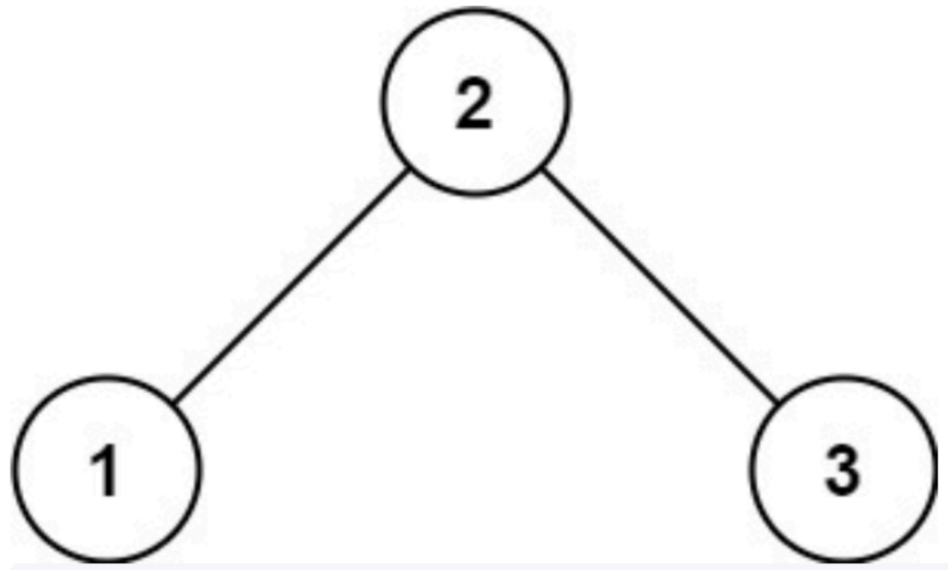
98. Validate Binary Search Tree

Given the `root` of a binary tree, *determine if it is a valid binary search tree (BST)*.

A **valid BST** is defined as follows:

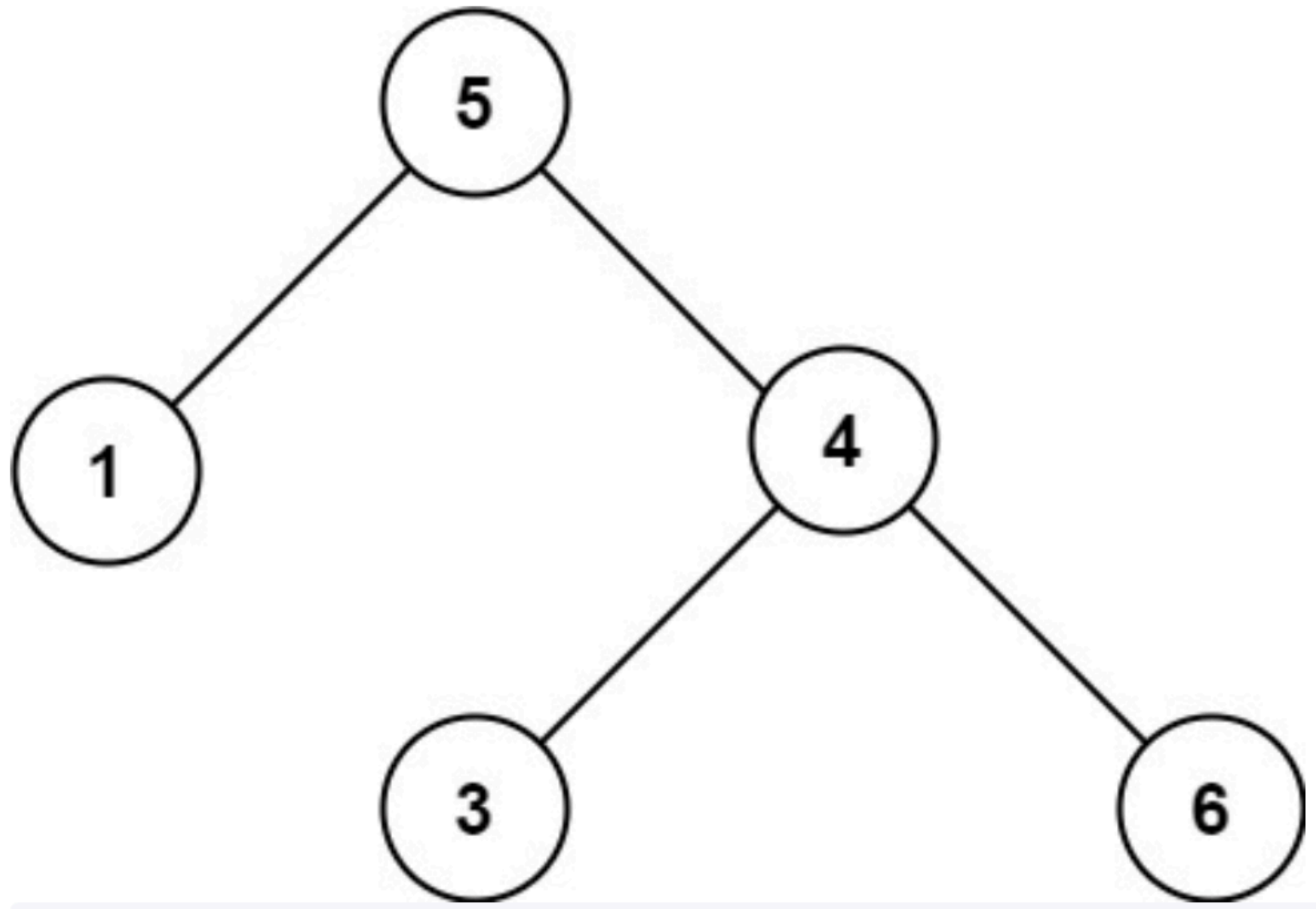
- The left subtree of a node contains only nodes with keys **less than** the node's key.
- The right subtree of a node contains only nodes with keys **greater than** the node's key.
- Both the left and right subtrees must also be binary search trees.

Example 1:




```
Input: root = [2,1,3]
Output: true
```

Example 2:



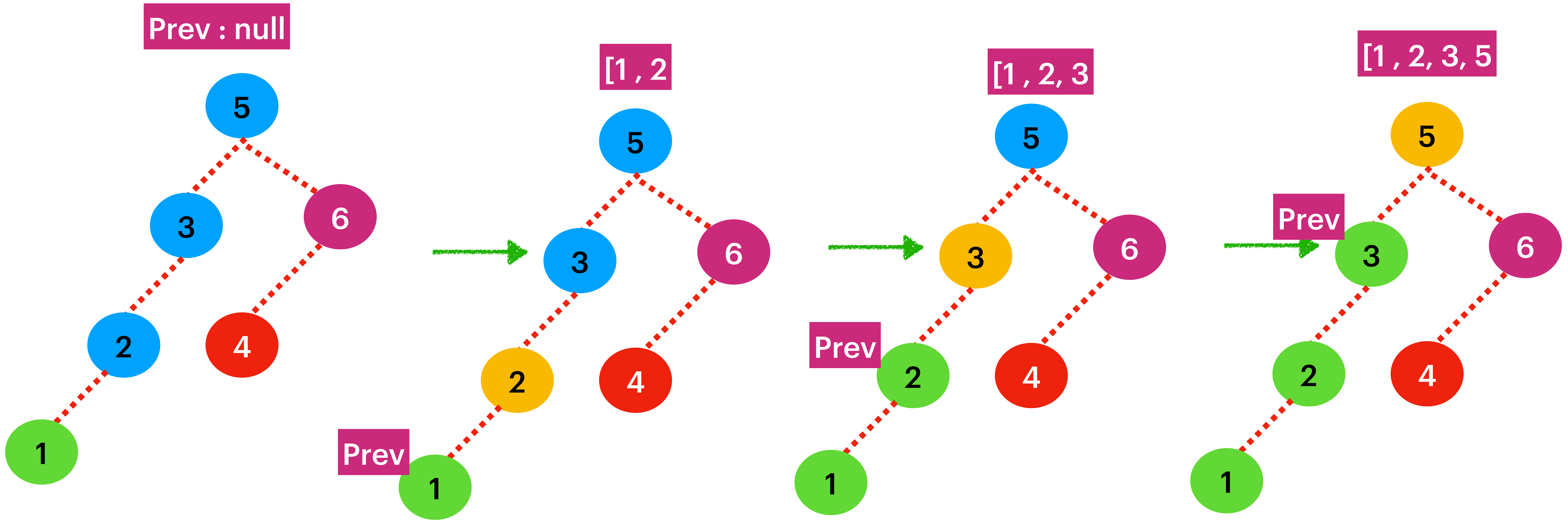

```
Input: root = [5,1,4,null,null,3,6]
Output: false
Explanation: The root node's value is 5 but its right child's
value is 4.
```

Constraints:

- The number of nodes in the tree is in the range $[1, 10^4]$.
- $-2^{31} <= \text{Node.val} <= 2^{31} - 1$

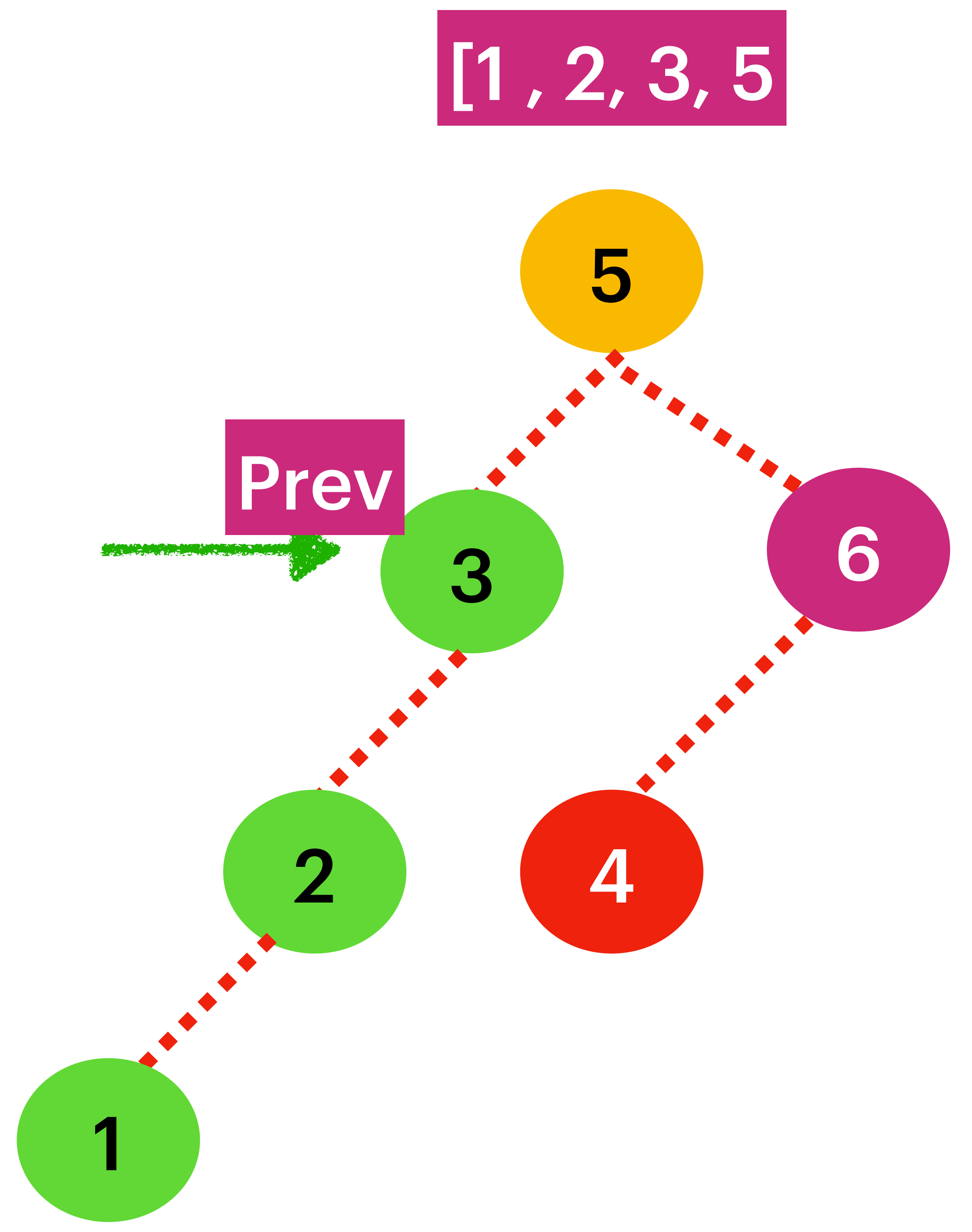
Algorithm :
For a valid binary search tree we always get elements in ascending order for InOrder traversal.

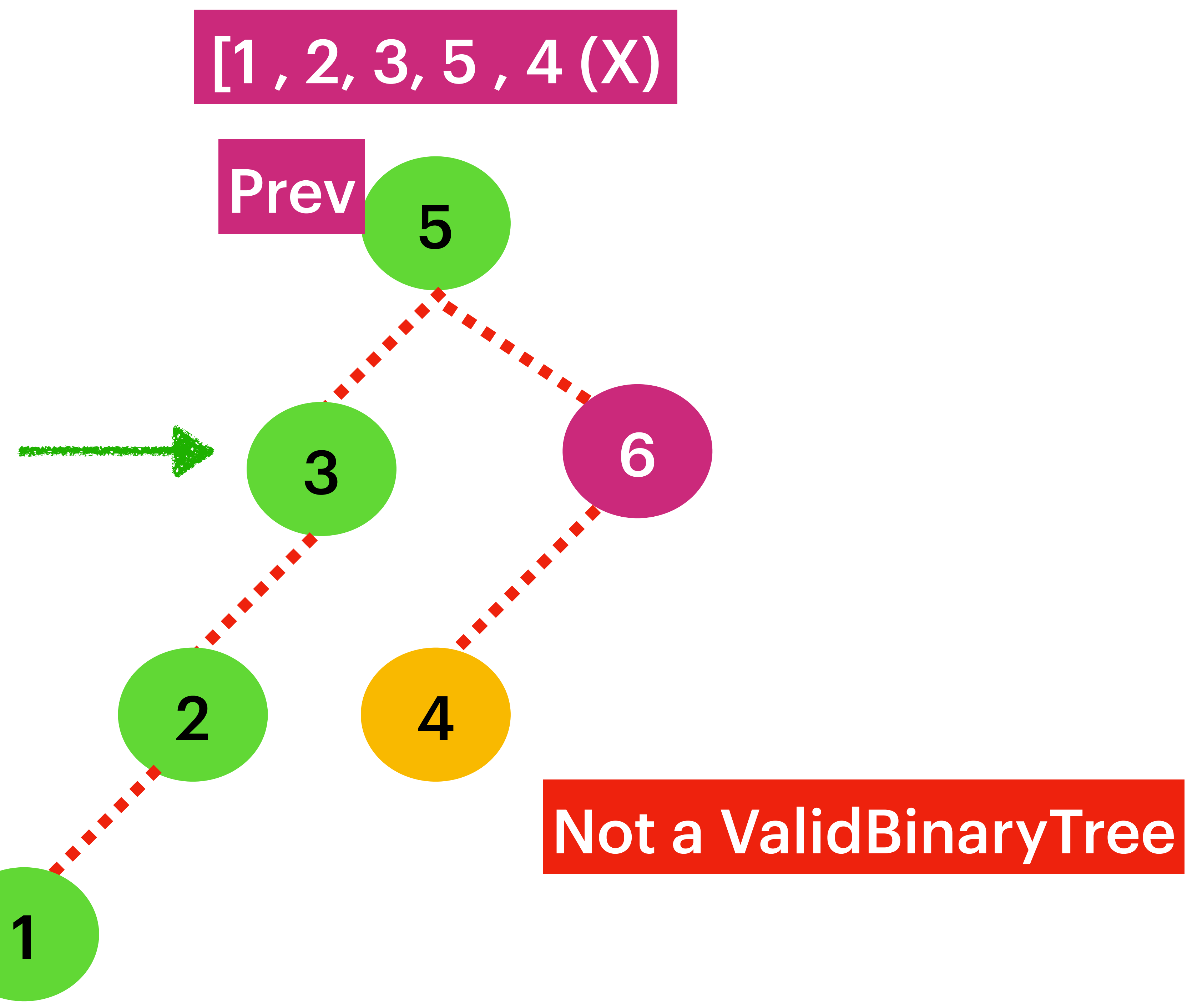
In an ascending order the next element > previous element.



Algorithm :
For a valid binary search tree we always get
elements in ascending order for InOrder traversal.

In an ascending order the next element > previous element.





1008. Construct Binary Search Tree from Preorder Traversal

Given an array of integers preorder, which represents the **preorder traversal** of a BST (i.e., **binary search tree**), construct the tree and return *its root*.

It is **guaranteed** that there is always possible to find a binary search tree with the given requirements for the given test cases.

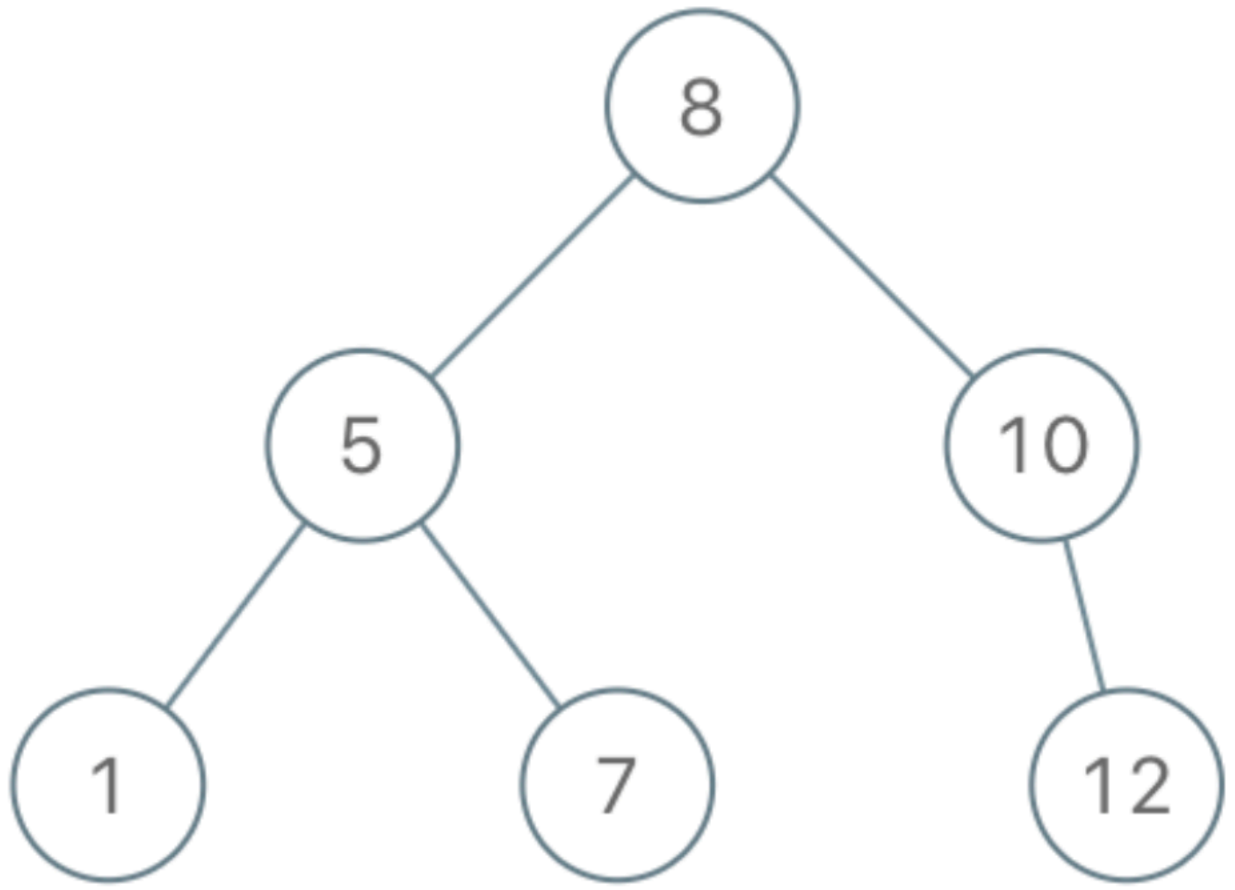
A **binary search tree** is a binary tree where for every node, any descendant of `Node.left` has a value **strictly less than** `Node.val`, and any descendant of `Node.right` has a value **strictly greater than** `Node.val`.

A **preorder traversal** of a binary tree displays the value of the node first, then traverses `Node.left`, then traverses `Node.right`.

Constraints:

- `1 <= preorder.length <= 100`
- `1 <= preorder[i] <= 1000`
- All the values of `preorder` are **unique**.

Example 1:




```
Input: preorder = [8,5,1,7,10,12]
Output: [8,5,10,1,7,null,12]
```

Example 2:

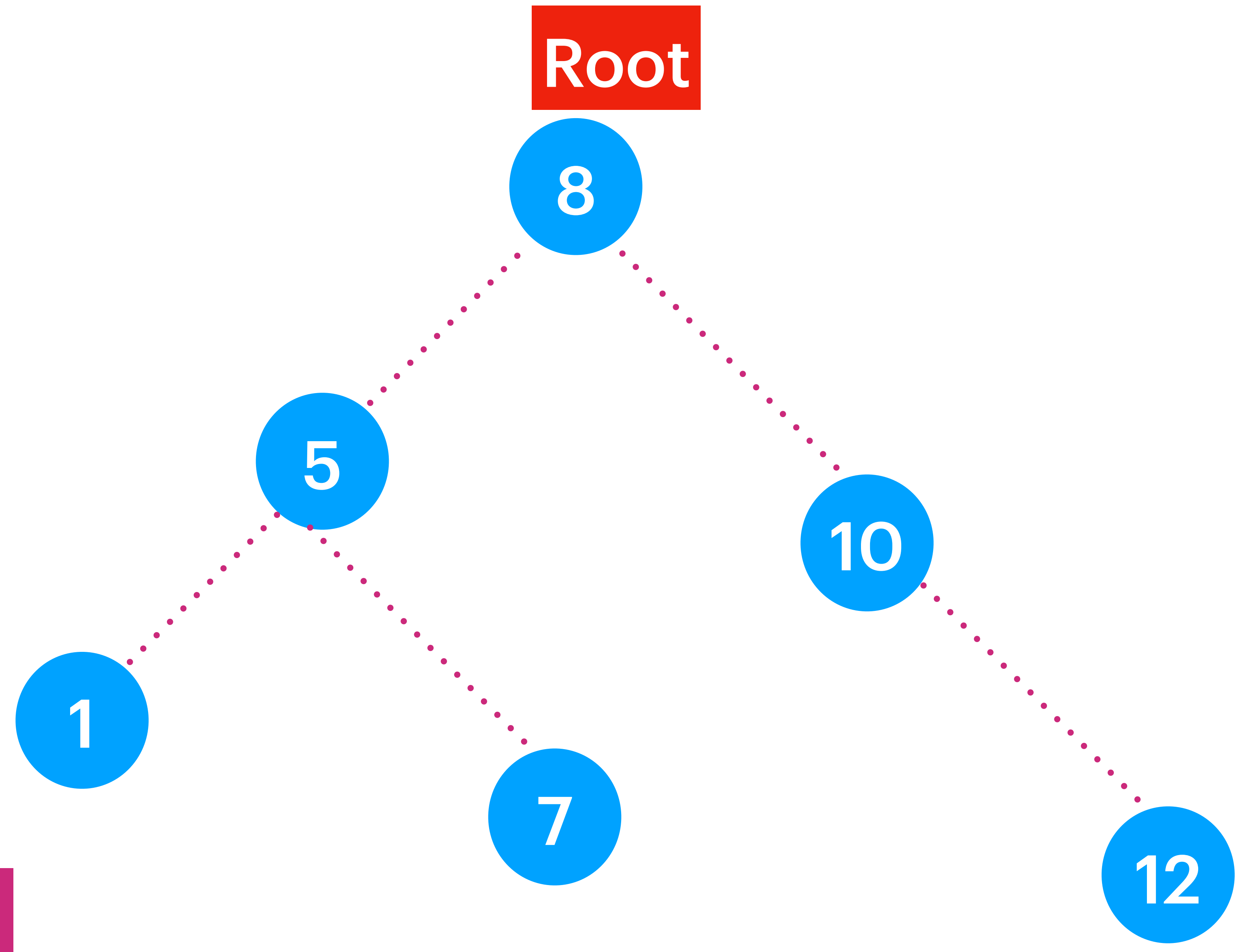
```
Input: preorder = [1,3]
Output: [1,null,3]
```

Input: preorder = [8,5,1,7,10,12]

Current

10

Child



Deque

12

Time Complexity : O(n)
Space Complexity : O(n)

1 5 7 8 10

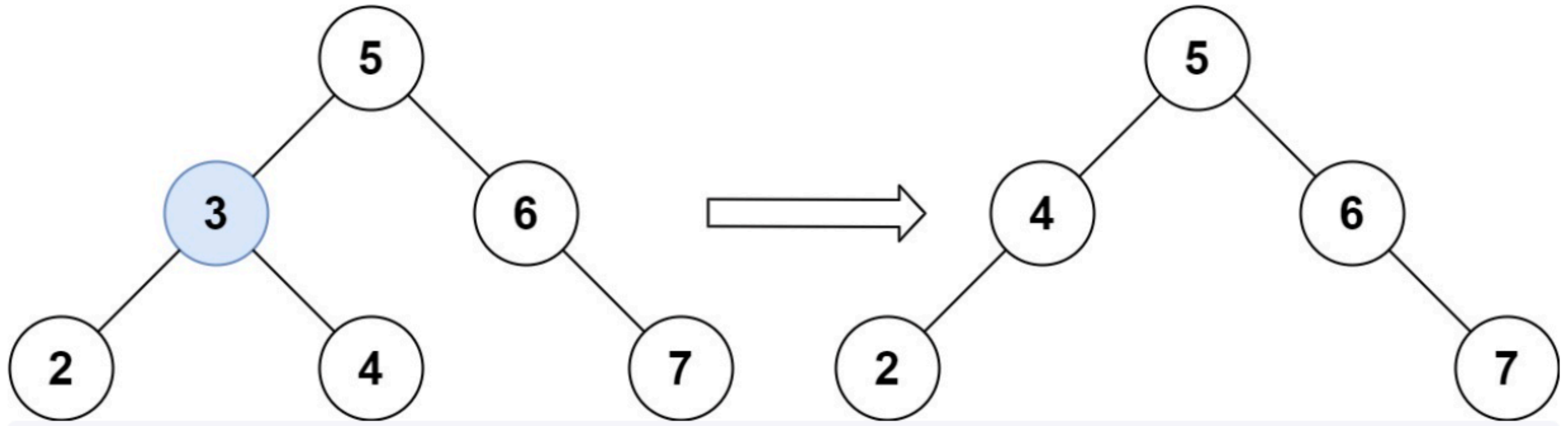
450. Delete Node in a BST

Given a root node reference of a BST and a key, delete the node with the given key in the BST. Return the root node reference (possibly updated) of the BST.

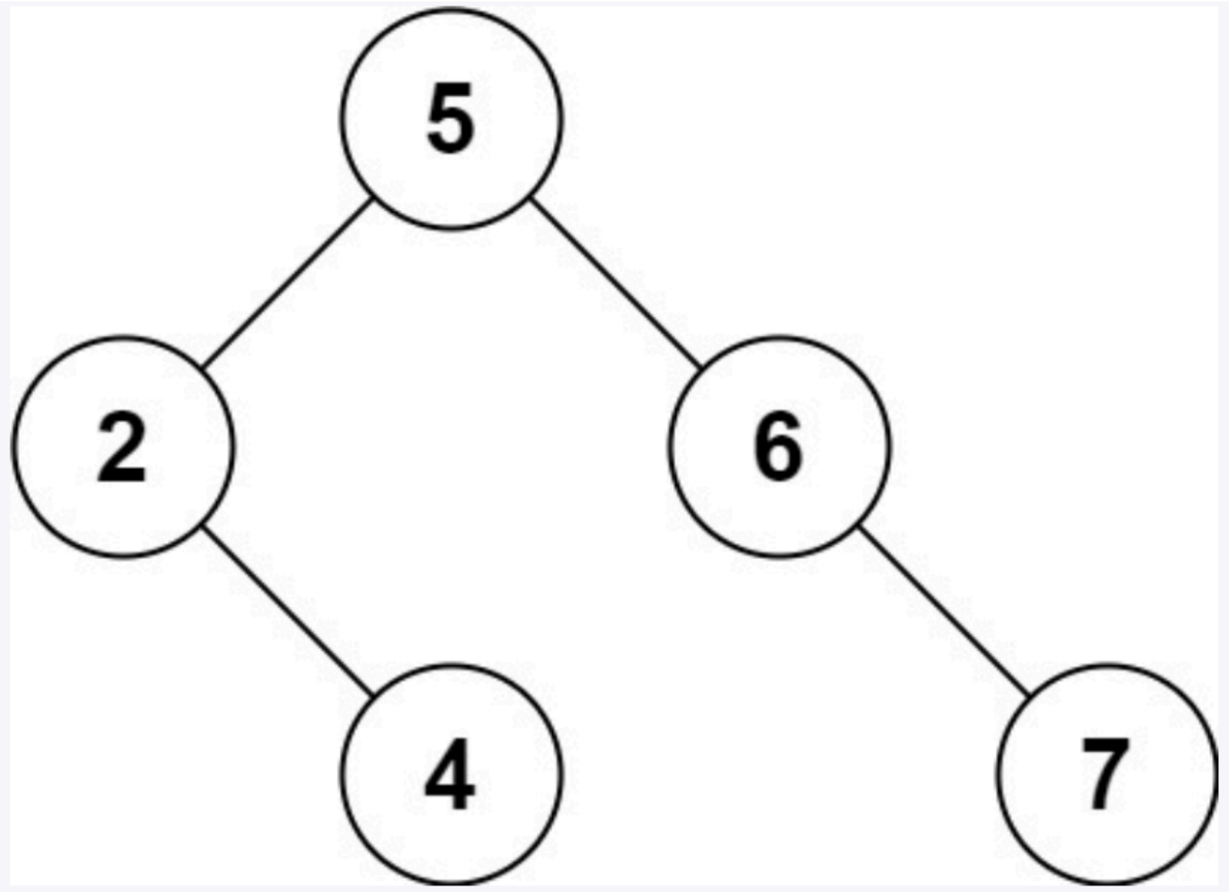
Basically, the deletion can be divided into two stages:

1. Search for a node to remove.
2. If the node is found, delete the node.

Example 1:




```
Input: root = [5,3,6,2,4,null,7], key = 3
Output: [5,4,6,2,null,null,7]
Explanation: Given key to delete is 3. So we find the node with value 3 and delete it.
One valid answer is [5,4,6,2,null,null,7], shown in the above BST.
Please notice that another valid answer is [5,2,6,null,4,null,7] and it's also accepted.
```



Example 2:

```
Input: root = [5,3,6,2,4,null,7], key = 0
Output: [5,3,6,2,4,null,7]
Explanation: The tree does not contain a node with value = 0.
```

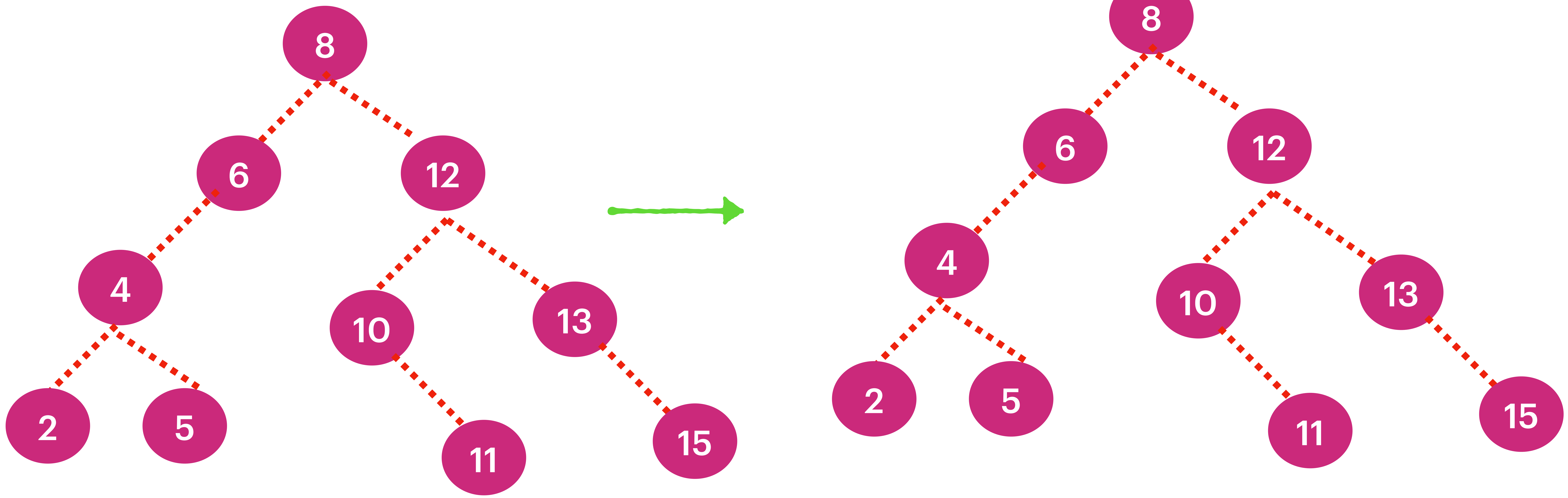
Example 3:

```
Input: root = [], key = 0
Output: []
```

Constraints:

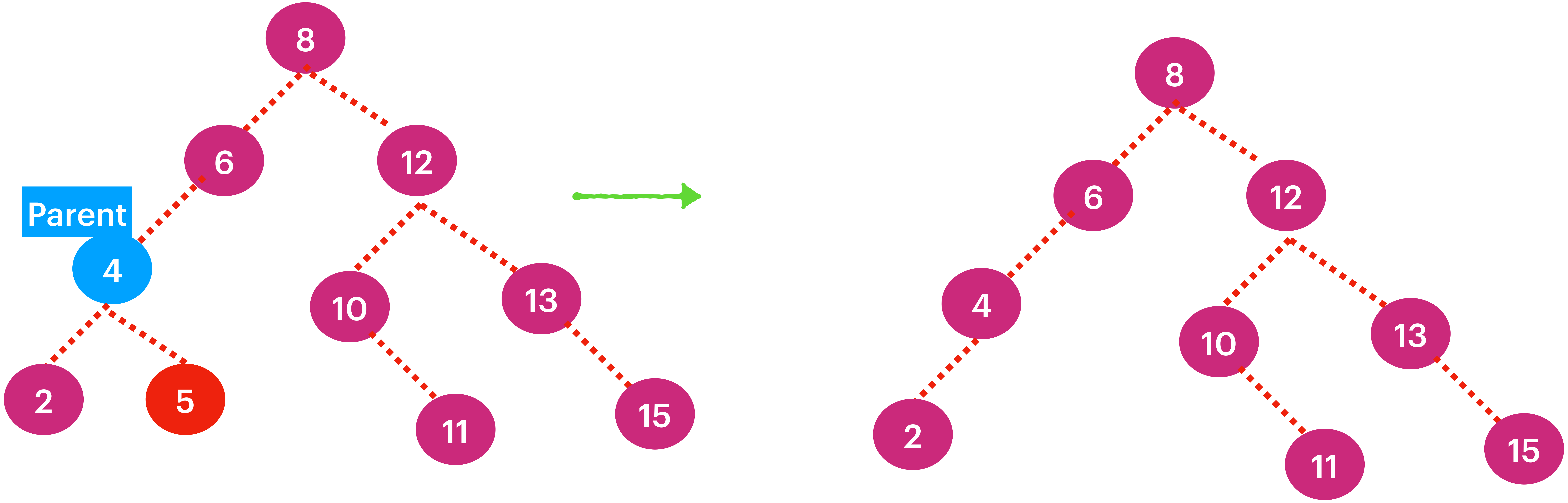
- The number of nodes in the tree is in the range `[0, 10^4]`.
- -10^5 <= Node.val <= 10^5
- Each node has a **unique** value.
- `root` is a valid binary search tree.
- -10^5 <= key <= 10^5

Case 1 : Element is Not Found delete(18)
Return same root



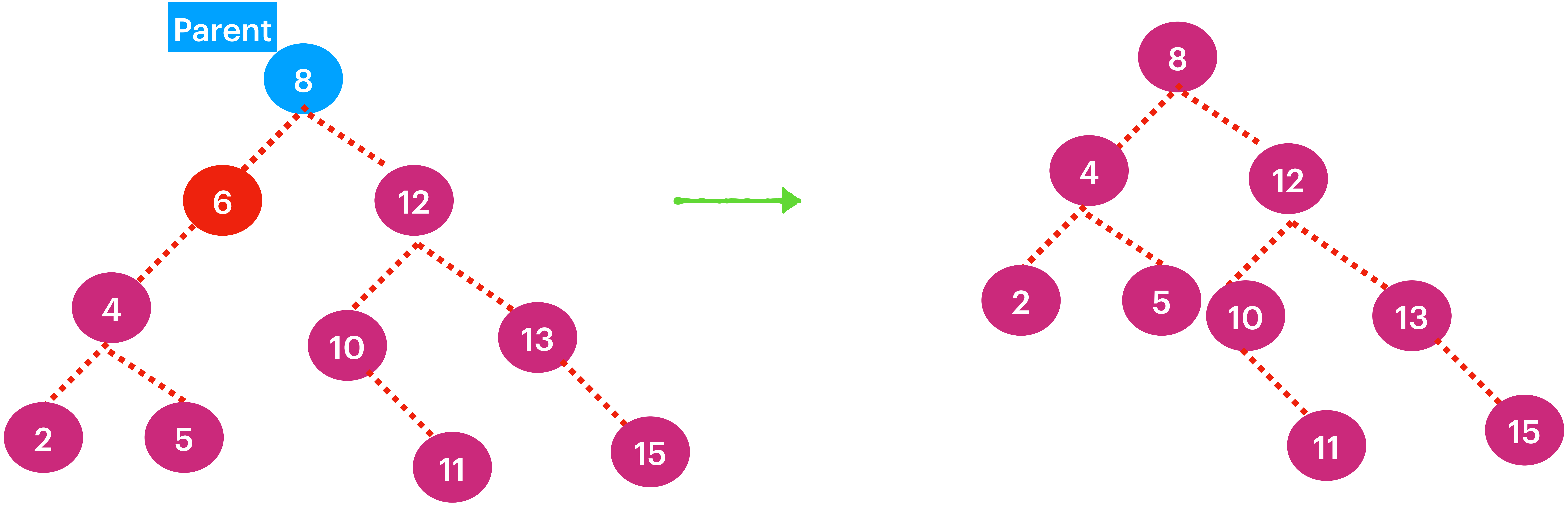
Case 2 : Element is a leaf Node,
does not have left & right child would be null
Return same root . Ex delete(5)

If leafNode is right of parent then mark parent.right = null else
Parent.left = null



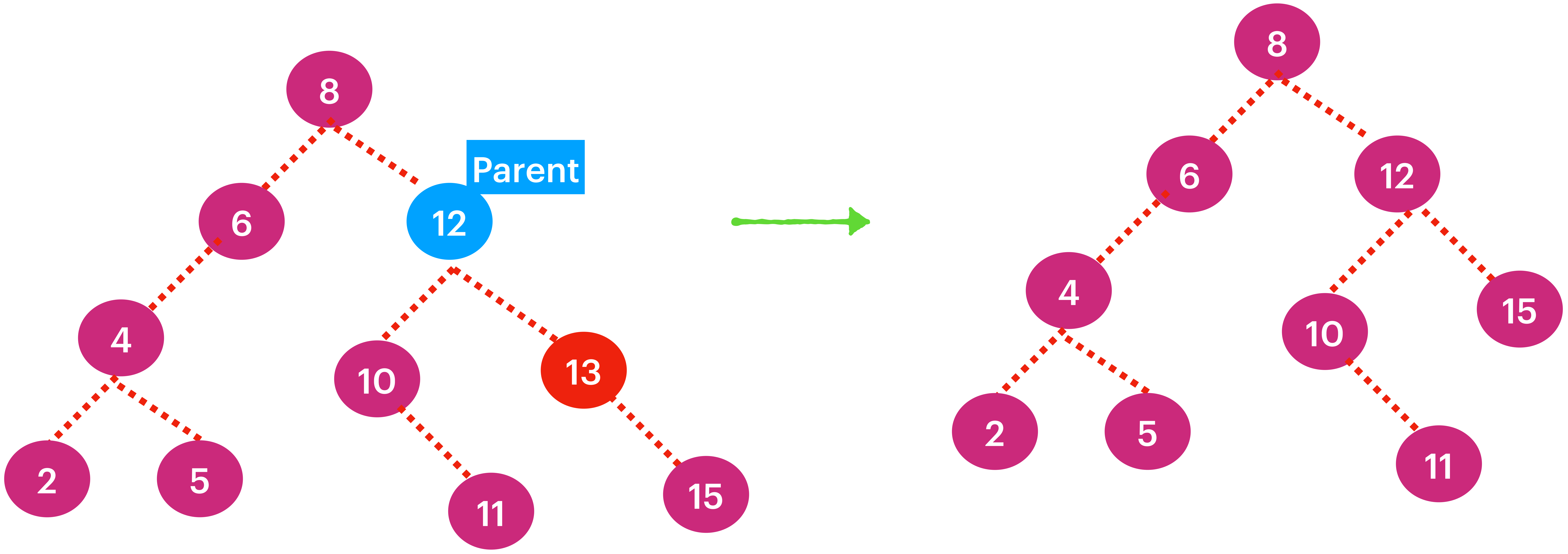
Case 3 : Element has only Left Child

Make parent.left = element.left



Case 4 : Element has only right Child

Make parent.right= element.right



Case 5 : Element has both the Childs

Two possibilities :

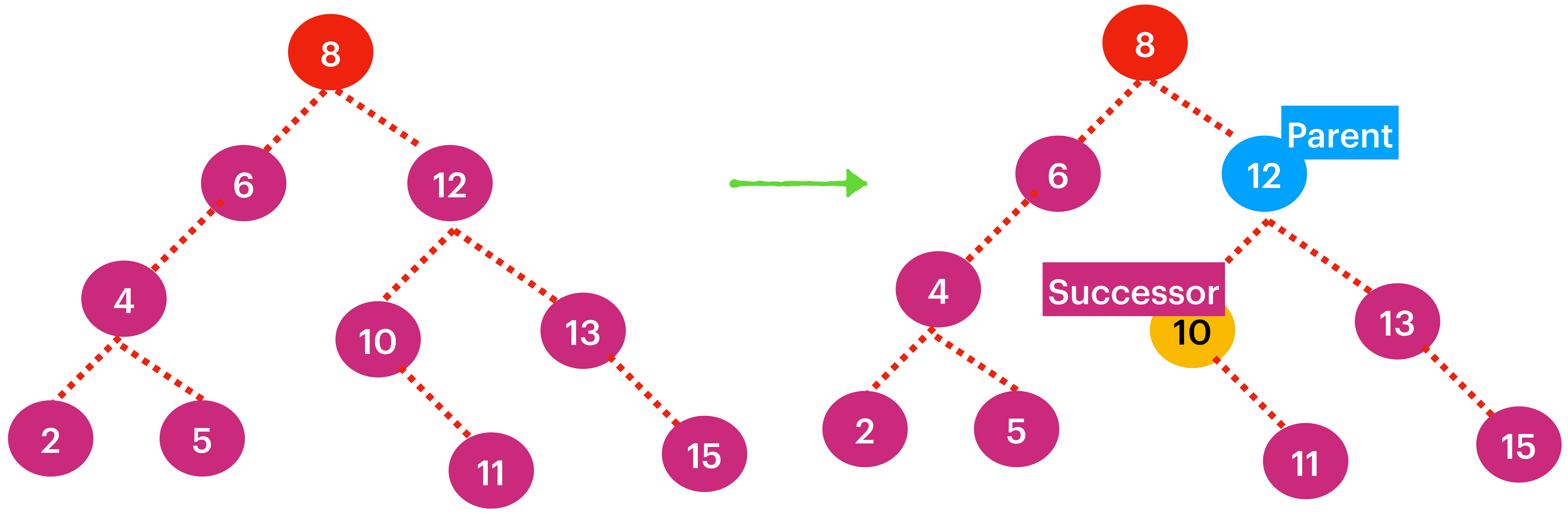
1. Get the max element from left subtree then replace. (Precedessor)
2. Get the min element from right subtree then replace. (Successor)

Lets go for 2nd possibility :

Successor => take element.right then pick least possible element.left

element.val = successor.val

Then remove successor => parent.left / right = successor.right



Case 5 : Element has both the Childs

Two possibilities :

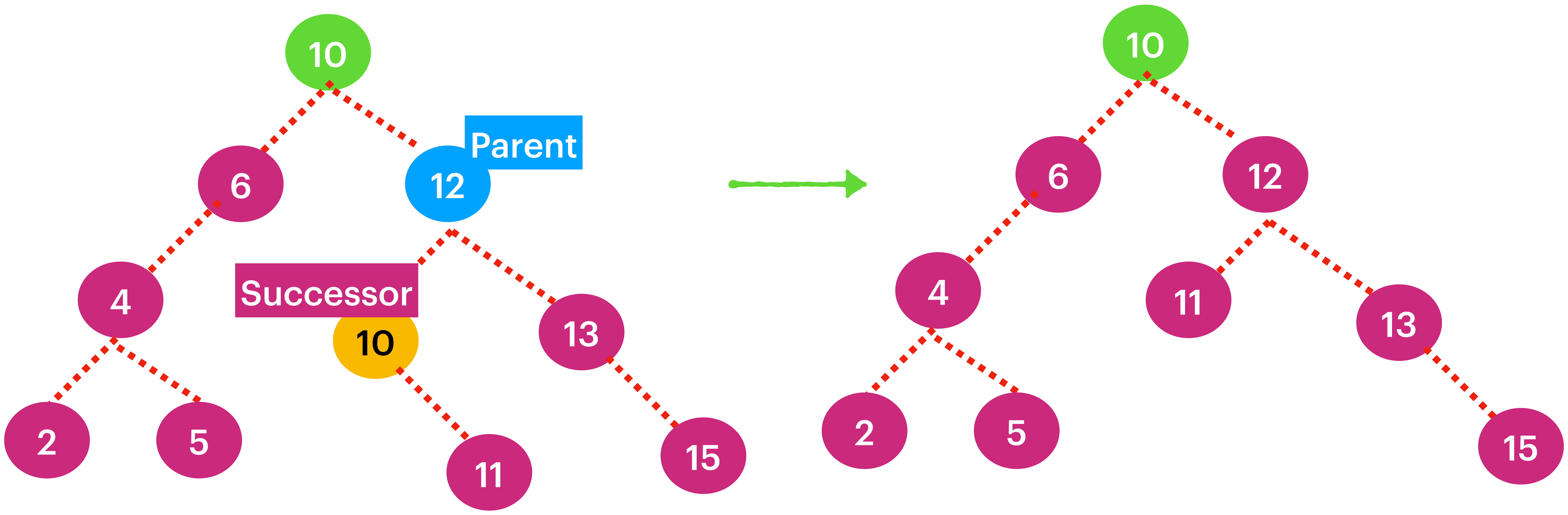
1. Get the max element from left subtree then replace. (Precedessor)
2. Get the min element from right subtree then replace. (Successor)

Lets go for 2nd possibility :

Successor => take element.right then pick least possible element.left

element.val = successor.val

Then remove successor => parent.left / right = successor.right



Time Complexity : O(H) ~ O(logn)

Space Complexity : O(1)