Department of
Computer Engineering

Data Structure & Algorithms

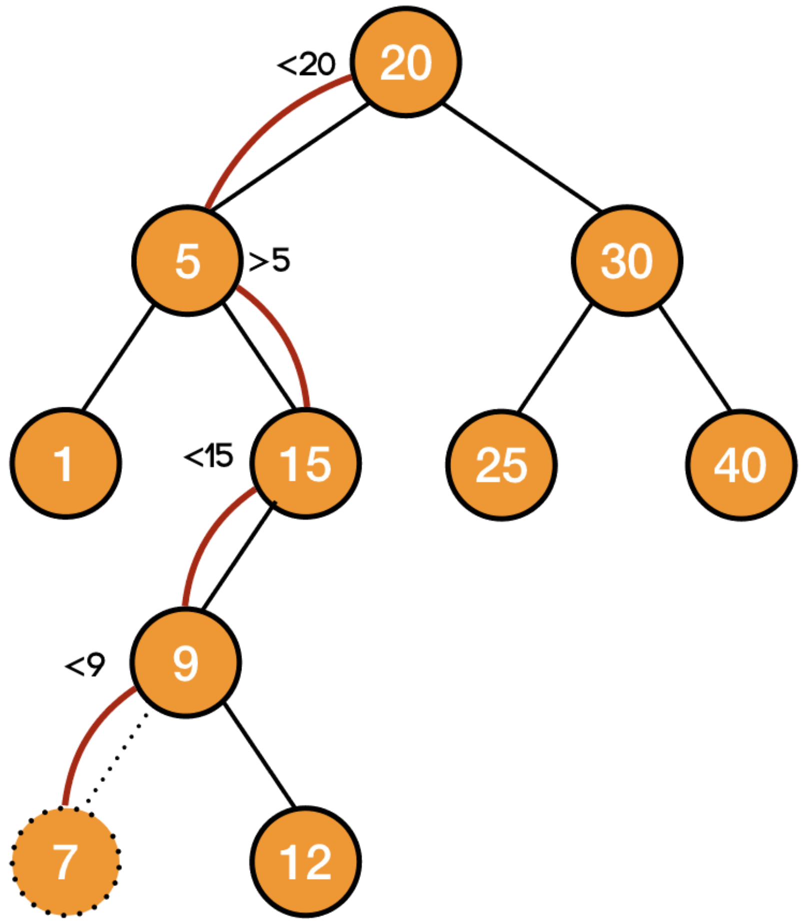
BST (Binary Search Tree)

Insertion and Deletion

Operation : Insert

- Insert a node z into a binary search tree T
- After insertion, T must remain a binary search tree

Example - Insert



INSERT 7

STEP 1: 7 IS SMALLER THAN 20: GO LEFT

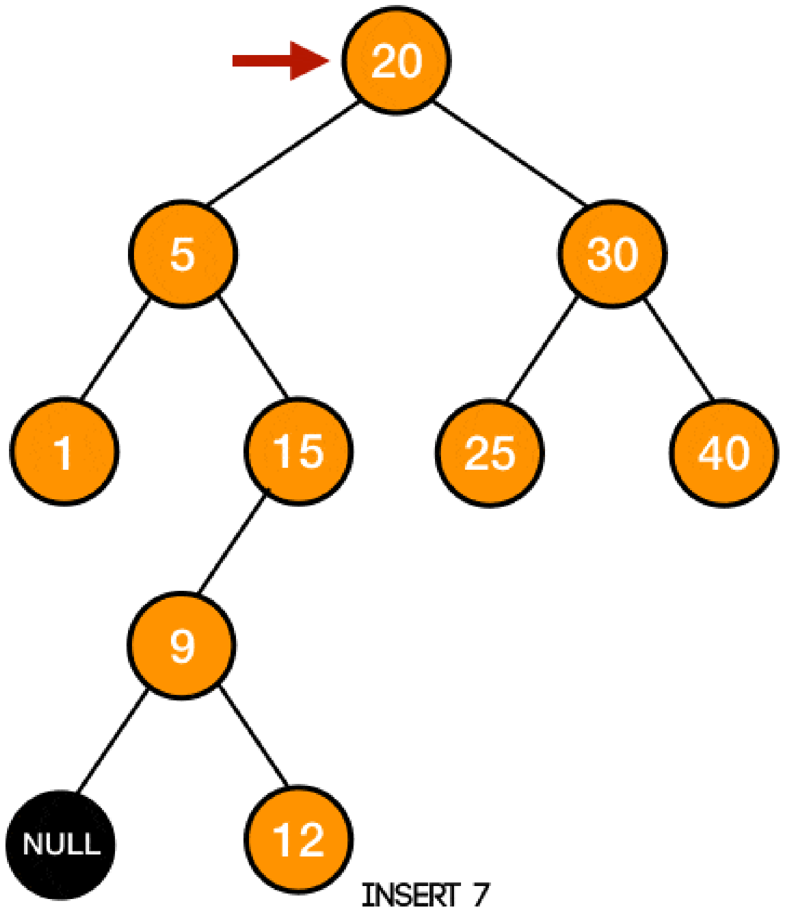
STEP 2: 7 IS GREATER THAN 5: GO RIGHT

STEP 3: 7 IS SMALLER THAN 15: GO LEFT

STEP 4: 7 IS SMALLER THAN 9: GO LEFT

STEP 5: NO LEFT CHILD: INSERT 7

Example – Insert



Insert – Alogrithm

```
Tree-Insert(T, z)
y = NIL
x = T.root
while x != NIL
    y = x
    if z.key < x.key
        x = x.left
    else x = x.right
z.p = y
if y == NIL
   T.root = z     // tree T was empty
elseif z.key < y.key
    y.left = z
else y.right = z
```

Insert – Code

```c
/* A utility function to insert
  a new node with given key in
* BST */
struct node* insert(struct node* node, int key)
{
    /* If the tree is empty, return a new node */
    if (node == NULL)
        return newNode(key);

    /* Otherwise, recur down the tree */
    if (key < node->key)
        node->left = insert(node->left, key);
    else if (key > node->key)
        node->right = insert(node->right, key);

    /* return the (unchanged) node pointer */
    return node;
}
```

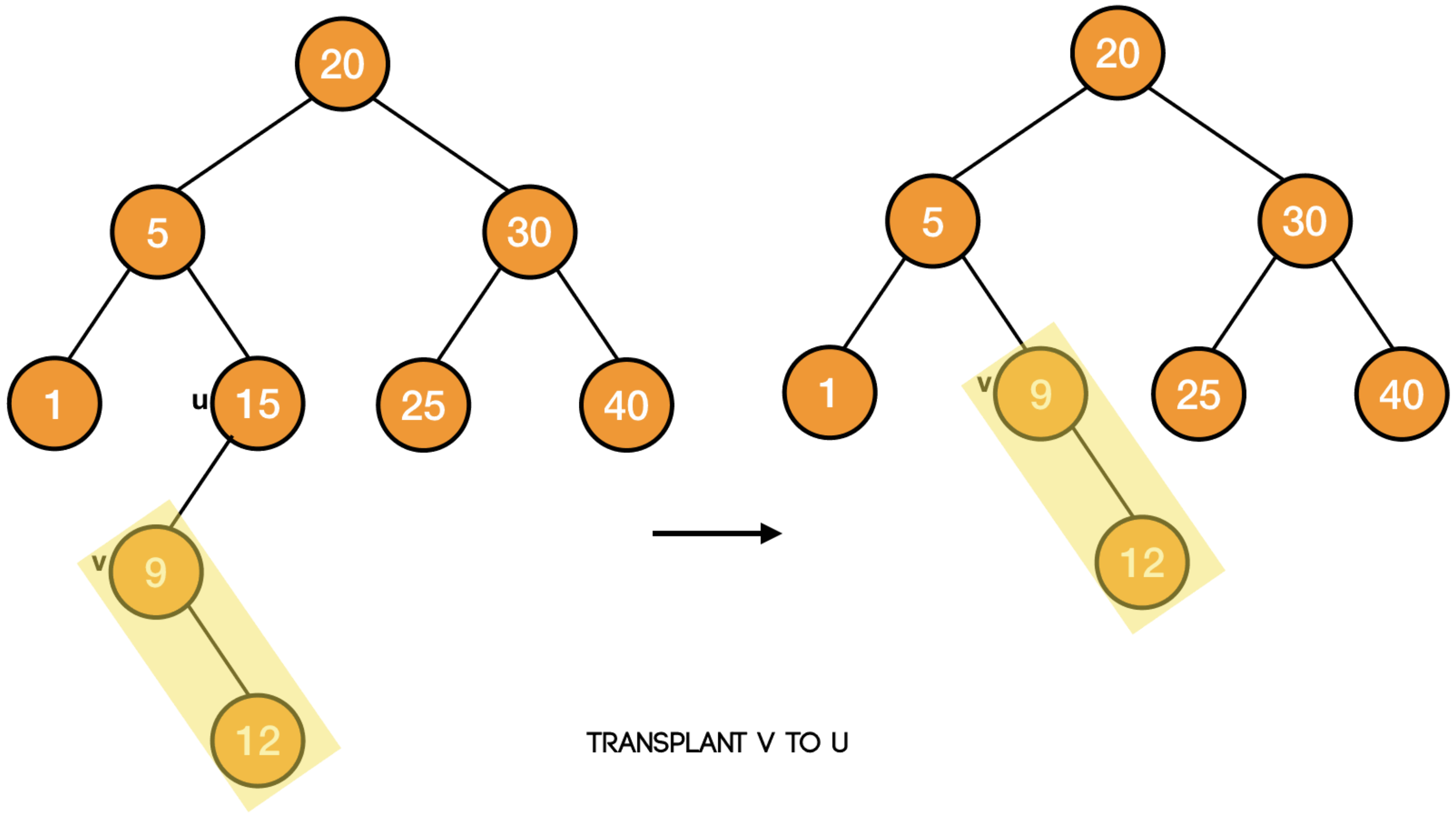
Insert – Time Complexity

- For each level of tree it takes constant time (according to code) → Total time will be:
  - Height of tree * Constant Time
  - Overall: $T(n) \in O(h)$ → $T(n) \in O(logn)$

Operation - Delete

- Delete a node z from a binary search tree T
- After delete, T must remain a binary search tree
- Cases:
  - Z has no child.
  - Z has one child.
  - Z has two child.

Example - Delete



TRANSPLANT V TO U

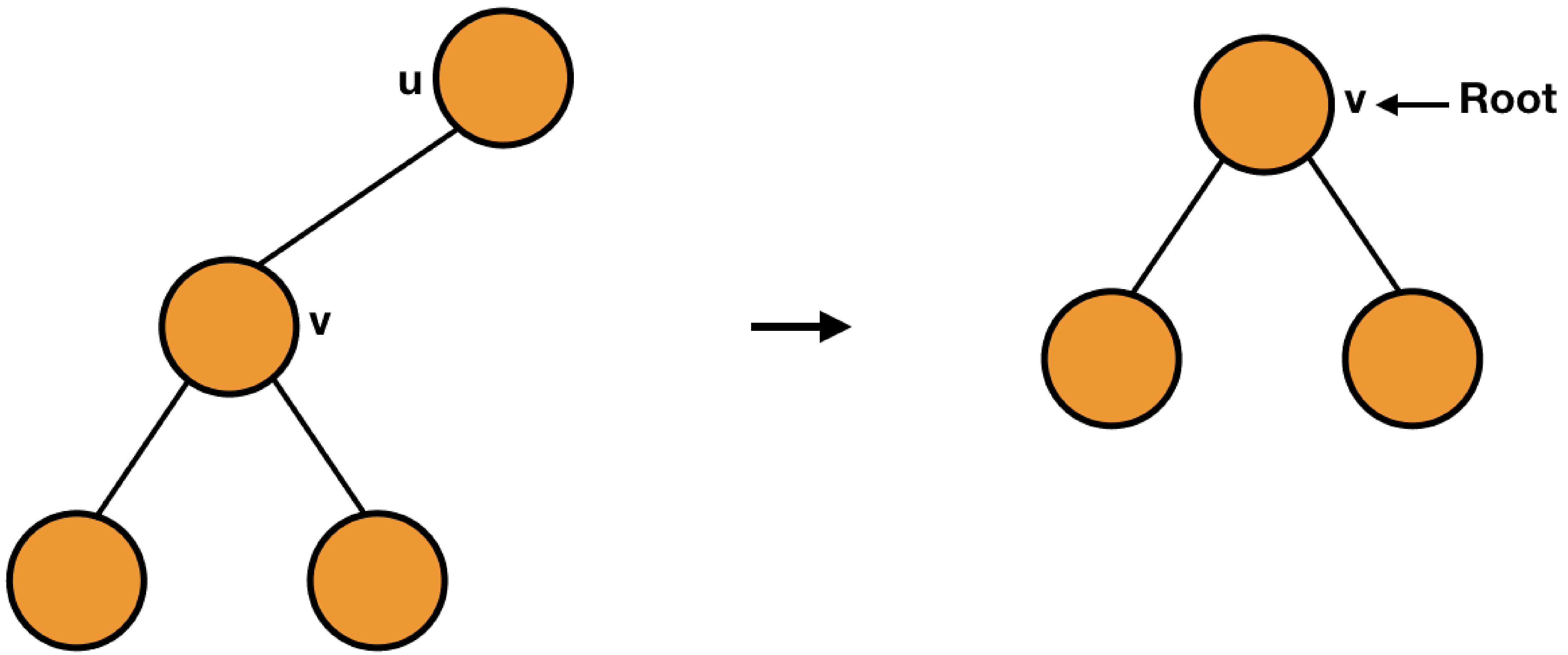
Transplant

- As we are going to use this technique in our delete procedure, so let's first write the code to transplant a subtree rooted at node v in place of the subtree rooted at node u.
- we want to place the subtree rooted at node v in place of the subtree rooted at node u. It means that we need to make v the child of the parent of u i.e., if u is the left child, then v will become the left child of u's parent. Similarly, if u is the right child, then v will become the right child of u's parent.

Transplant

- It is also possible that u doesn't have any parent i.e., u is the root of the tree T. In that case, we will simply make v as the root of the tree.

Example - Transplant



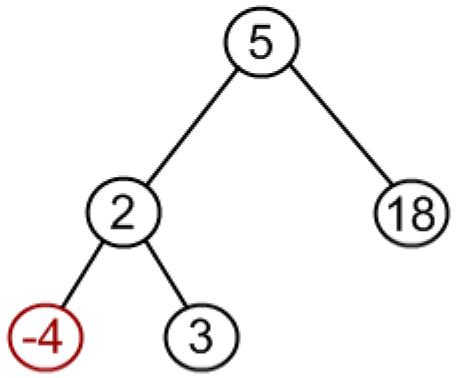
BST Delete – Transplant

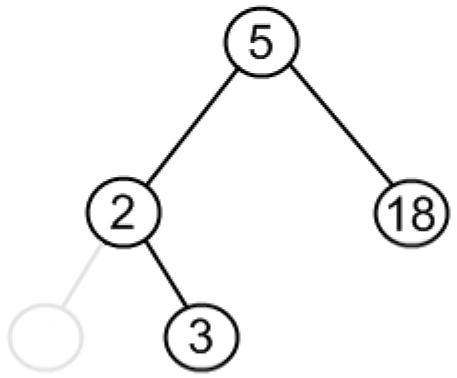
```
Transplant(T, u, v)
if u.p == NIL
    T.root = v
elseif u == u.p.left
    e.p.keft = v
else u.p.right = v
if v != NIL
    v.p = u.p
```

Tree Delete – case 1 (no child)

- Suppose the node to be deleted is a leaf, we can easily delete that node by pointing the parent of that node to NIL
- We can also say that we are transplanting the right or the left child (both are NIL) to the node to be deleted.

Tree Delete – case 1 (no child)

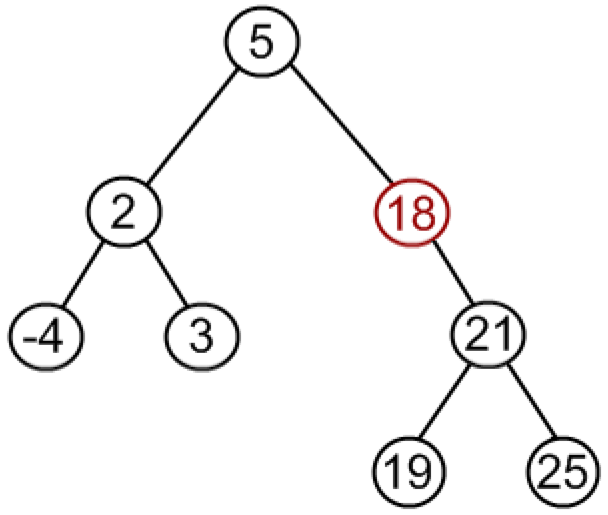




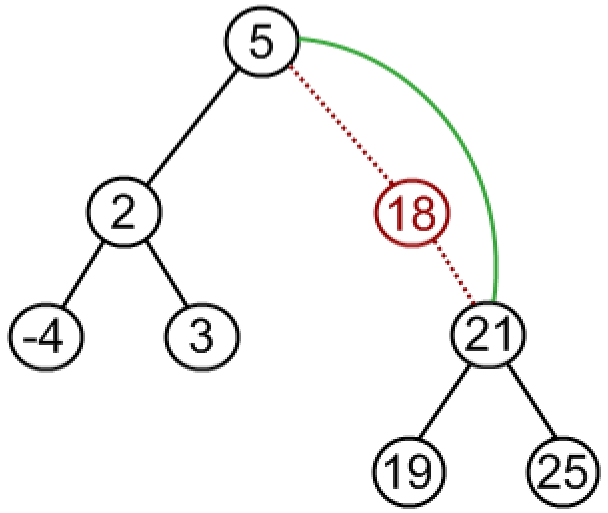
Tree Delete – case 2 (one child)

- We can also delete a node with only one child by transplanting its child to the node and it will not affect the property of the binary search tree.

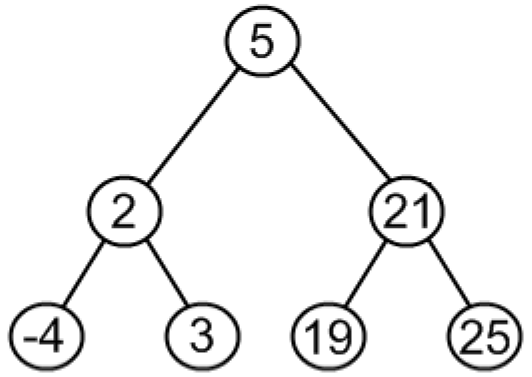
Tree Delete – case 2 (one child)



Tree Delete – case 2 (one child)



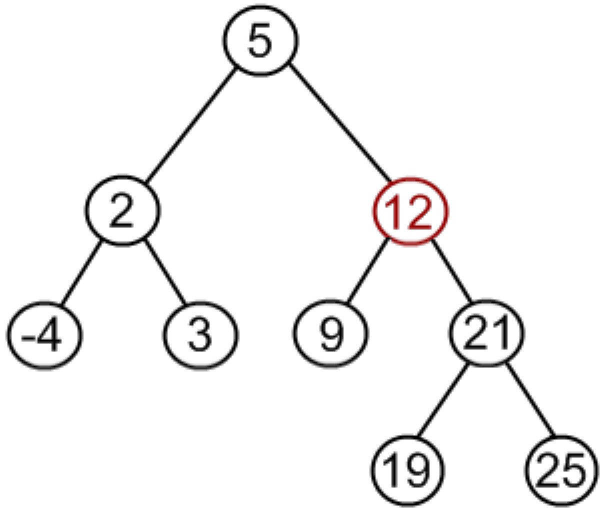
Tree Delete – case 2 (one child)



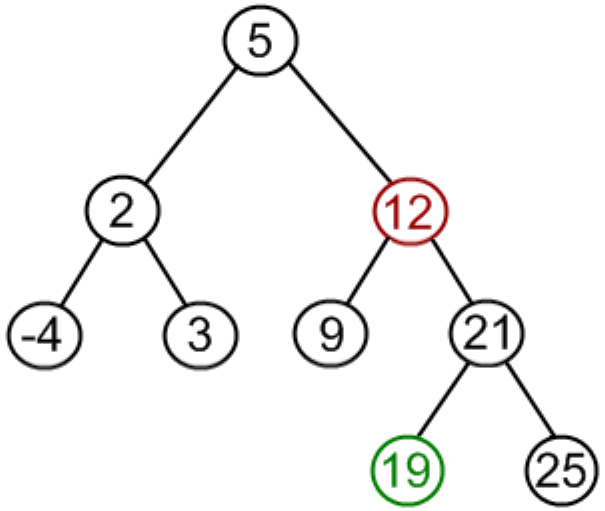
Tree Delete – case 3 (two children)

- But things will become a bit little complicated when the node to be deleted has both the children.
- In this case, we can find the smallest element of the right subtree of the node to be deleted (element with no left child in the right subtree) and replace its content with the node to be deleted.

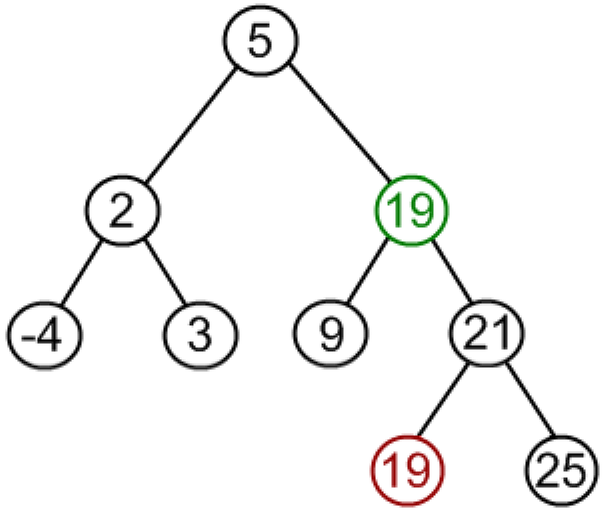
Tree Delete – case 3 (two children)



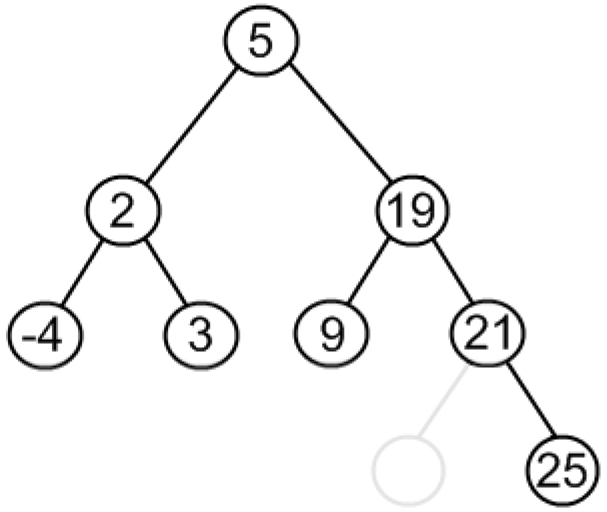
Tree Delete – case 3 (two children)



Tree Delete – case 3 (two children)



Tree Delete – case 3 (two children)

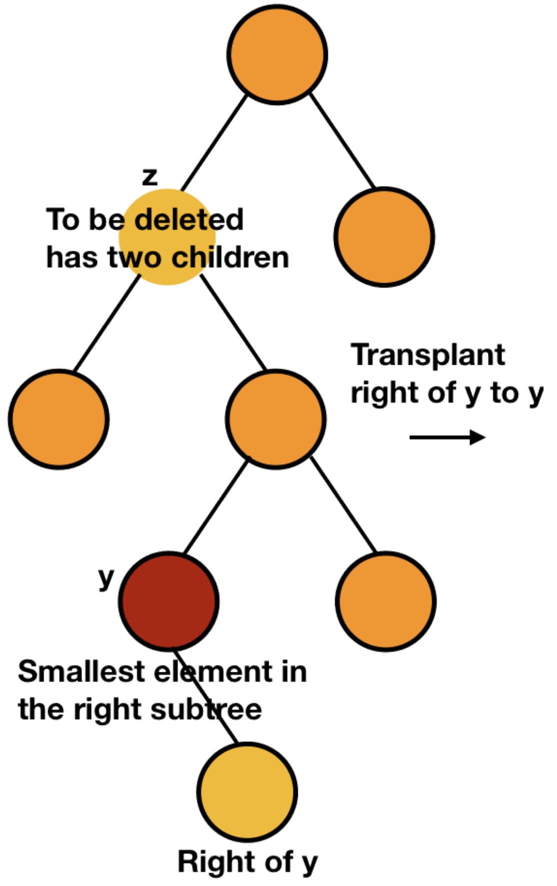


Tree Delete – case 3 (two children)

- Doing so is not going to affect the property of binary search tree because it is the smallest element of the right subtree, so all the elements in the right subtree are still greater than it. Also, all the elements in the left subtree were smaller than it because it was in the right subtree, so they are still smaller.

Smallest element of the right subtree

- The smallest element of the right subtree will have either have no child or one child because if it has left child, then it will not be the smallest element. So, we can delete this node easily as discussed in the first two cases.


z
To be deleted has two children
Transplant right of y to y
y
Smallest element in the right subtree
Right of y

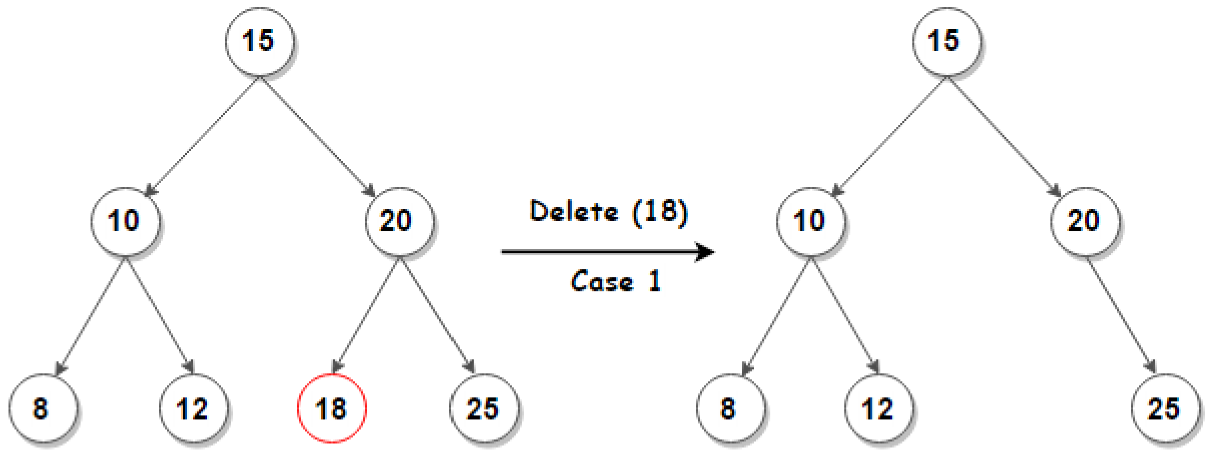
Tree Delete code

- Case 1 & 2:
  - we have to check the number of children of the node z. We will first check if the left child of the node z is NULL or not. If the left child is NULL, then either it has only one child (right one) or none. In both the cases, we can transplant its right child to it.
  - Else we will next check if the right child is NULL or not.

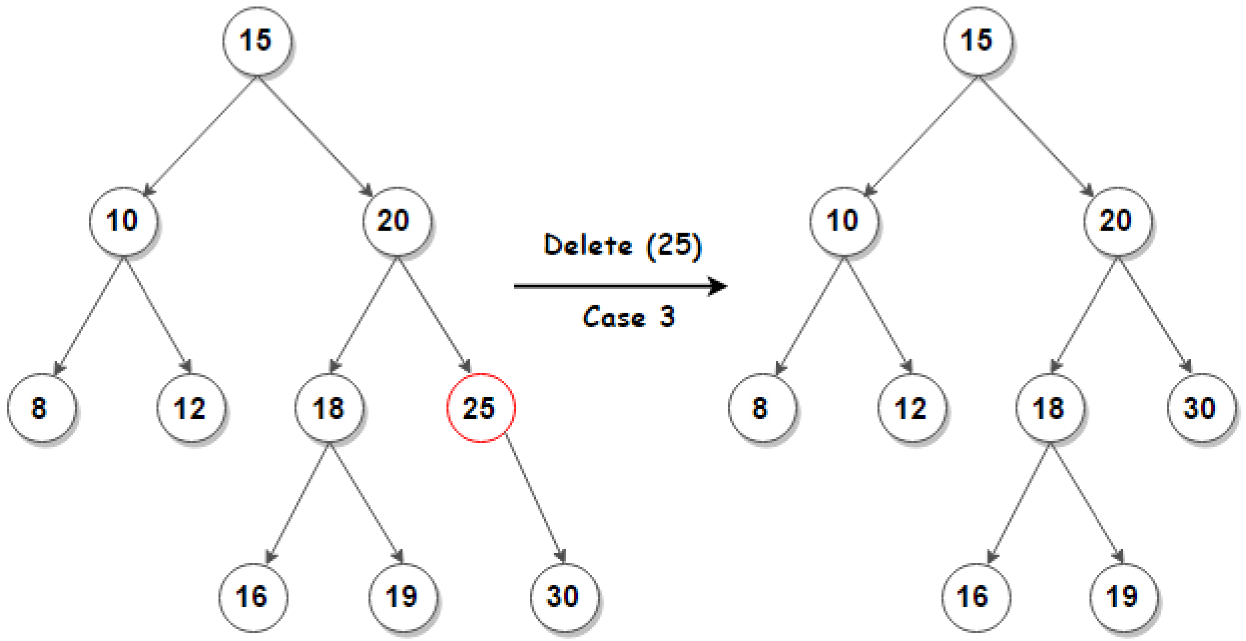
Tree Delete code

- Case 3:
    - If the node z has both children, we will find the minimum in the right subtree (y). Now, we have to put this minimum node (y) in the place of z.
        - transplant the right of y to y
        - take the right subtree of z and make it the right subtree of y.
        - transplant y to z.
        - change the left child of y to the left child of z. (y’s left child is NIL)

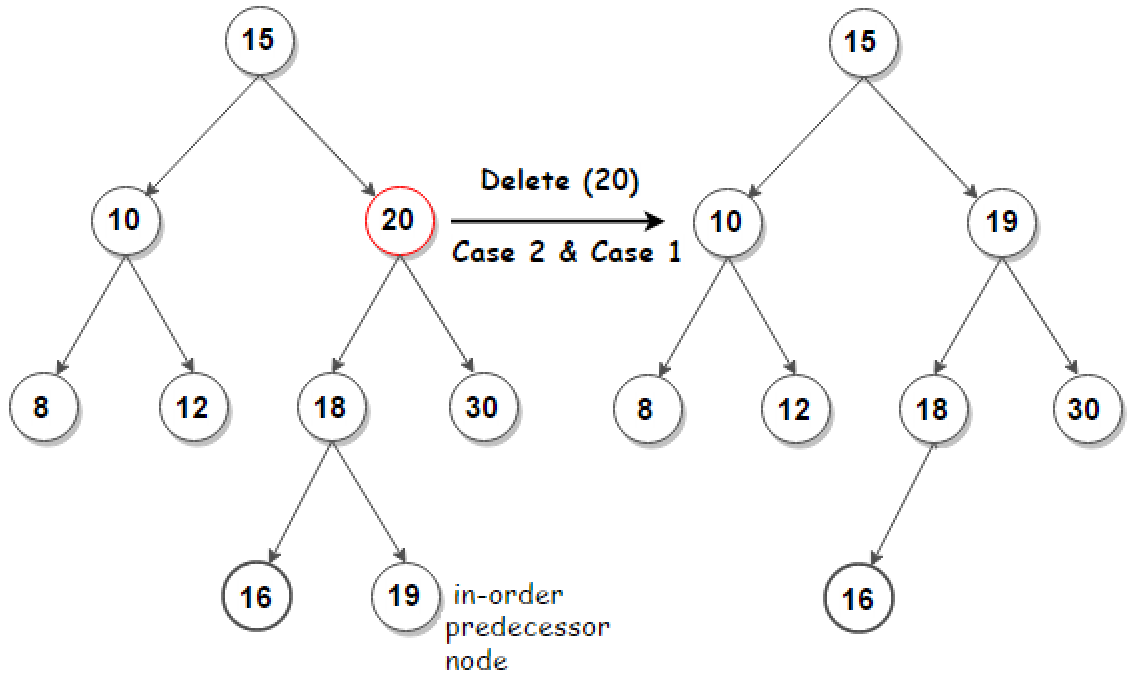
Tree Delete case 1



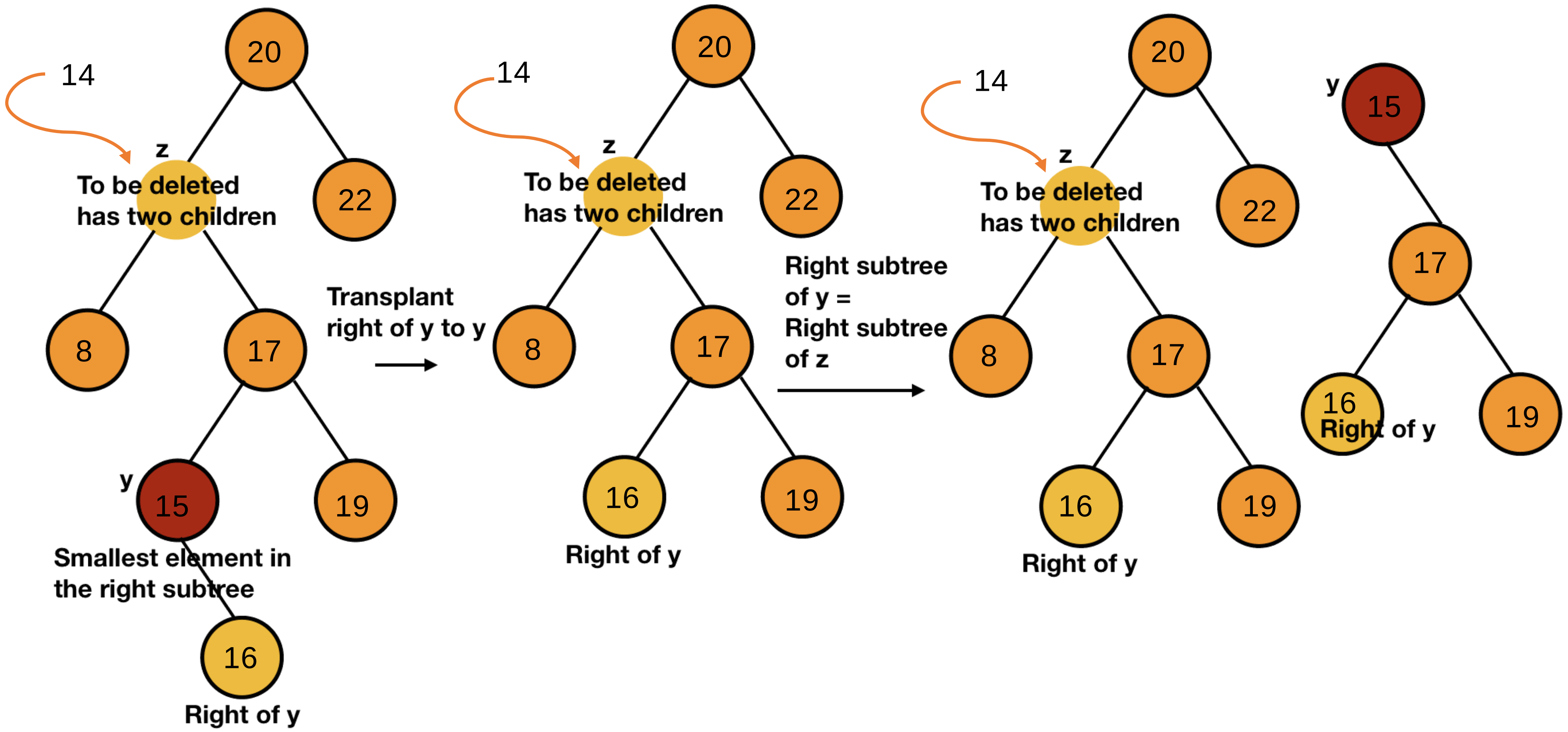
Tree Delete case 2



Tree Delete case 3

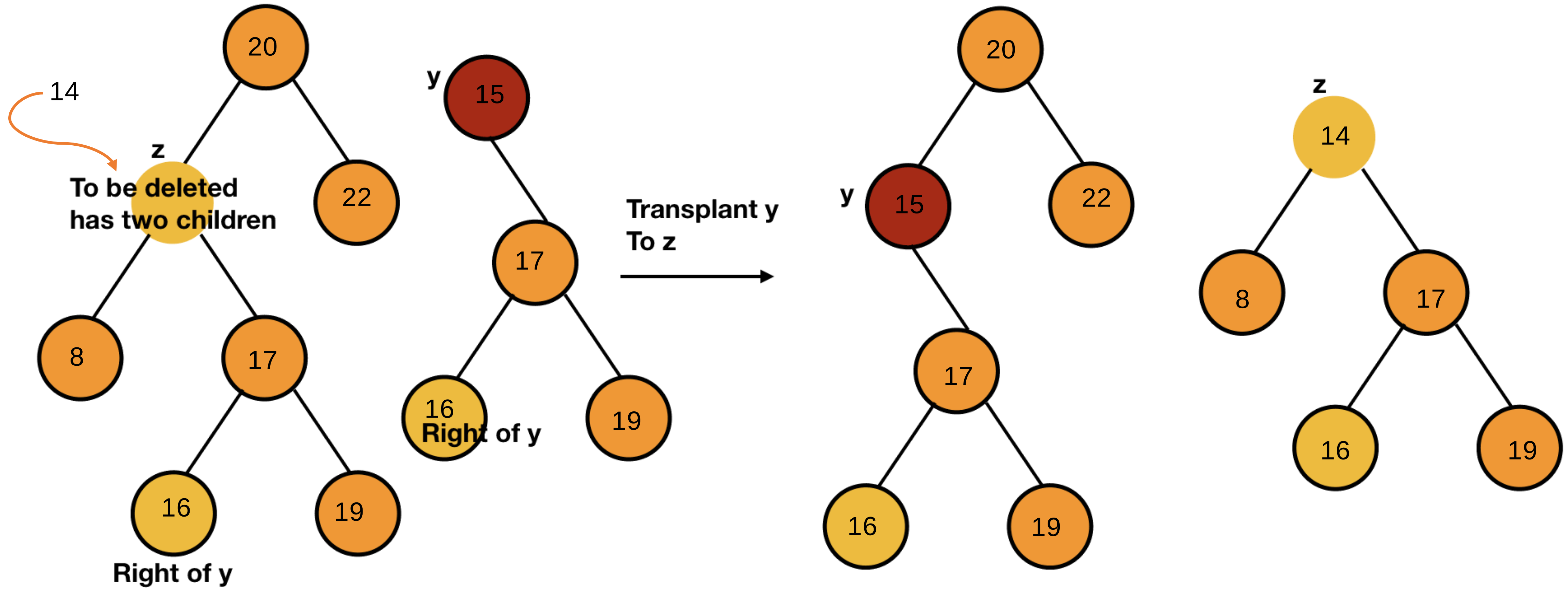


Case 3 code – Example 1

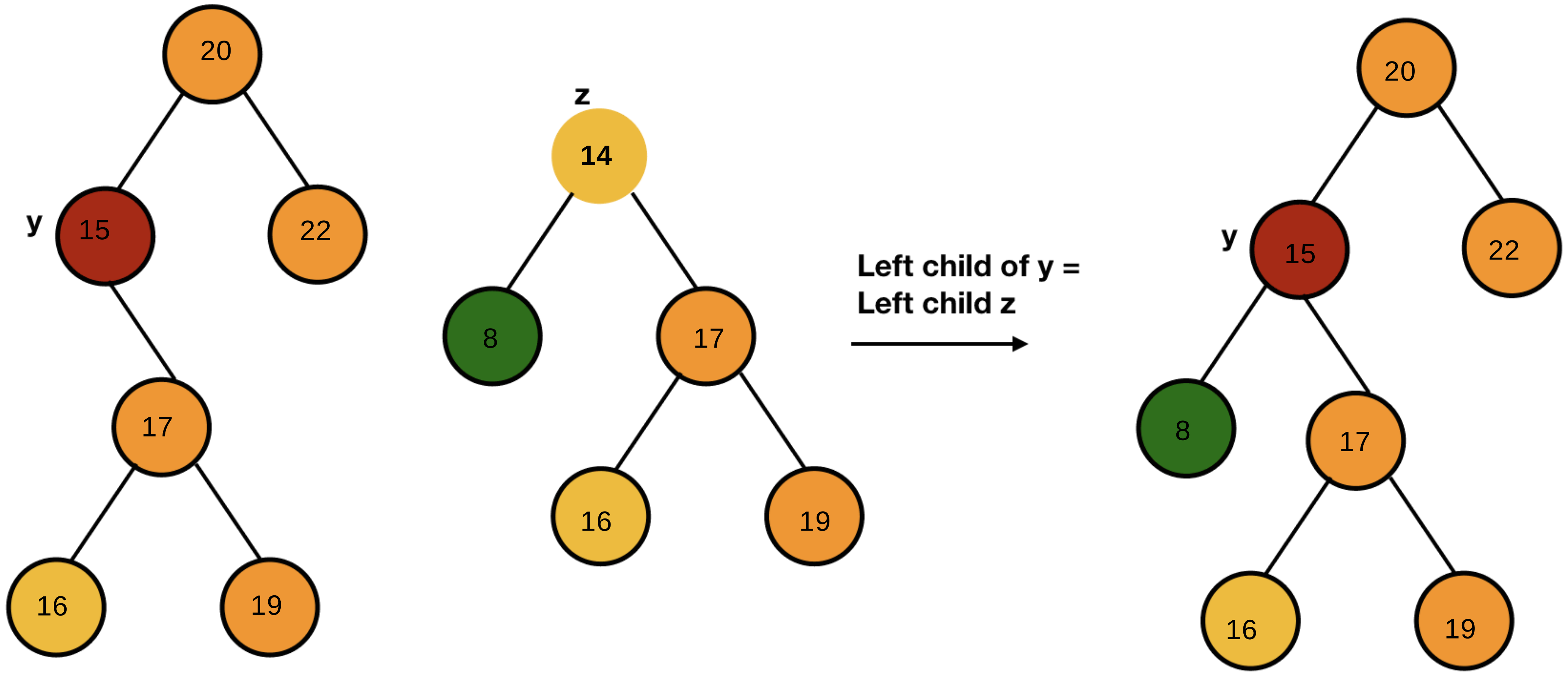


14
20
z
To be deleted has two children
22
8
17
y
15
19
Smallest element in the right subtree
16
Right of y
Transplant right of y to y
14
20
z
To be deleted has two children
22
8
17
16
19
Right of y
Right subtree of y = Right subtree of z
14
20
z
To be deleted has two children
22
8
17
16
19
Right of y
y
15
17
16
Right of y
19

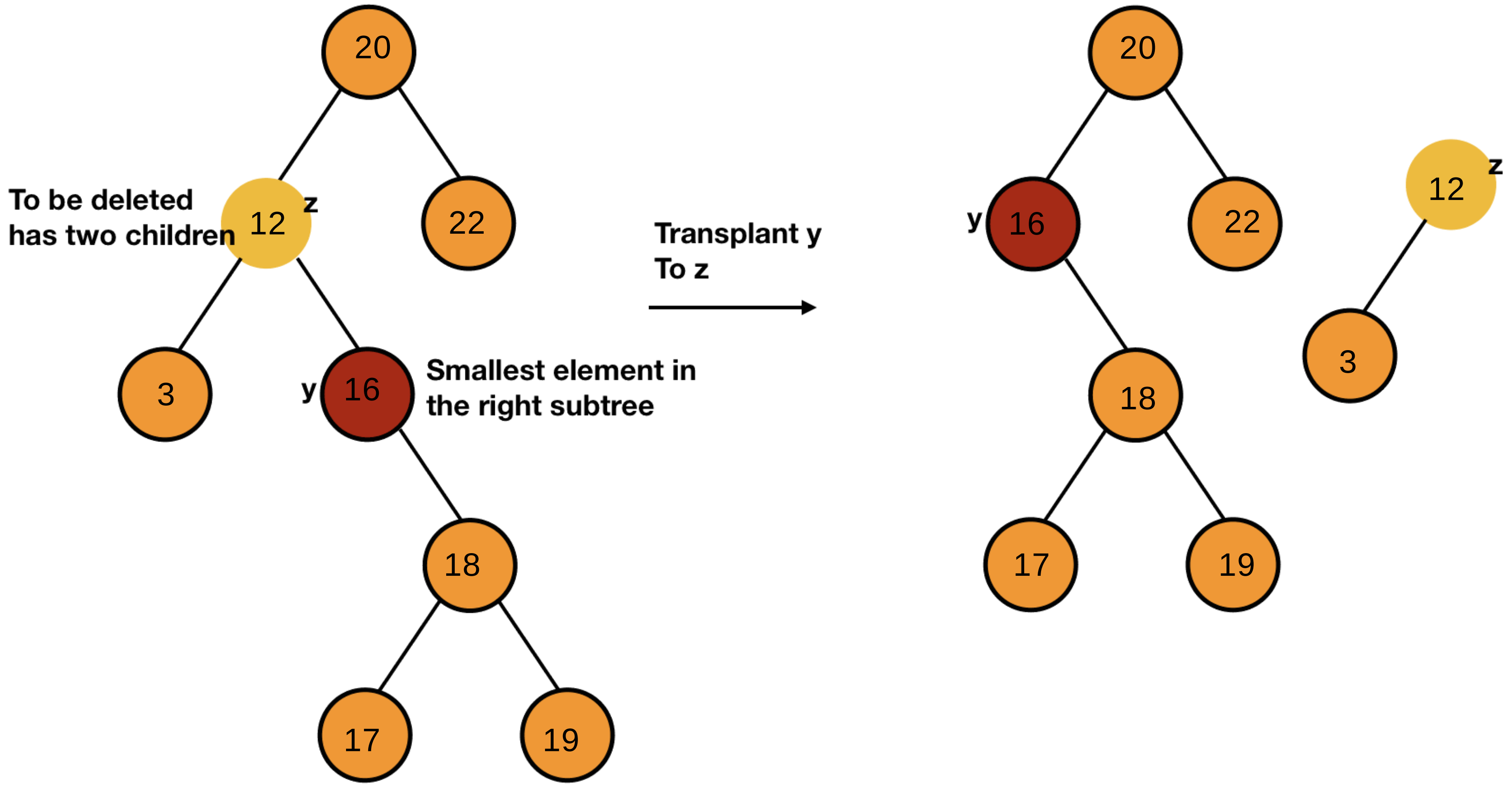
Case 3 code – Example 1



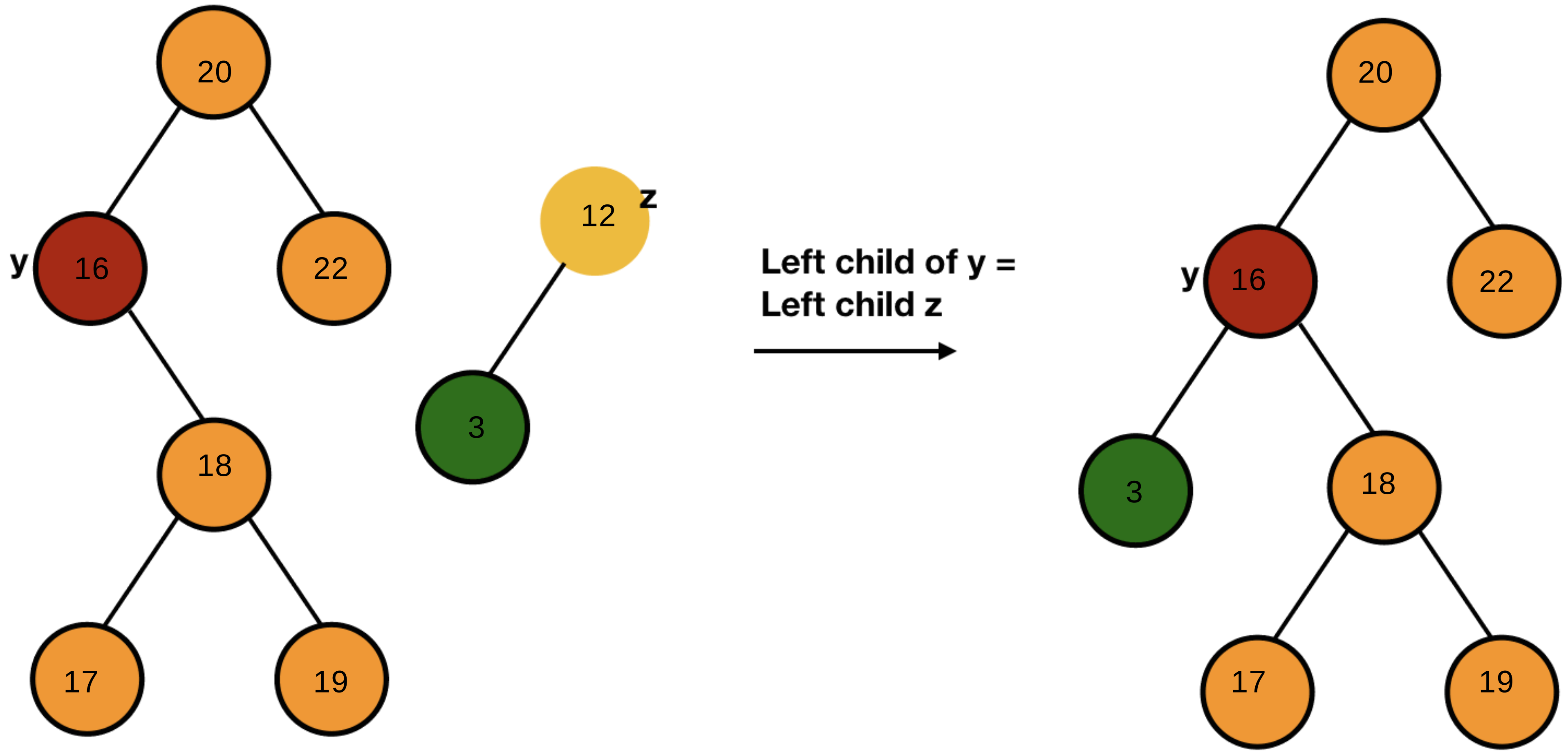
Case 3 code – Example 1



Case 3 code – Example 2



Case 3 code – Example 2



Delete - Algorithm

```
DELETE(T, z)
    if z.left == NIL
        TRANSPLANT(T, z, z.right)
    elseif z.right == NIL
        TRANSPLANT(T, z, z.left)
    else
        y = MINIMUM(z.right) //minimum element in right subtree
        if y.parent != z //z is not direct child
            TRANSPLANT(T, y, y.right)
            y.right = z.right
            y.right.parent = y
        TRANSPLANT(T, z, y)
        y.left = z.left
        y.left.parent = y
```

Delete – Code

```
// funnction to delete a node
struct node* delete(struct node *root, int x)
{
    //searching for the item to be deleted
    if(root==NULL)
        return NULL;
    if (x>root->data)
        root->right_child = delete(root->right_child, x);
    else if(x<root->data)
        root->left_child = delete(root->left_child, x);
    else
    {
        //No Children
        if(root->left_child==NULL && root->right_child==NULL)
        {
            free(root);
            return NULL;
        }
```

Delete – Code

```
        //One Child
        else if(root->left_child==NULL || root->right_child==NULL)
        {
            struct node *temp;
            if(root->left_child==NULL)
                temp = root->right_child;
            else
                temp = root->left_child;
            free(root);
            return temp;
        }

        //Two Children
        else
        {
            struct node *temp = find_minimum(root->right_child);
            root->data = temp->data;
            root->right_child = delete(root->right_child, temp->data);
        }
    }
    return root;
}
```

BST Delete

- The previous implementation of BST Delete considered replacing the deleted node z with its successor
- Another similar solution is to replace the deleted node z with its predecessor

Delete – Time Complexity

- For each level of tree it takes constant time (according to code) → Total time will be:
  - Height of tree * Constant Time
  - Overall: $T(n) \in O(h) \rightarrow T(n) \in O(logn)$