DATA 301
Data Structures and Algorithms

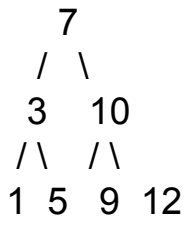
Dr. Aye Hninn Khine

PARAMI UNIVERSITY

Construct BST from a list of values

```
function build_BST(values): # [7, 3, 10, 1, 5, 9, 12]
    Initialize root as null
    for each value in values:
        root = insert_into_BST(root, value)
     return root
function insert_into_BST(node, value):
    if node is null:
        return new Node(value)
    if value < node.value:
        node.left = insert_into_BST(node.left, value)
    else:
        node.right = insert_into_BST(node.right, value)
    return node
```

Dry Run



Step	Insert Value	Path Followed	Inserted At
1	7	root	root
2	3	3 < 7 → left of 7	7.left
3	10	10 > 7 → right of 7	7.right
4	1	1 < 7 ➔ 1 < 3 → left of 3	3.left
5	5	5 < 7 ➔ 5 > 3 → right of 3	3.right
6	9	9 > 7 ➔ 9 < 10 → left of 10	10.left
7	12	12 > 7 ➔ 12 > 10 → right of 10	10.right

In-Class Practices

Exercise 1: [6, 2, 8, 1, 4, 7, 9]

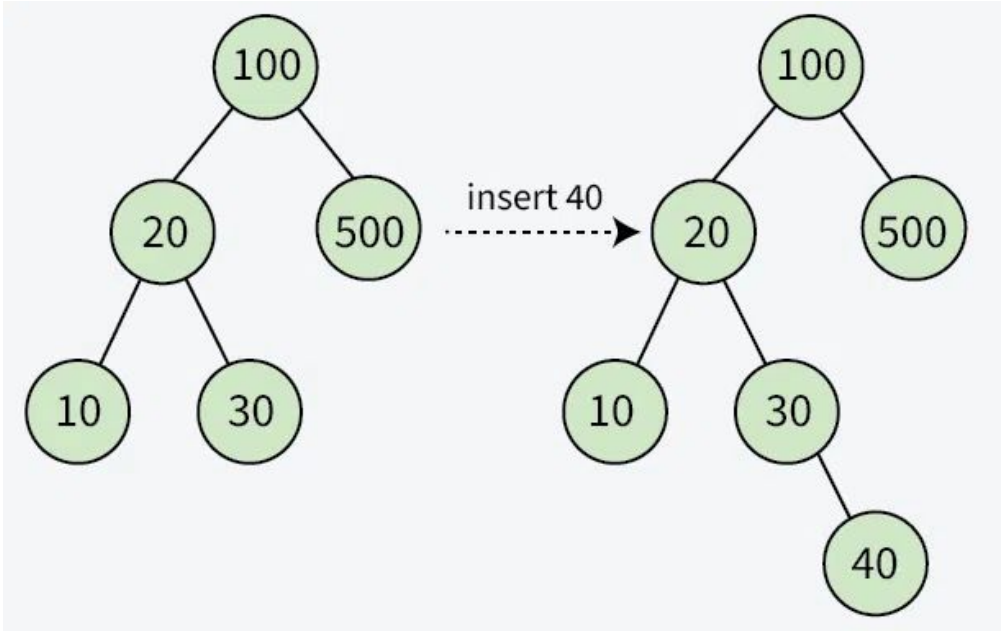
Exercise 2: [12, 5, 18, 2, 9, 15, 19, 13, 17]

Write Python Code

[Make a copy] - https://colab.research.google.com/drive/11e3ecBgrtxT4ApJwNvbOqkswqxOED269?usp=sharing

Insertion a key in a binary search tree

- Initialize the current node (say, **currNode or node**) with root node
- Compare the **key** with the current node.
- **Move left** if the **key** is less than or equal to the current node value.
- **Move right** if the **key** is greater than current node value.
- Repeat steps 2 and 3 until you reach a leaf node.
- Attach the **new key** as a left or right child based on the comparison with the leaf node's value.



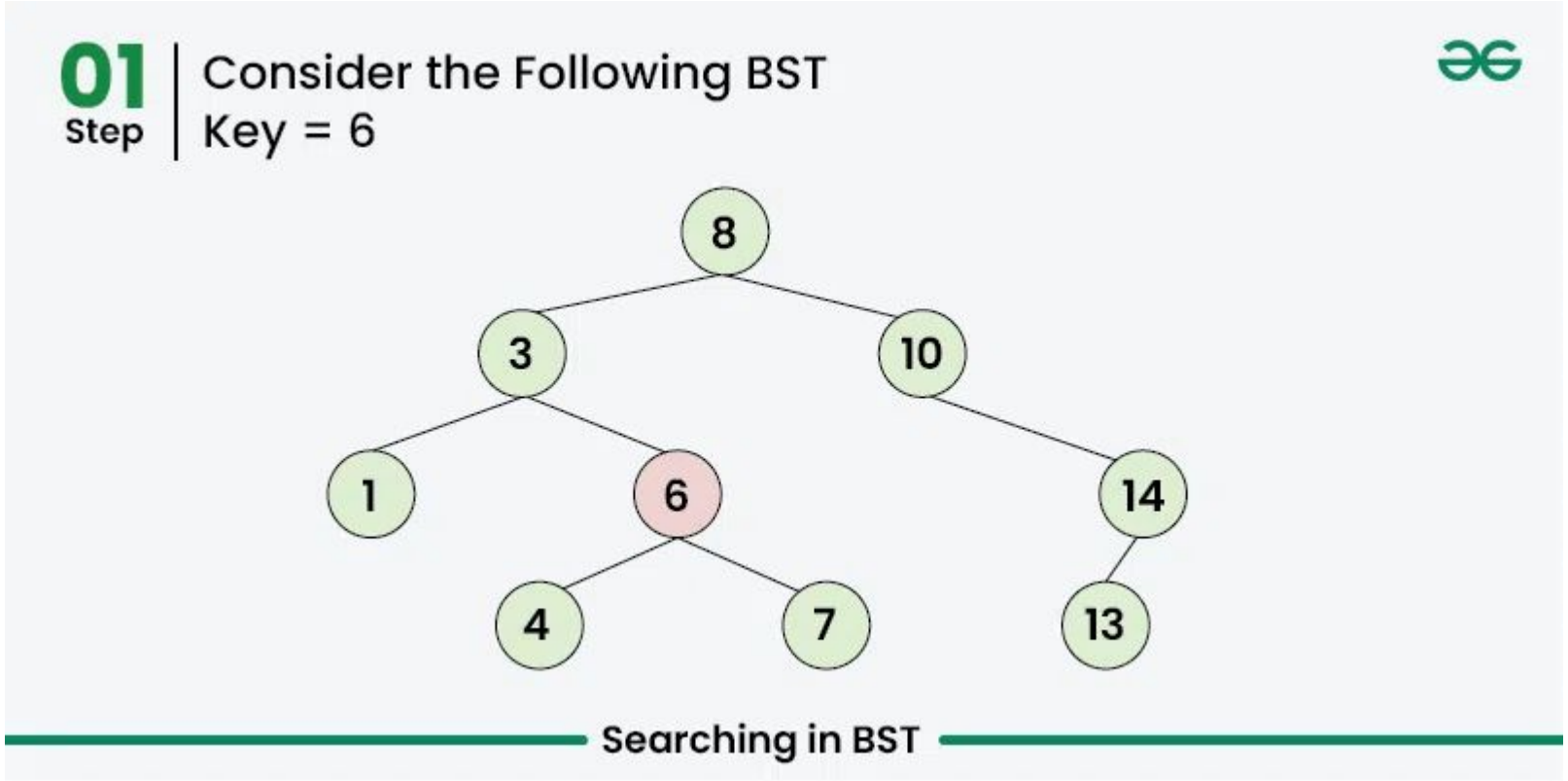
Insertion a key in a binary search tree

```
# A utility function to insert
# a new node with the given key
def insert(root, key):
    if root is None:
        return Node(key)
    if root.val == key:
            return root
    if root.val < key:
            root.right = insert(root.right, key)
    else:
            root.left = insert(root.left, key)
      return root
```

Time Complexity Analysis

- The worst-case time complexity of insert operations is **O(h)** where **h** is the height of the Binary Search Tree.
- In the worst case, we may have to travel from the root to the deepest leaf node. The height of a skewed tree may become **n** and the time complexity of insertion operation may become **O(n).**

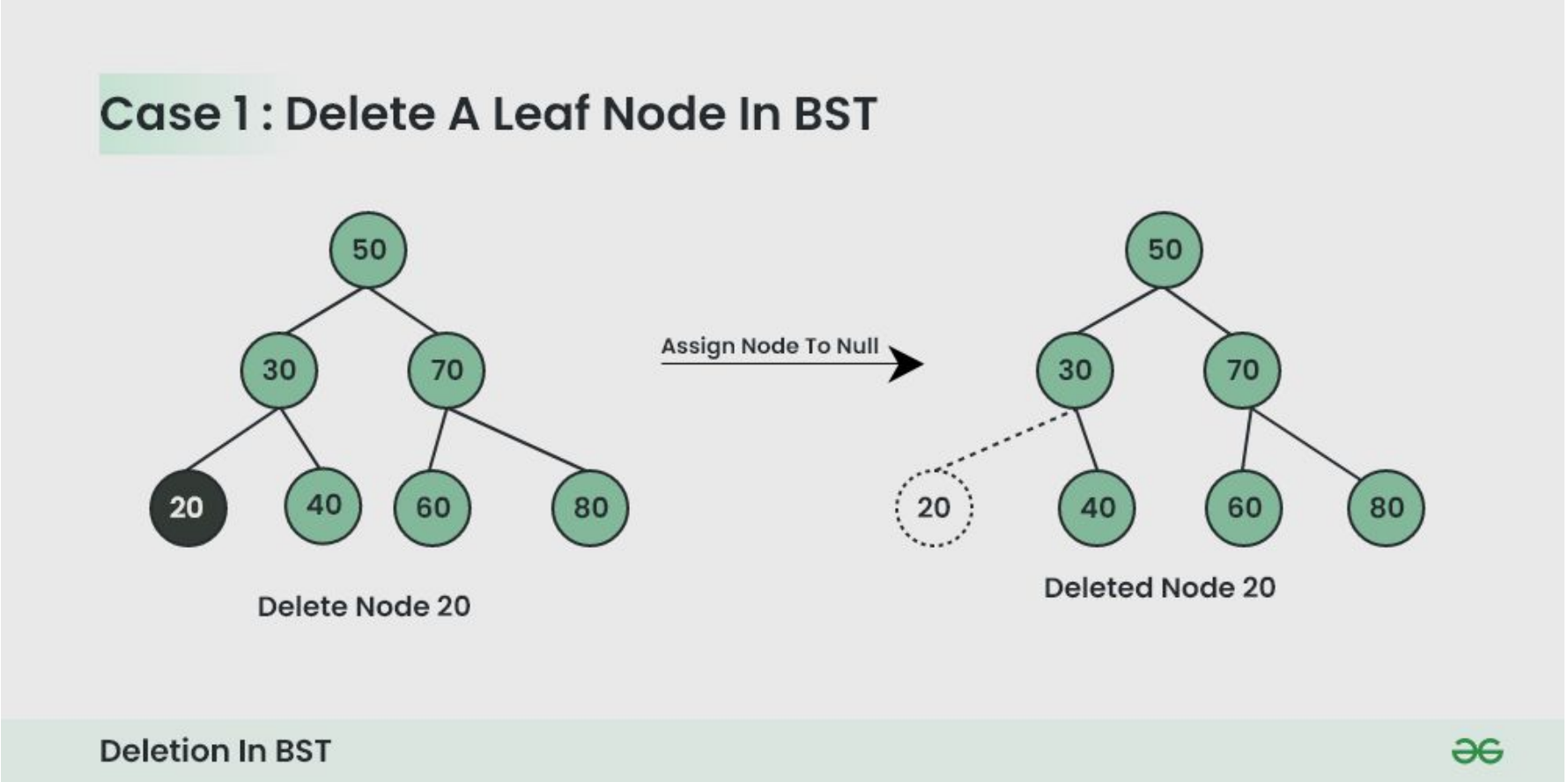
Searching in a BST



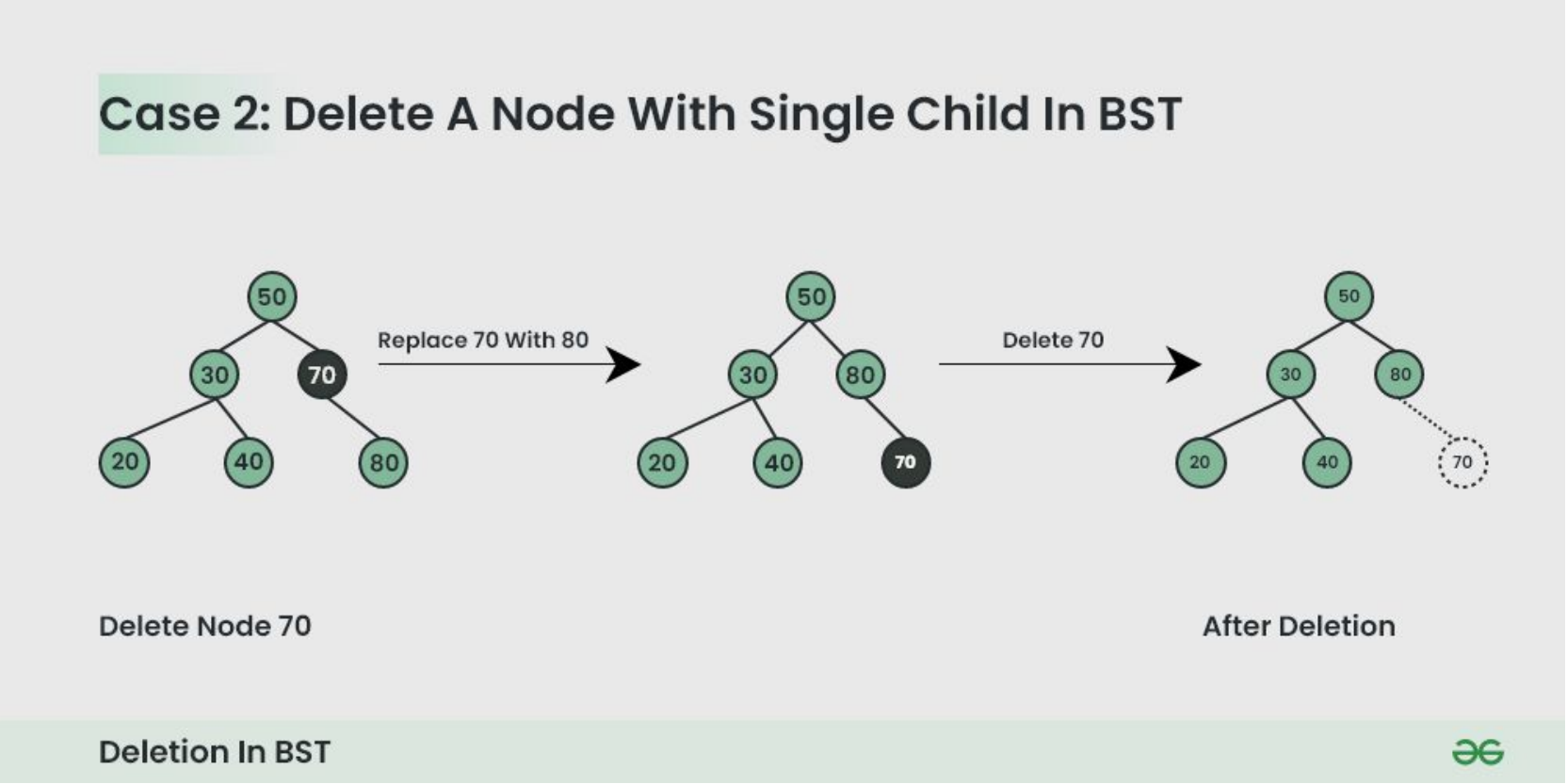
Searching in a BST

```
# function to search a key in a BST
def search(root, key):

    # Base Cases: root is null or key
    # is present at root
    if root is None or root.key == key:
        return root

    # Key is greater than root's key
    if root.key < key:
        return search(root.right, key)

    # Key is smaller than root's key
    return search(root.left, key)
```

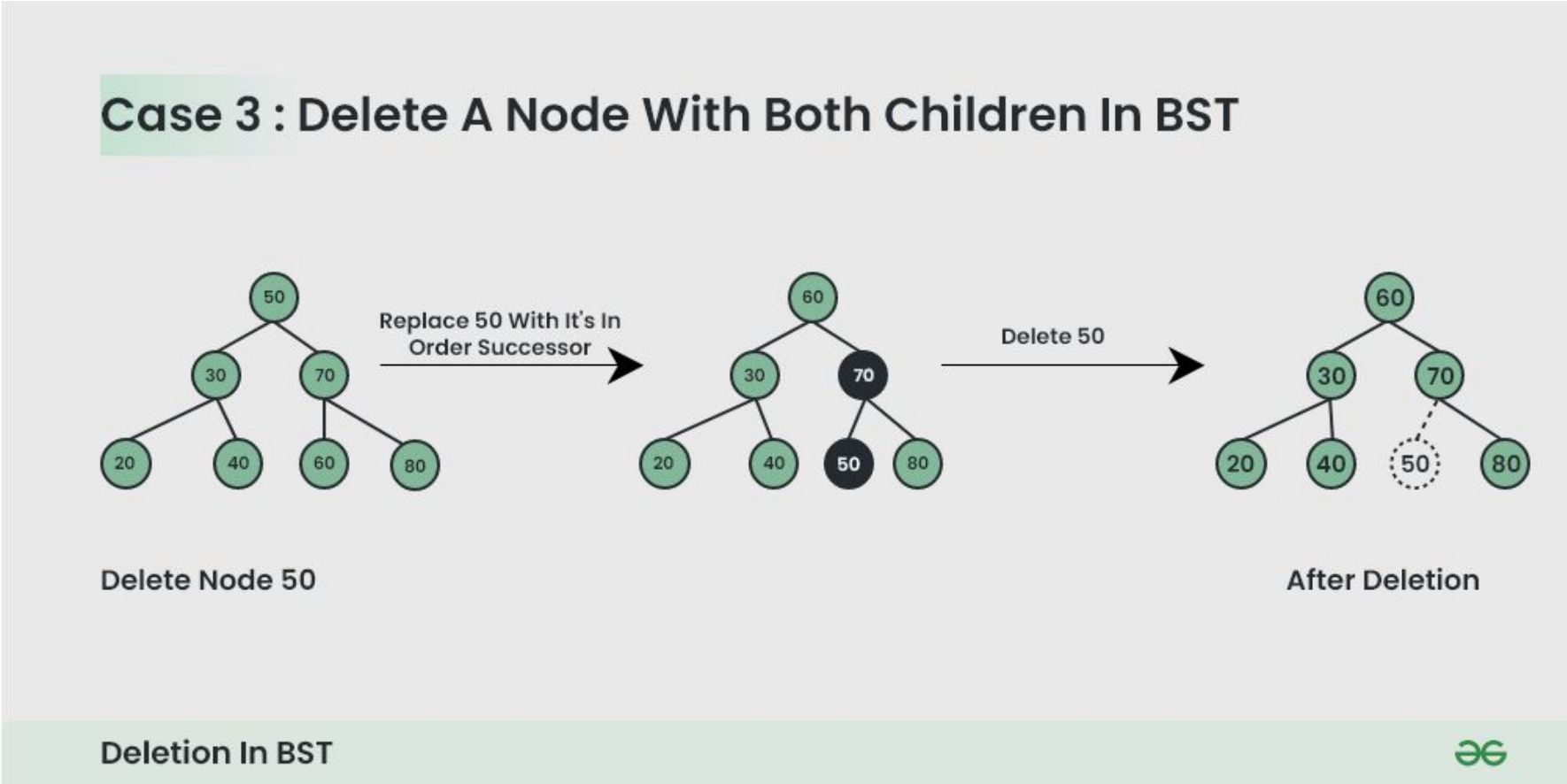
Deletion (Case 1)



Deletion (Case 2)



Deletion (Case 3) - find the smallest node in right subtree



Deletion (Pseudo Code)

Case 1

```
if root.left is NULL and root.right is NULL:
        delete root
        return NULL
```

Case 2

```
if root.left is NULL:
        temp = root.right
        delete root
        return temp
 else if root.right is NULL:
        temp = root.left
        delete root
        return temp
```

Deletion (Pseudo Code)

Case 3

```
 temp = FindMin(root.right)
            root.value = temp.value
            root.right = DeleteNode(root.right, temp.value)
```

```
Function FindMin(node):
    while node.left is not NULL:
        node = node.left
    return node
```

Deletion (Case 1) - Dry Run

🥇 Case 1: Deleting a **Leaf Node** (No Children)

Tree Before Deletion:

markdown Copy Edit

```
      8
     / \
    3   10
   / \
  1   6
```

Task: Delete node `1`

Dry Run Steps:

Step	Current Node	Value to Delete	Decision (Left/Right/Found)	Action
1	8	1	1 < 8 → Go Left	Move to 3
2	3	1	1 < 3 → Go Left	Move to 1
3	1	1	Found	Delete (leaf node, simply remove)

Deletion (Case 2) - Dry Run

🥈 Case 2: Deleting a **Node with One Child**

Tree Before Deletion:

markdown Copy Edit

```
    8
   / \
  3   10
 / \
1   6
     \
      7
```

Task: Delete node `6`

Dry Run Steps:

Step	Current Node	Value to Delete	Decision (Left/Right/Found)	Action
1	8	6	6 < 8 → Go Left	Move to 3
2	3	6	6 > 3 → Go Right	Move to 6
3	6	6	Found	Replace node with its right child (7)

Tree After Deletion:

markdown Copy Edit

```
    8
   / \
  3   10
 / \
1   7
```

PARAMI UNIVERSITY

🥉 Case 3: Deleting a **Node with Two Children**

Tree Before Deletion:

markdown Copy Edit

```
    8
   / \
  3   10
 / \
1   6
   / \
  4   7
```

Task: Delete node `3`

Dry Run Steps:

Step	Current Node	Value to Delete	Decision (Left/Right/Found)	Action
1	8	3	3 < 8 → Go Left	Move to 3
2	3	3	Found	Find in-order successor (smallest in right subtree)
3	6	(Right child of 3)	Move to 4 (leftmost)	4 is the successor
4	Replace value of 3 with 4	Now delete 4	4 is a leaf node	

Tree After Deletion:

markdown Copy Edit

```
    8
   / \
  4   10
 / \
1   6
     \
      7
```

Exercises

1. Manual Tracing Exercise

Insert the following numbers into an initially empty BST in order:

45, 20, 60, 10, 30, 50, 70

- Draw the BST after all insertions.
- What is the inorder traversal of the BST?

Exercises

2. Search Exercise

Given the BST created above:

- Search for the following numbers and state if they are found or not:
  - 30
  - 15
  - 60
  - 75

Exercises

3. Deletion - Case 1 (No Child)

Delete a **leaf node** (node with no children) from the BST:

- Delete 10.
- Draw the BST after deletion.

4. Deletion - Case 2 (One Child)

Delete a **node with only one child**:

- Delete 30.
- Draw the BST after deletion.

5. Deletion - Case 3 (Two Children)

Delete a **node with two children**:

- Delete 45.
- Draw the BST after deletion.

REFERENCES

The lectures are adapted from
https://www.geeksforgeeks.org/

Thank You