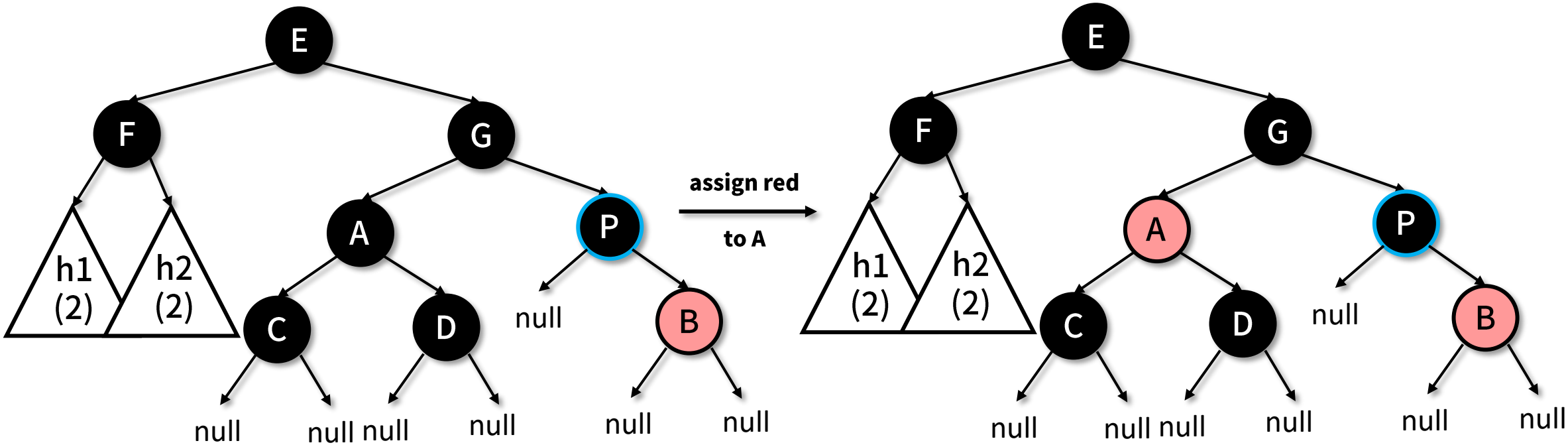


Red-Black Tree

Deletion (51)

case 7.7:

1. **grandparent is black (number of black: 4)**
2. **brother of parent is black**
3. **children of brother of parent is black**
4. **grand-grand parent is black**
5. **left child of grand-grand parent is black**



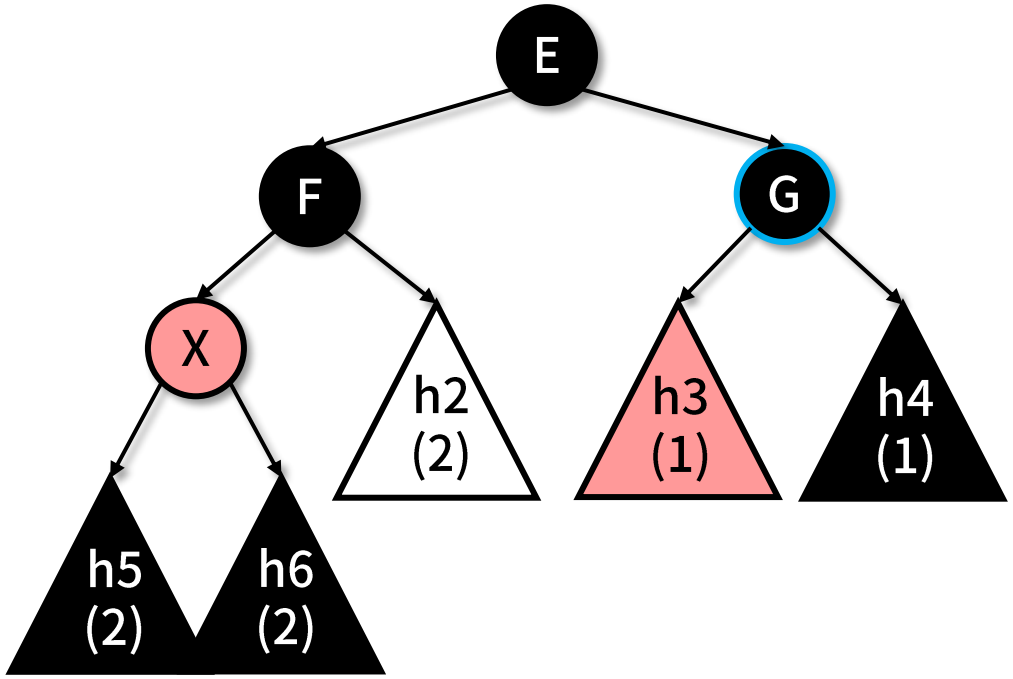
iterate to G

Red-Black Tree

Deletion (52)

case 7.7.1:

1. **grandparent is black (number of black: 4)**
2. **brother of parent is black**
3. **children of brother of parent is black**
4. **grand-grand parent is black**
5. **left child of grand-grand parent is black**
6. **left child of brother of grand-parent is red**

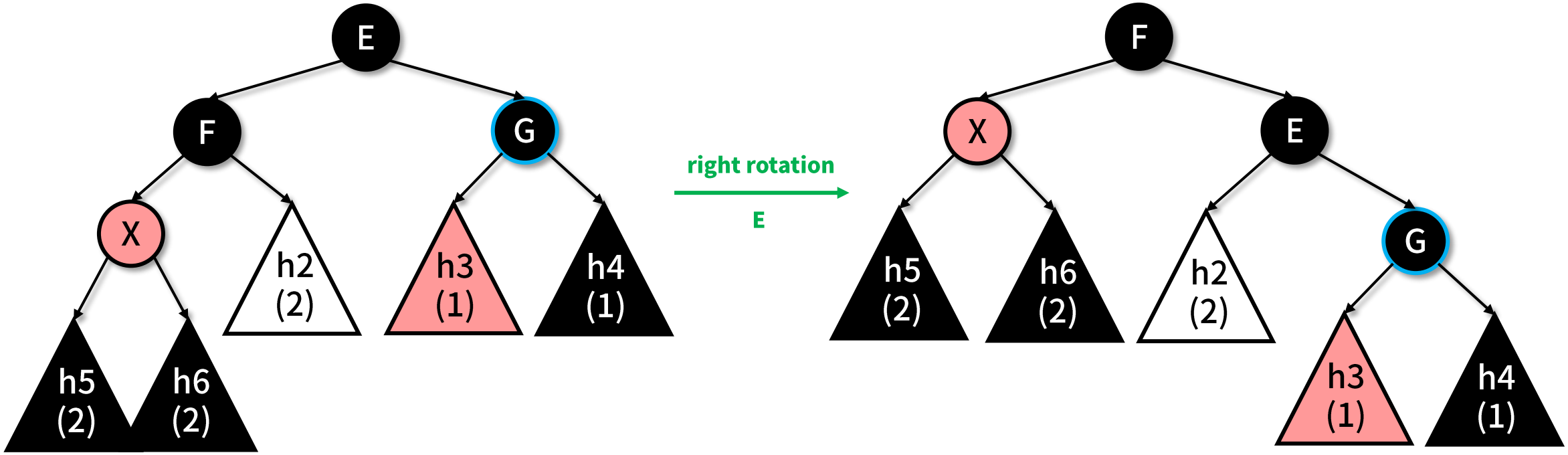


Red-Black Tree

Deletion (53)

case 7.7.1:

1. **grandparent is black (number of black: 4)**
2. **brother of parent is black**
3. **children of brother of parent is black**
4. **grand-grand parent is black**
5. **left child of grand-grand parent is black**
6. **left child of brother of grand-parent is red**

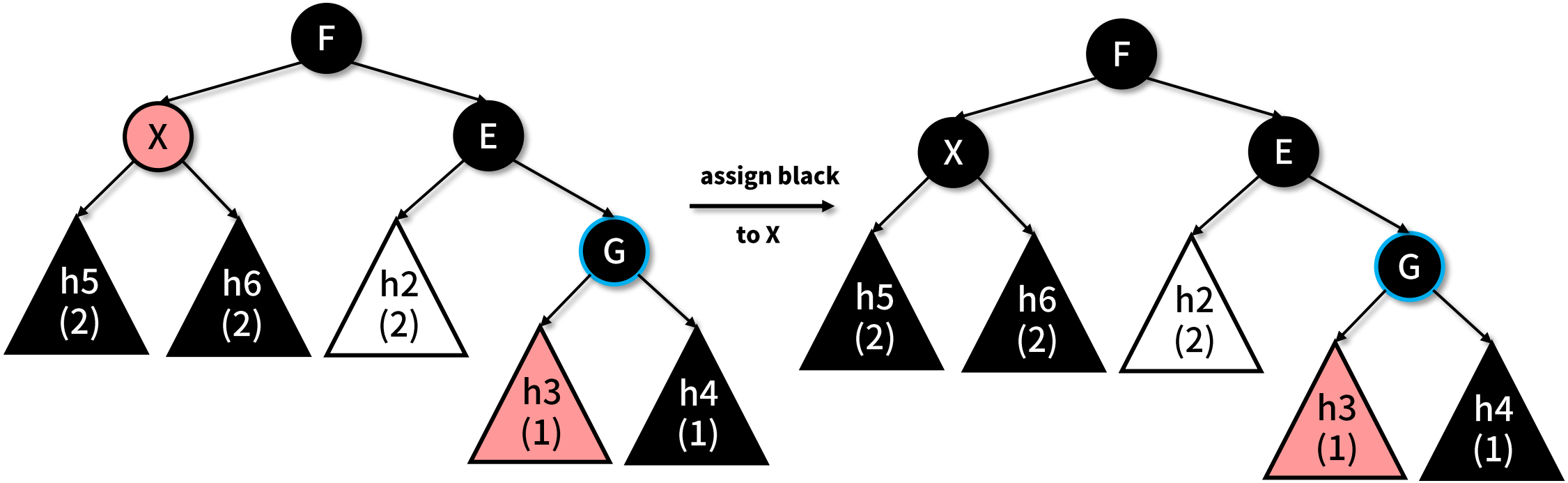


Red-Black Tree

Deletion (54)

case 7.7.1:

1. **grandparent is black (number of black: 4)**
2. **brother of parent is black**
3. **children of brother of parent is black**
4. **grand-grand parent is black**
5. **left child of grand-grand parent is black**
6. **left child of brother of grand-parent is red**



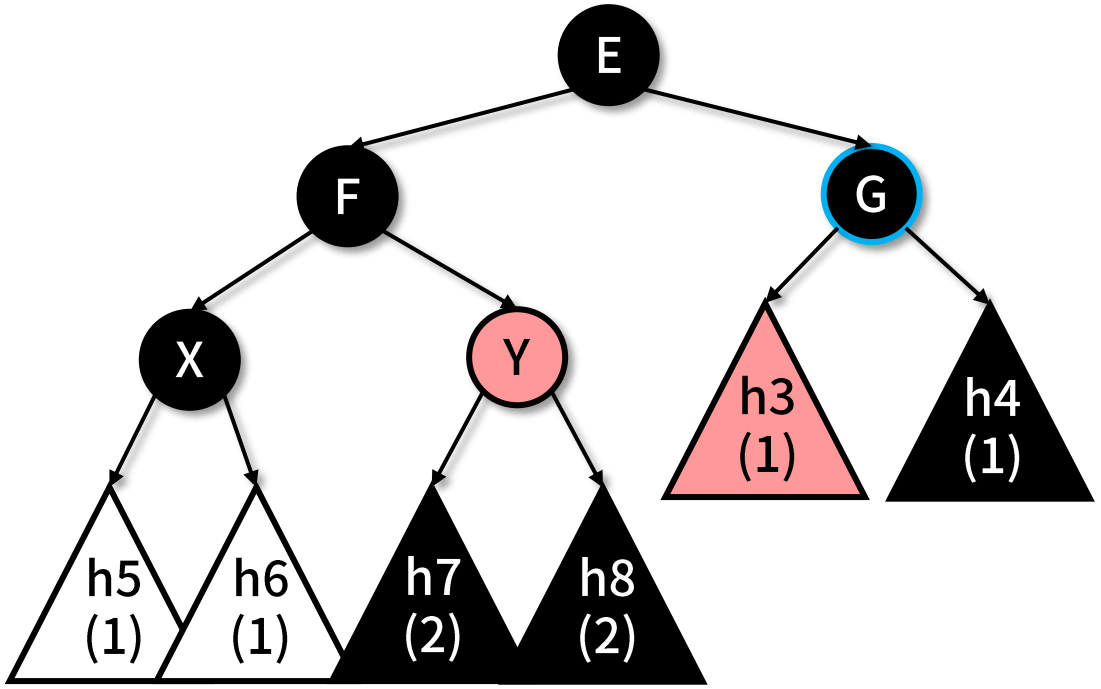
the number of black nodes to the leaf node remains the same

Red-Black Tree

Deletion (55)

case 7.7.2:

1. **grandparent is black (number of black: 4)**
2. **brother of parent is black**
3. **children of brother of parent is black**
4. **grand-grand parent is black**
5. **left child of grand-grand parent is black**
6. **left child of brother of grand-parent is black**
7. **right child of brother of grand-parent is red**

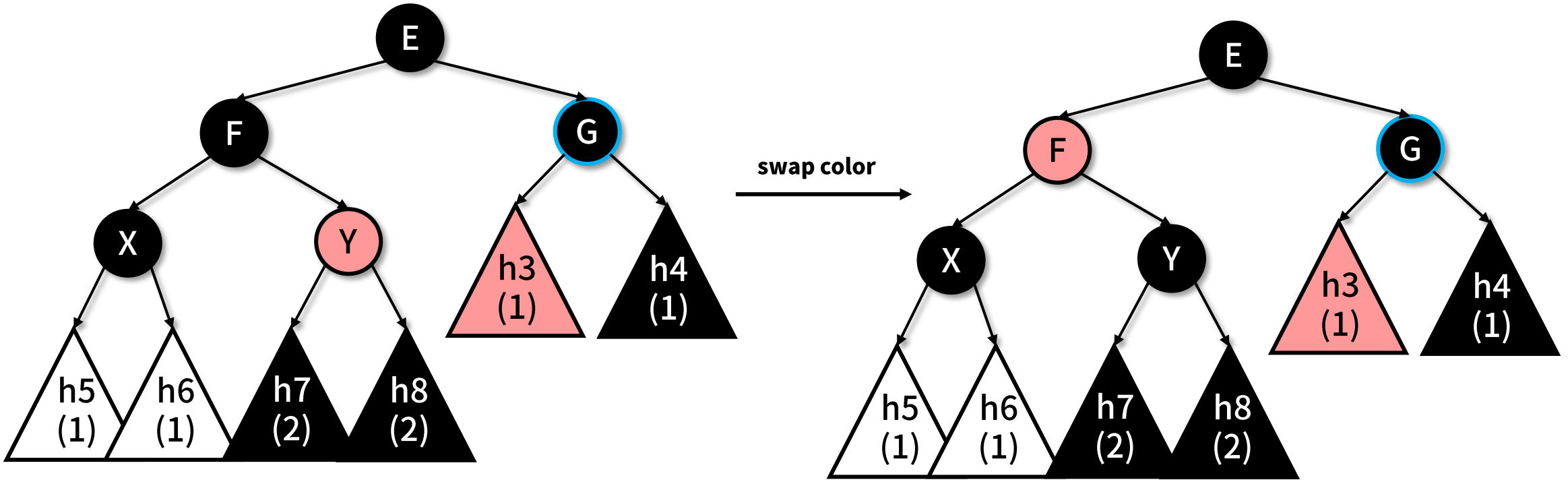


Red-Black Tree

Deletion (56)

case 7.7.2:

1. **grandparent is black (number of black: 4)**
2. **brother of parent is black**
3. **children of brother of parent is black**
4. **grand-grand parent is black**
5. **left child of grand-grand parent is black**
6. **left child of brother of grand-parent is black**
7. **right child of brother of grand-parent is red**

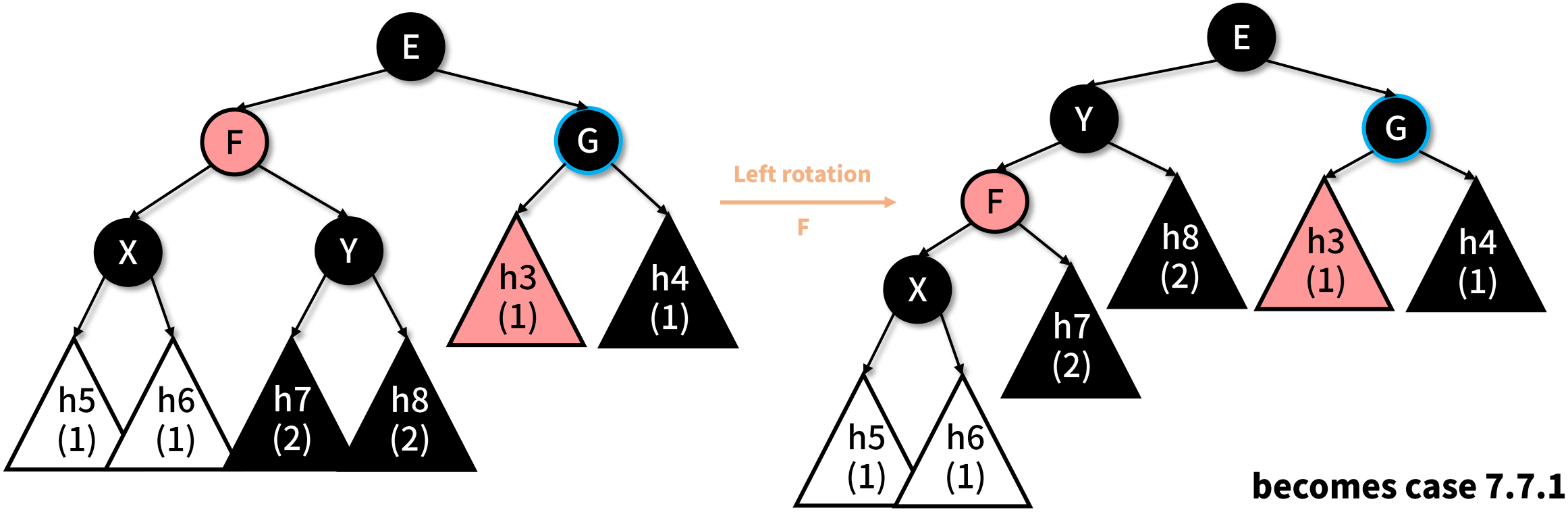


Red-Black Tree

Deletion (57)

case 7.7.2:

1. **grandparent is black (number of black: 4)**
2. **brother of parent is black**
3. **children of brother of parent is black**
4. **grand-grand parent is black**
5. **left child of grand-grand parent is black**
6. **left child of brother of grand-parent is black**
7. **right child of brother of grand-parent is red**



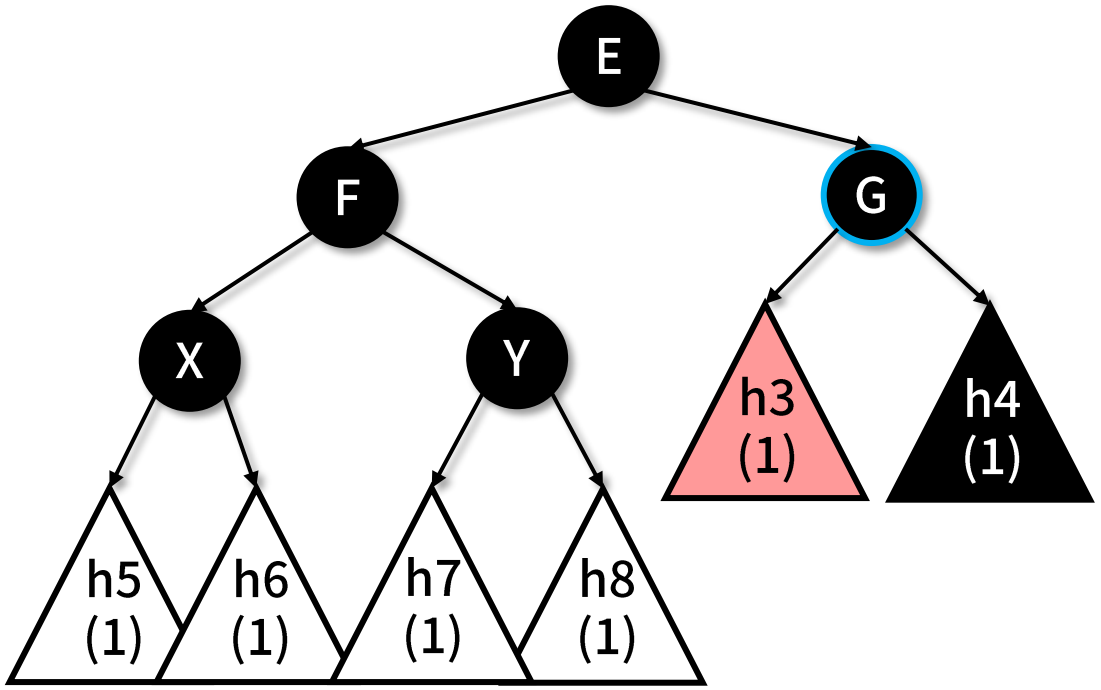
becomes case 7.7.1

Red-Black Tree

Deletion (58)

case 7.7.3:

1. **grandparent is black (number of black: 4)**
2. **brother of parent is black**
3. **children of brother of parent is black**
4. **grand-grand parent is black**
5. **left child of grand-grand parent is black**
6. **children of brother of grand-parent are black**

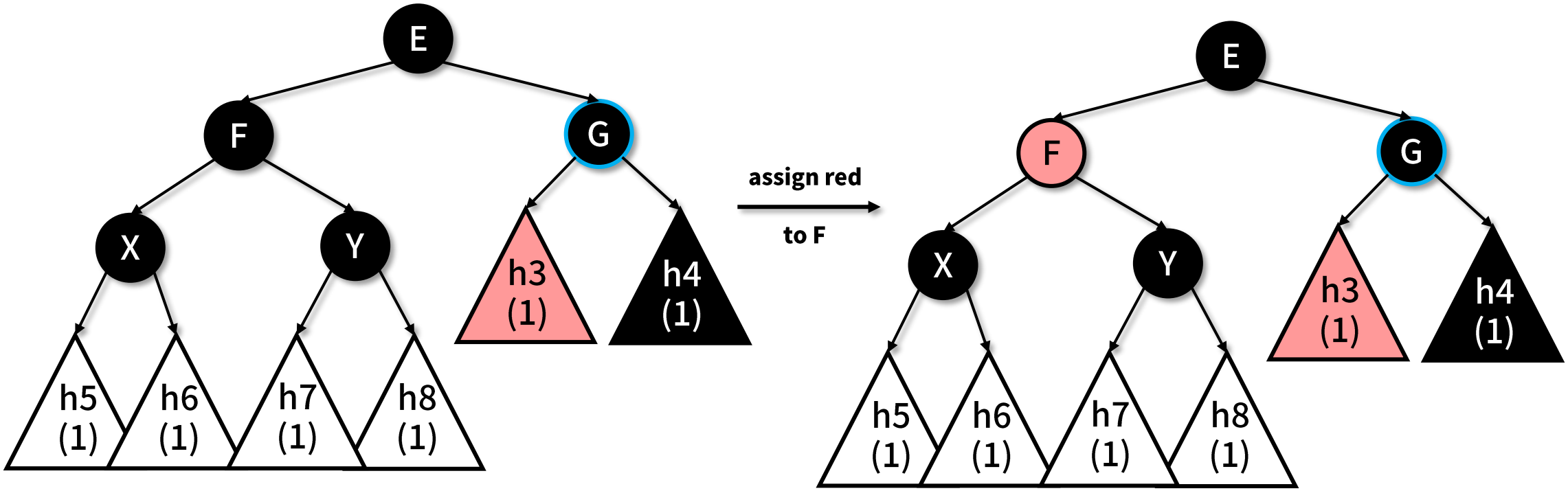


Red-Black Tree

Deletion (59)

case 7.7.3:

1. **grandparent is black (number of black: 4)**
2. **brother of parent is black**
3. **children of brother of parent is black**
4. **grand-grand parent is black**
5. **left child of grand-grand parent is black**
6. **children of brother of grand-parent are black**

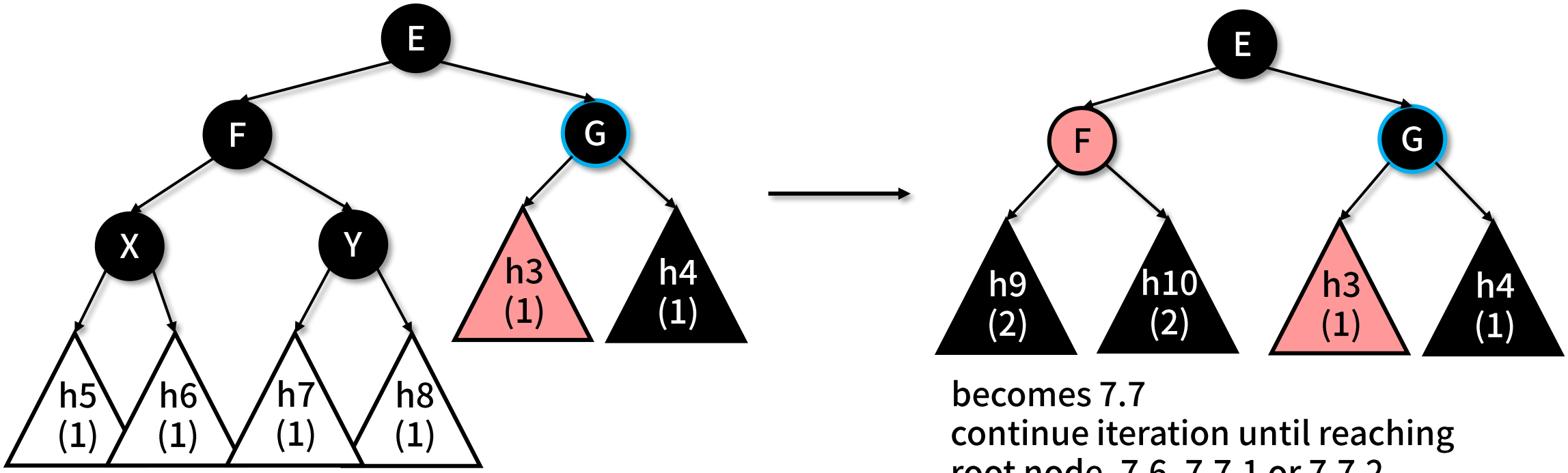


Red-Black Tree

Deletion (60)

case 7.7.3:

1. **grandparent is black (number of black: 4)**
2. **brother of parent is black**
3. **children of brother of parent is black**
4. **grand-grand parent is black**
5. **left child of grand-grand parent is black**
6. **children of brother of grand-parent are black**



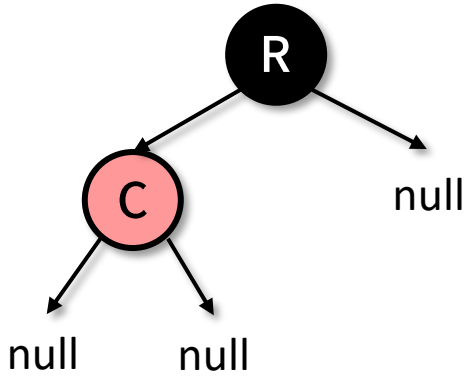
becomes 7.7
continue iteration until reaching
root node, 7.6, 7.7.1 or 7.7.2.

Red-Black Tree

Deletion (61)

case 8:

1. **remove black node with one red child leaf node**

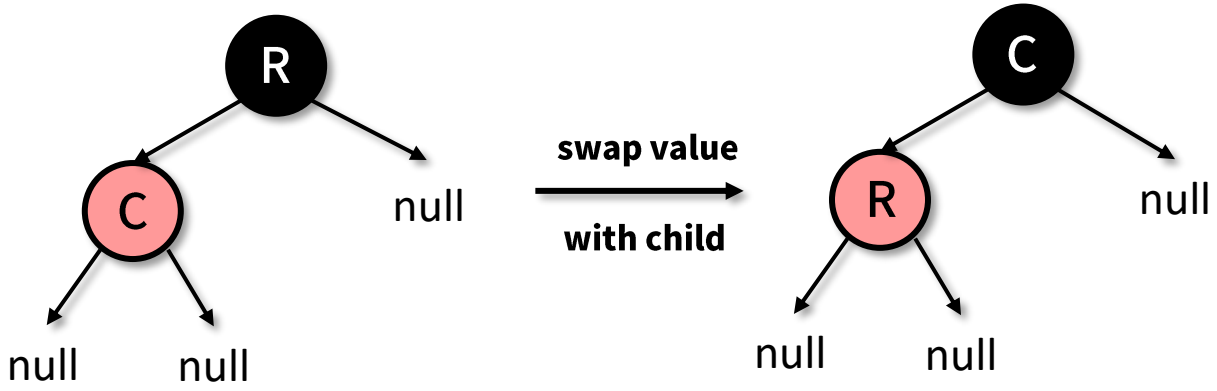


Red-Black Tree

Deletion (62)

case 8:

1. **remove black node with one red child leaf node**

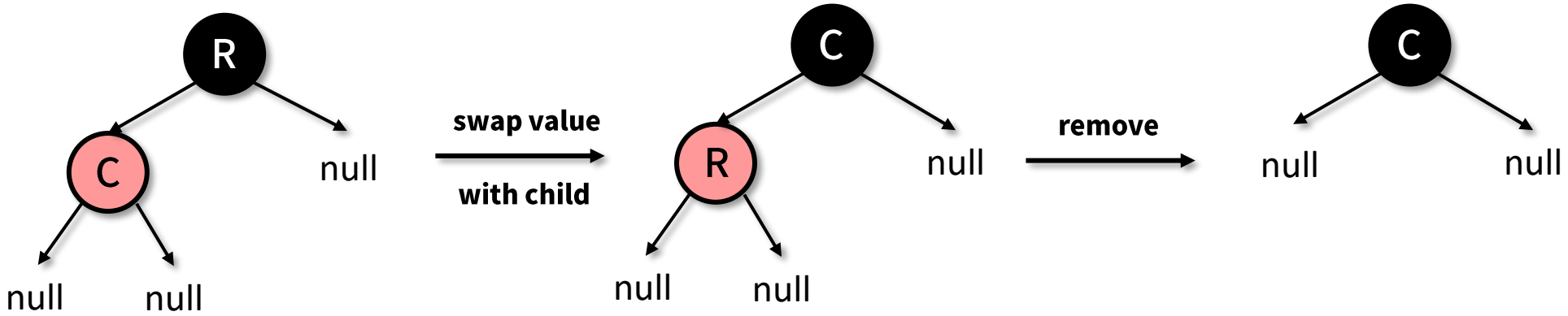


Red-Black Tree

Deletion (63)

case 8:

1. **remove black node with one red child leaf node**



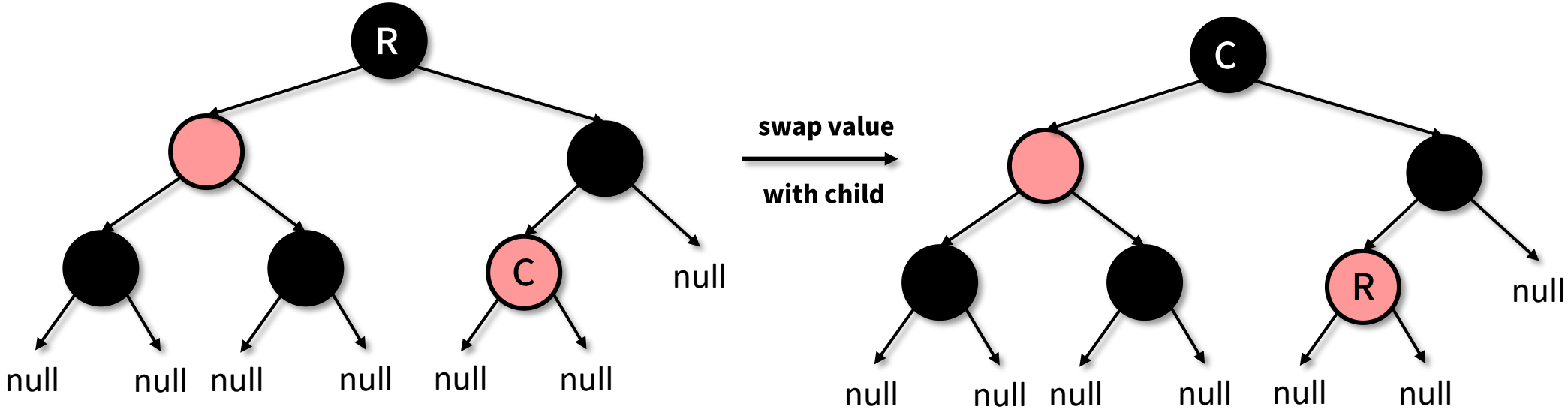
the number of black nodes to the leaf node remains the same

Red-Black Tree

Deletion (64)

case 9:

1. **remove black node with two children**
2. **the minimum node in right subtree is red**



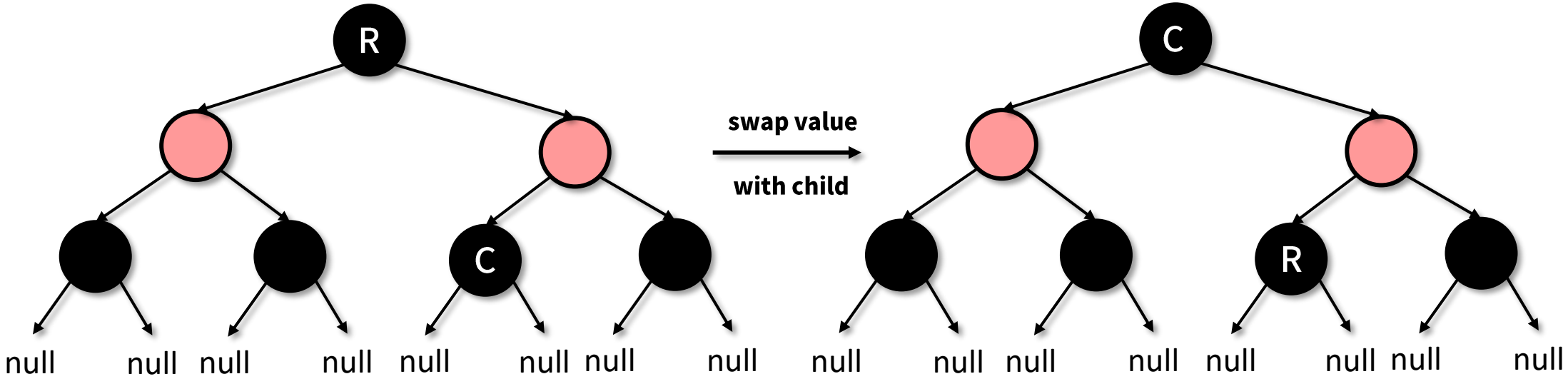
becomes case 1

Red-Black Tree

Deletion (65)

case 9:

1. **remove black node with two children**
2. **the minimum node in right subtree is black**



check case 2-7

Binary Search Tree

Summary

Worst Time Complexity

	Binary Search Tree	AVL Tree	Red-Black Tree
Search	$O(N)$	$O(\log N)$	$O(\log N)$ *
Insert	$O(N)$	$O(\log N)$ (2 rotations)	$O(\log N)$ (2 rotations)
Delete	$O(N)$	$O(\log N)$ (multiple)	$O(\log N)$ (3 rotations)

* 2 times longer than AVL Tree

Binary Search Tree

Summary

Average Time Complexity

	Binary Search Tree	AVL Tree	Red-Black Tree
Search	$O(\log N)$	$O(\log N)$	$O(\log N)$
Insert	$O(\log N)$	$O(\log N)$	$O(\log N)$
Delete	$O(\log N)$	$O(\log N)$	$O(\log N)$

Mutable Mapping

Summary

Average Time Complexity

	Array	Hash Table	Binary Search Tree
Key	integer	hashable object	comparable object
Ranged search	supported	not supported	supported
Search	$O(1)$	$O(1)$	$O(\log N)$
Insert	$O(N)$	$O(1)$	$O(\log N)$
Delete	$O(N)$	$O(1)$	$O(\log N)$

Binary Search Tree

Practices

- Range Sum of BST (Leetcode Problem 938)
- Unique Binary Search Trees (Leetcode Problem 96)