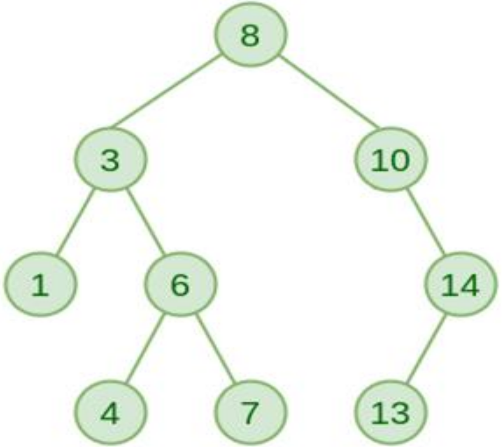
Binary Search Tree

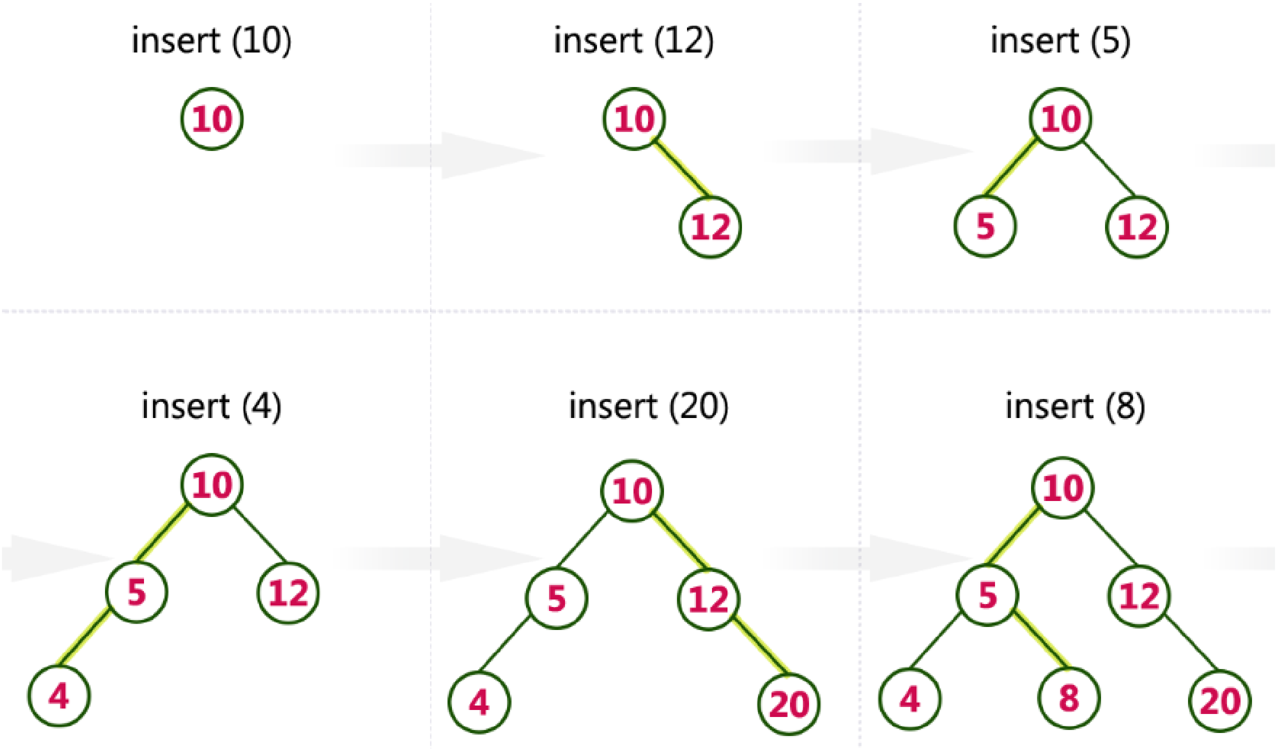
- A Binary Search Tree (BST) is a special type of binary tree
- Properties of BST::
  - The left subtree of a node contains only nodes with keys lesser than the node's key.
  - The right subtree of a node contains only nodes with keys greater than the node's key.
  - The left and right subtree each must also be a binary search tree.
- It makes it possible to efficiently search, insert, and delete elements in the tree.



Example

Construct a Binary Search Tree by inserting the following sequence of numbers...

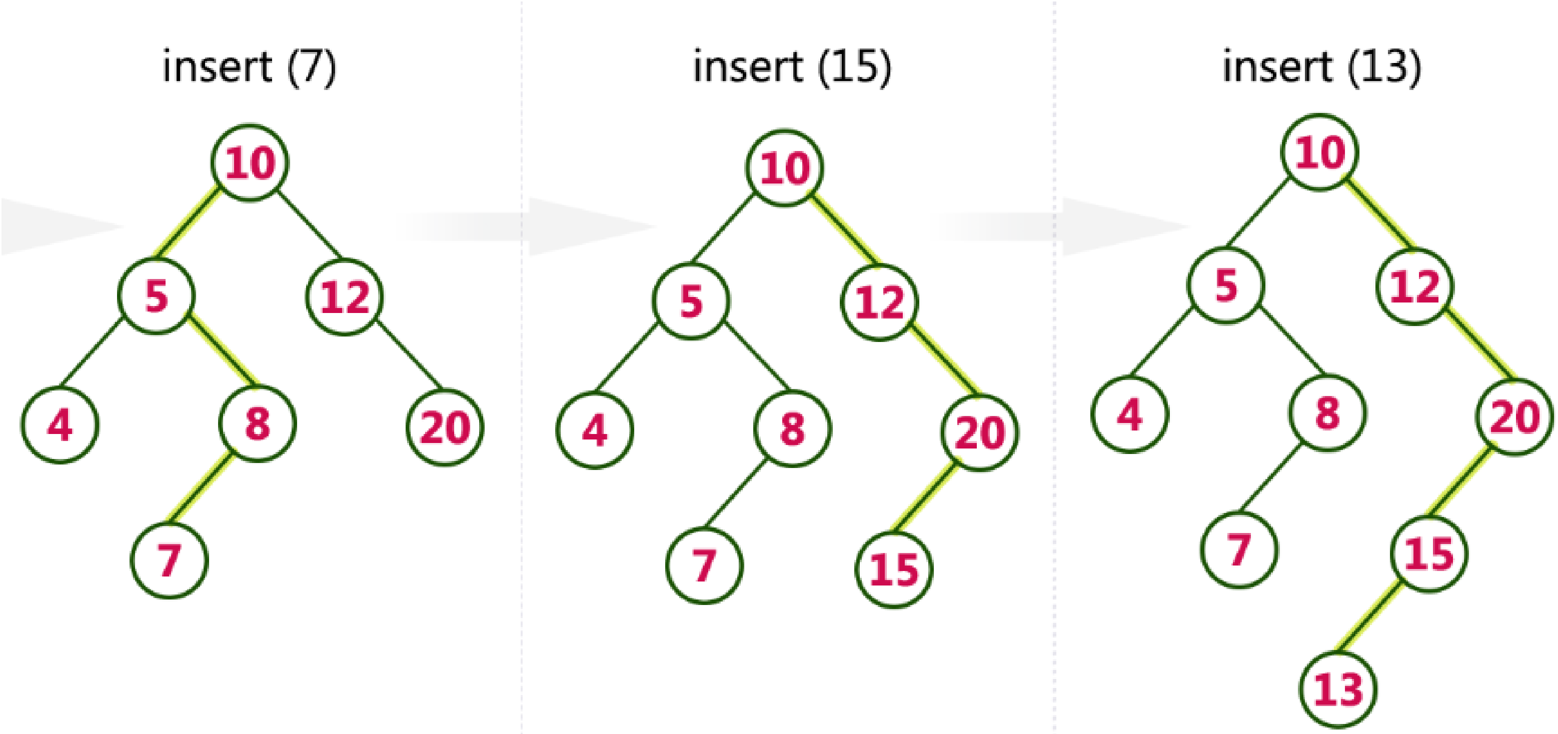
10,12,5,4,20,8,7,15 and 13



Example

Construct a Binary Search Tree by inserting the following sequence of numbers...

10,12,5,4,20,8,7,15 and 13



insert (7)

insert (15)

insert (13)

BST

```
class Node:
    def __init__(self, data):
        self.data = data
        self.leftChild = None
        self.rightChild = None

class BinarySearchTree:
    def __init__(self):
        self.root = None
```

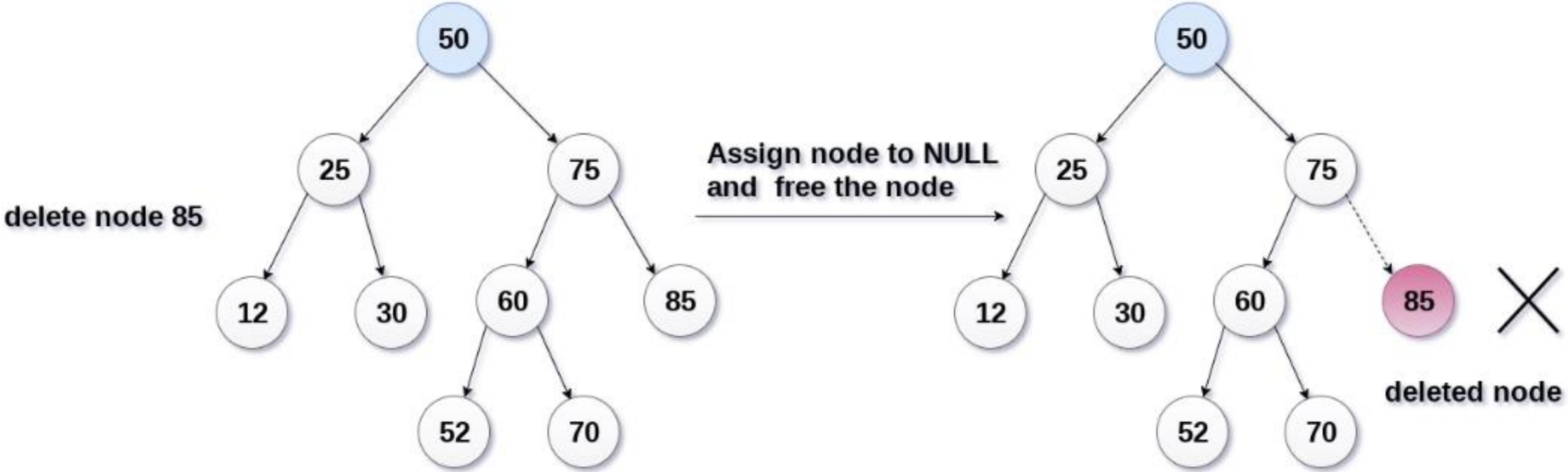
```
def _insert(self, node, value):
        if node is None:
            return Node(value)
        if value < node.data:
            node.leftChild = self._insert(node.leftChild, value)
        elif value > node.data:
            node.rightChild = self._insert(node.rightChild, value)
        return node
```

```
self.root = self._insert(self.root, value)
```

Deletion

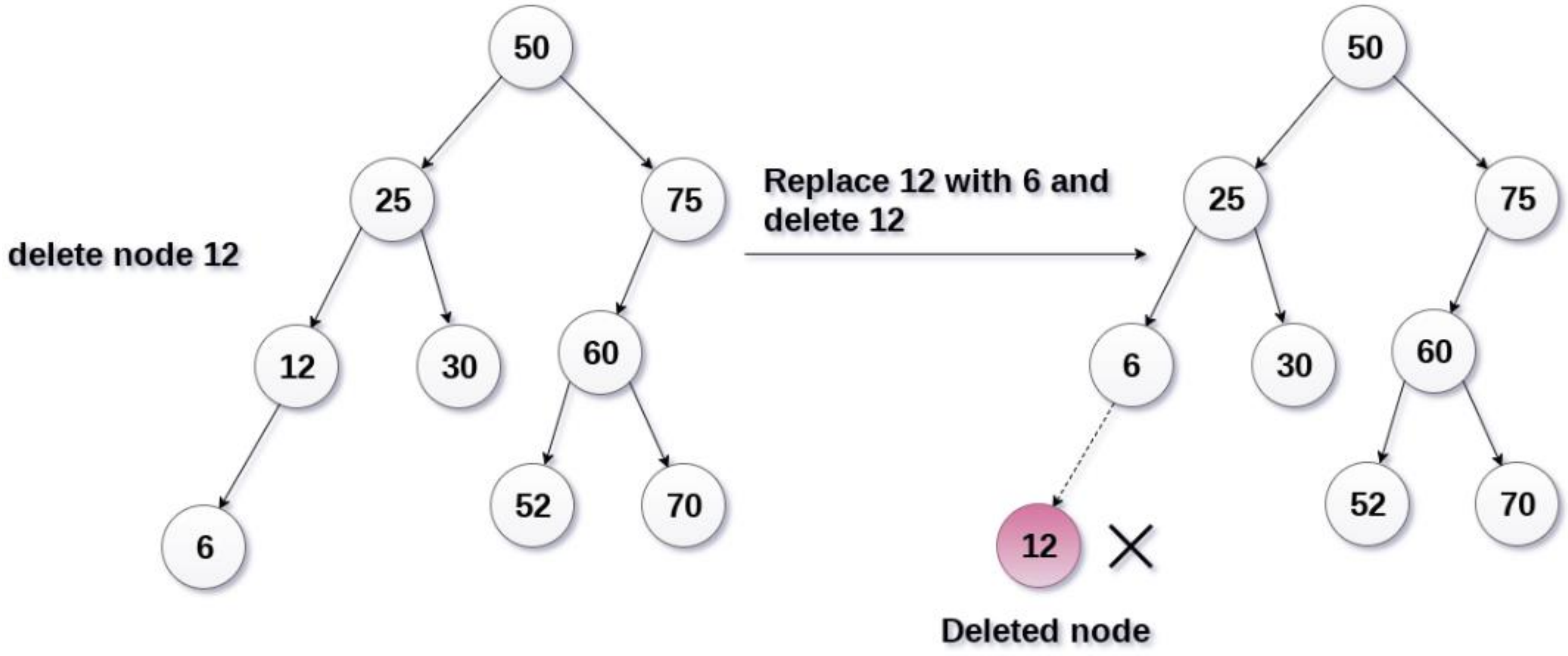
When we delete a node, three possibilities arise.

1) Node to be deleted is the leaf: Simply remove it from the tree.



Deletion

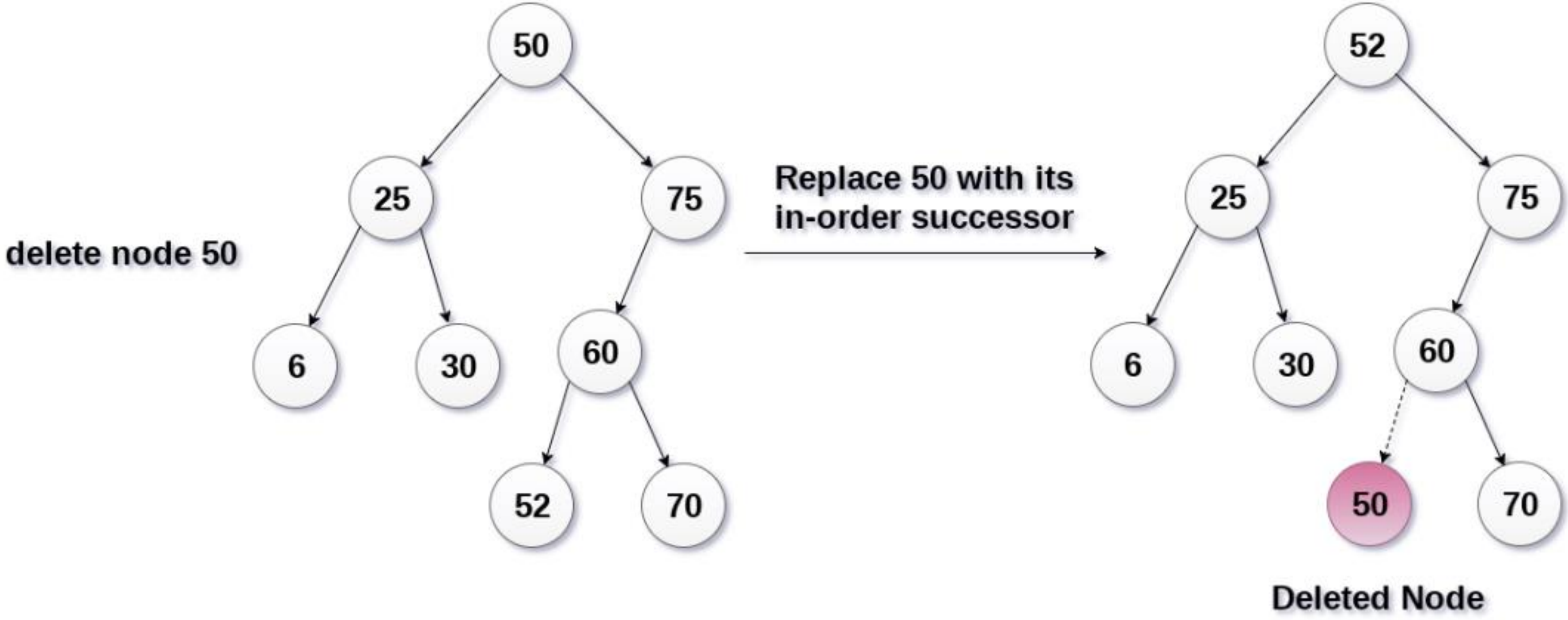
2) **Node to be deleted has only one child:** replace the node with its child and delete the node.



Deletion

3) **Node to be deleted has two children:** the node that is to be removed is replaced by its in-order successor or predecessor recursively.

In Binary Tree, Inorder successor of a node is the next node in Inorder traversal of the Binary Tree.



```
def find_minimum(self, node):
    if node.leftChild is None:
        return node.data
    return self.find_minimum(node.leftChild)
```

```
def _delete(self, node, value):
    if node is None:
        return None
    if value < node.data:
        node.leftChild = self._delete(node.leftChild, value)
    elif value > node.data:
        node.rightChild = self._delete(node.rightChild, value)
    else:
        if node.leftChild is None:
            return node.rightChild
        elif node.rightChild is None:
            return node.leftChild
        min_value = self.find_minimum(node.rightChild)
        node.data = min_value
        node.rightChild = self._delete(node.rightChild, min_value)
    return node
```

```
self.root = self._delete(self.root, value)
```

```
def _search(self, node, value):
        if node is None or node.data == value:
            return node
        if value < node.data:
            return self._search(node.leftChild, value)
        return self._search(node.rightChild, value)

 def _print_tree(self, node):
        if node:
            self._print_tree(node.leftChild)
            print(node.data, end=" ")
            self._print_tree(node.rightChild)
```