Department of
Computer Engineering

Data Structure & Algorithms

Red Black Trees Insertion

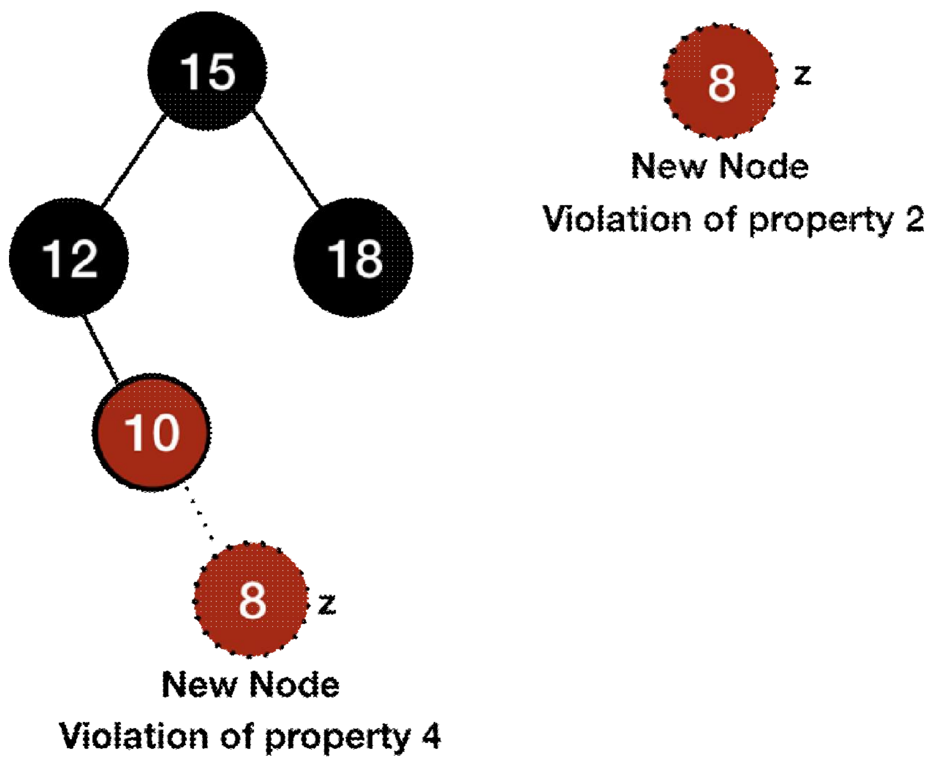
Properties of Red-Black Trees – Review

1. Every node is colored either **red** or **black**.
2. Root of the tree is **black**.
3. All leaves are **black**.
4. Both children of a red node are black i.e., there can't be consecutive **red** nodes.
5. All the simple paths from a node to descendant leaves contain the same number of **black** nodes.

Insertion

- We insert a new node to a **red-black tree** in a similar way as we do in a normal **binary search tree**. We just call a function at last to fix any kind of violations that could have occurred in the process of insertion.
- We set the color of any newly inserted node to **red**.
- Doing so can violate the property 4 of **red-black trees** which we will fix after the insertion process as stated above.
- There can be a violation of second property, but it can be easily fixed by coloring the root **black**. Also, there can't be any other violations.

Insertion (cont.)



Insertion (cont.)

- Why don’t we set the color of any new node to **black**?

  Think of a case when the newly inserted node is **black**. This would affect the black height and fixing that would be difficult.

Code for Insertion

```
INSERT(T, n)
  y = T.NIL
  temp = T.root

  while temp != T.NIL
      y = temp
      if n.data < temp.data
          temp = temp.left
      else
          temp = temp.right
  n.parent = y
  if y == T.NIL
      T.root = n
  else if n.data < y.data
      y.left = n
  else
      y.right = n

  n.left = T.NIL
  n.right = T.NIL
  n.color = RED
  INSERT_FIXUP(T, n)
```

Insertion Code

- Here, we have used $T.NIL$ instead of $NULL$ unlike we do with normal binary search tree.
- Also, those $T.NIL$ are the leaves and they all are **black**, so there won't be a violation of property 3.
- In the last few lines, we are making the left and right of the new node $T.NIL$ and also making it **red**. At last, we are calling the fixup function to fix the violations of the **red-black** properties. Rest of the code is similar to a normal binary search tree.
- Let's focus on the function that fixes the violations.

Insertion Fixup

- Which of the red-black properties might be violated upon the Insertion call?

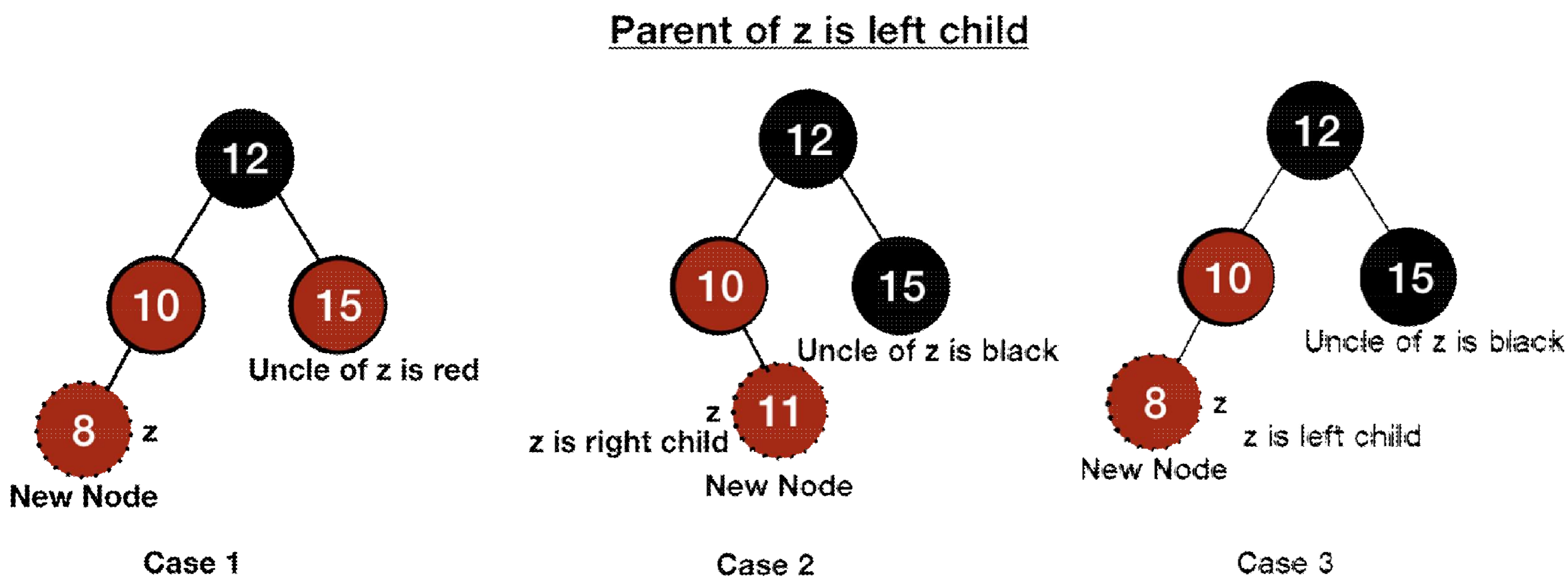
  - The property 2 might be violated. We will fix this violation by coloring the **root** node to black (very easy).

  - The property 4 will be violated when the parent of the inserted node is **red**. So, we will fix the violations if the parent of the new node is red.

Insertion Fixup (cont.)

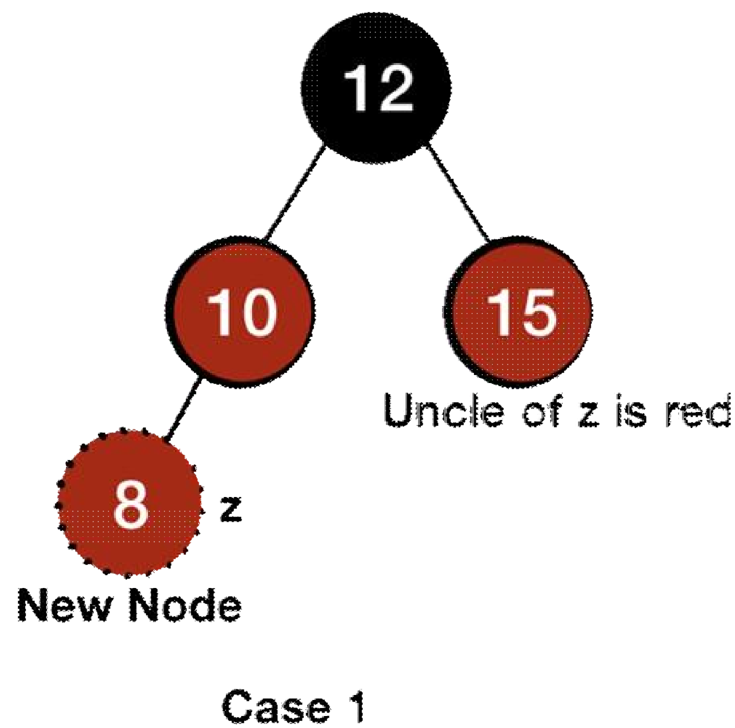
- There can be six cases if the 4th property was violated.
- Case 1, 2, 3:
  - the parent of node z (AKA the new node) is a left child of its parent.
- Case 4, 5, 6:
  - the parent of node z is a right child of its parent. (Symmetric to previous cases)

Insertion Fixup – Parent of Z is the left child

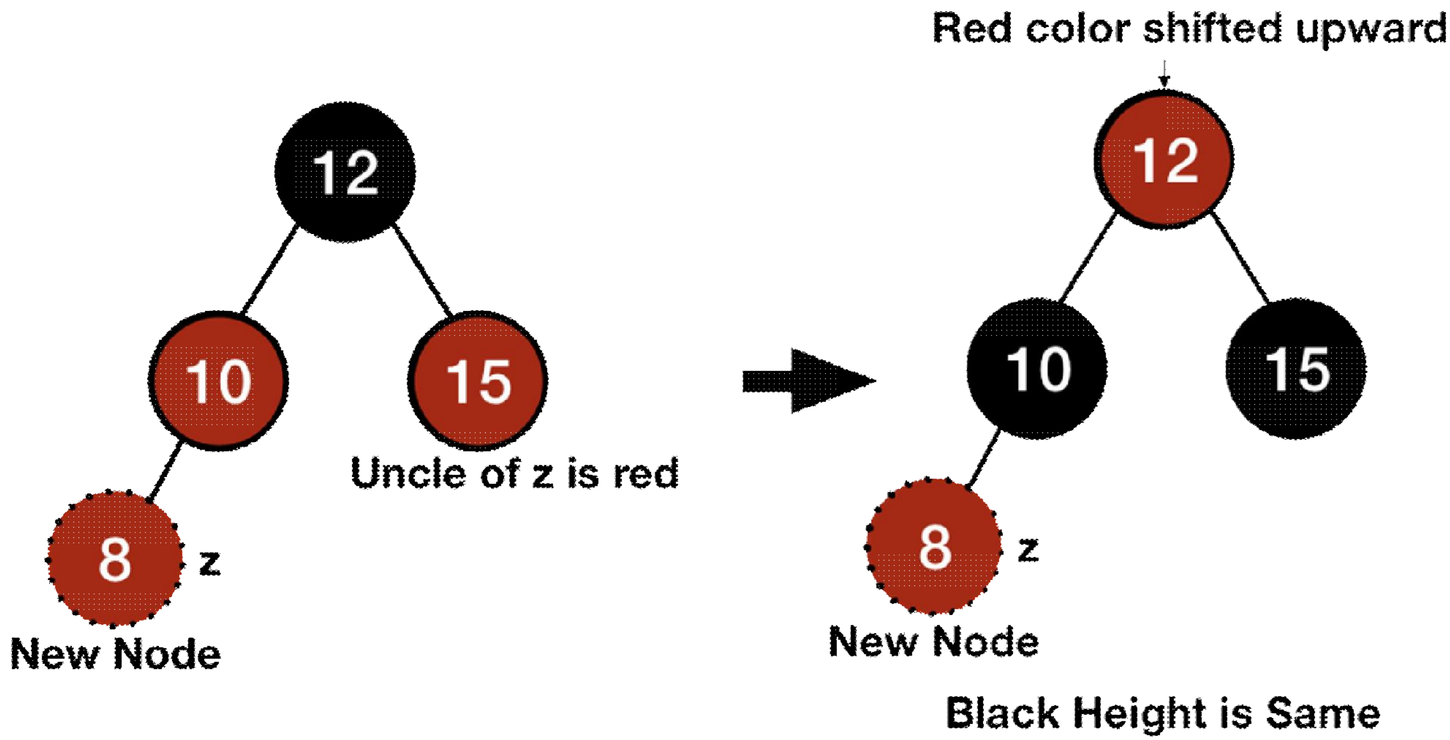


Insertion Fixup – Case 1

- The first case is when the uncle of z is also **red**.
- In this case, we will shift the **red** color upward until there is no violation.
- Otherwise, if it reaches the root, we can just color it **black** without any consequences.



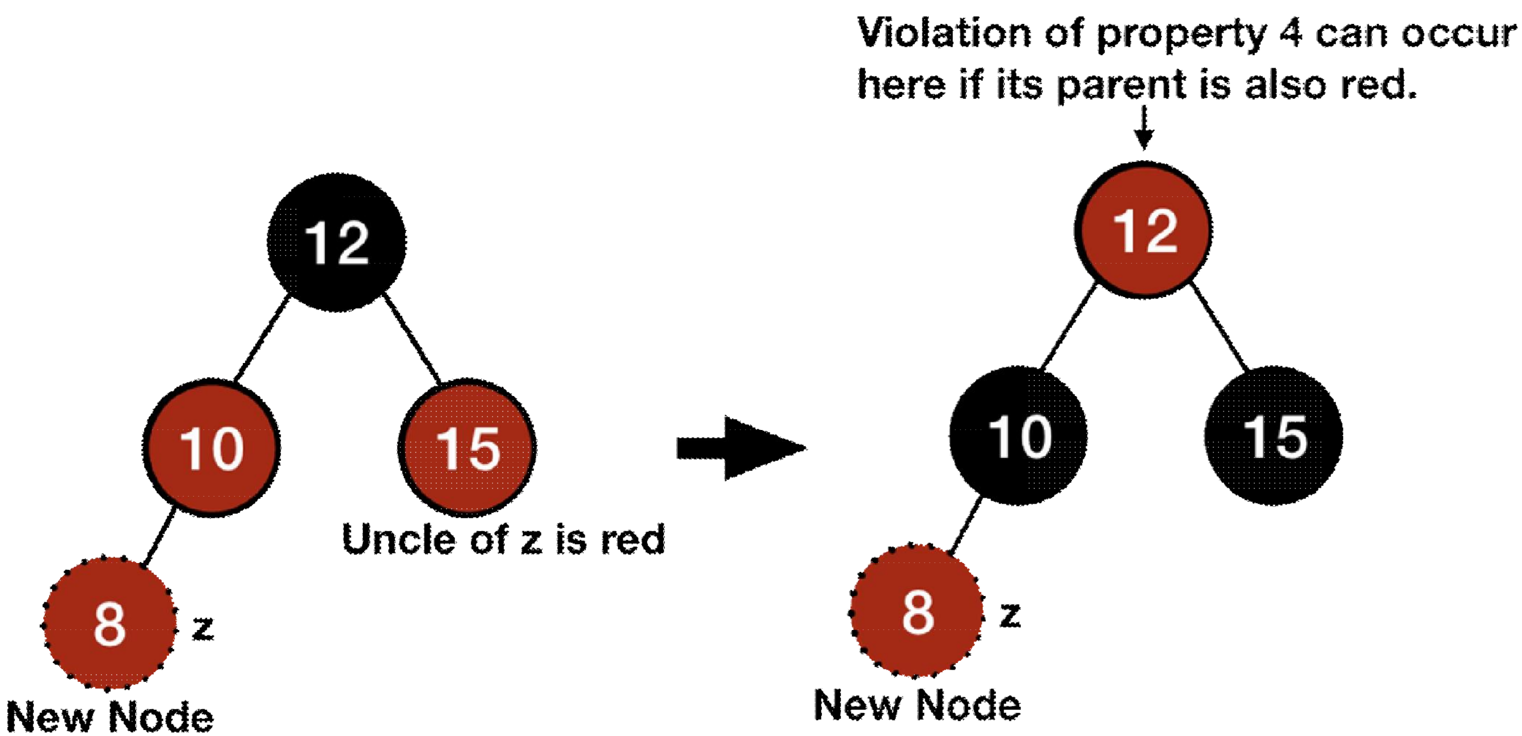
Insertion Fixup – Case 1 (cont.)



Insertion Fixup – Case 1 (cont.)

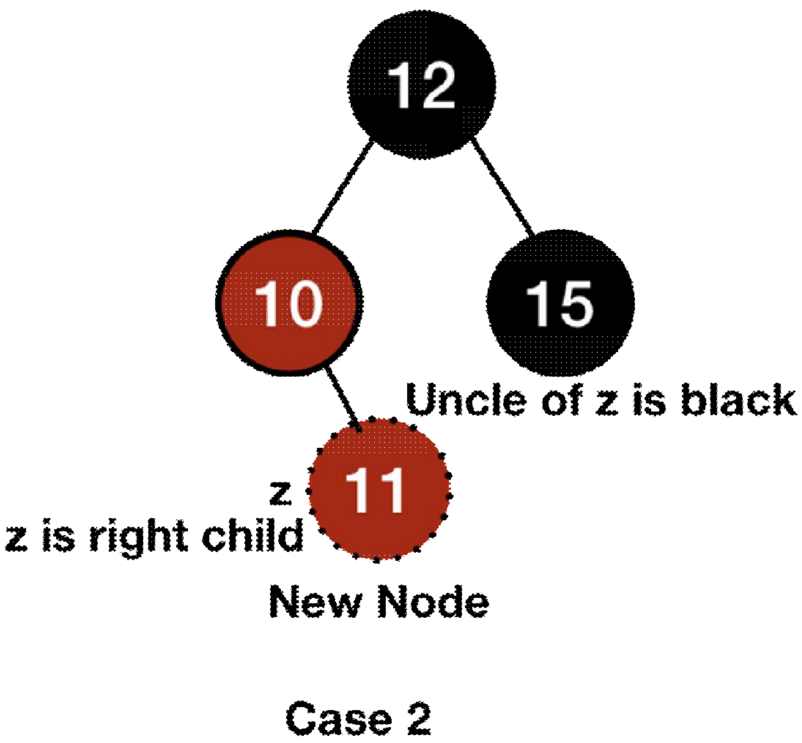
- To solve this violation we set the color of both the parent and the uncle of z **black** and its grandparent **red**. In this way, the **black** height of any node won't be affected and we can successfully shift the **red** color upward.
- However, coloring the grandparent of z **red** might cause violation of the 4th property. So, we will do the fixup again on that node.

Insertion Fixup – Case 1 (cont.)



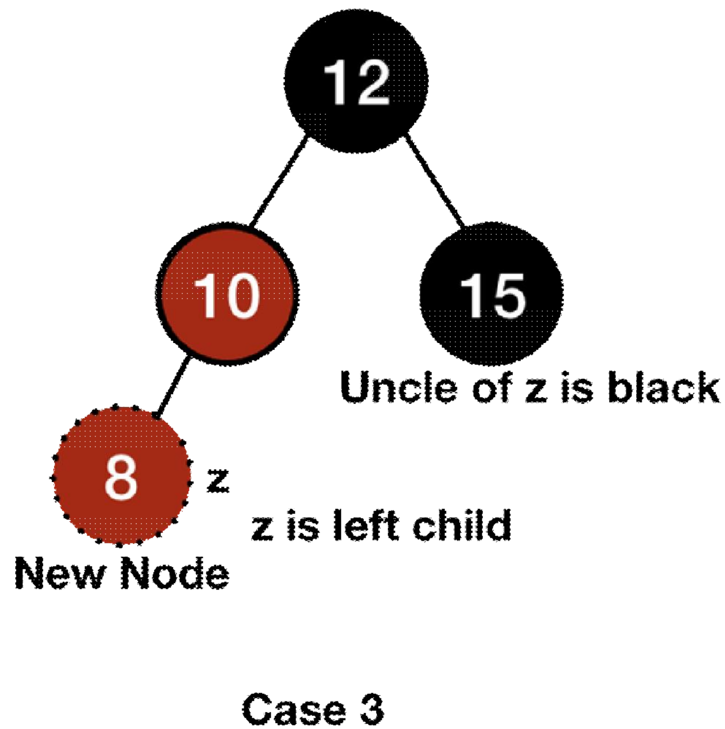
Insertion Fixup – Case 2

- In the second case, the uncle of the node z is **black** and the node z is the right child.



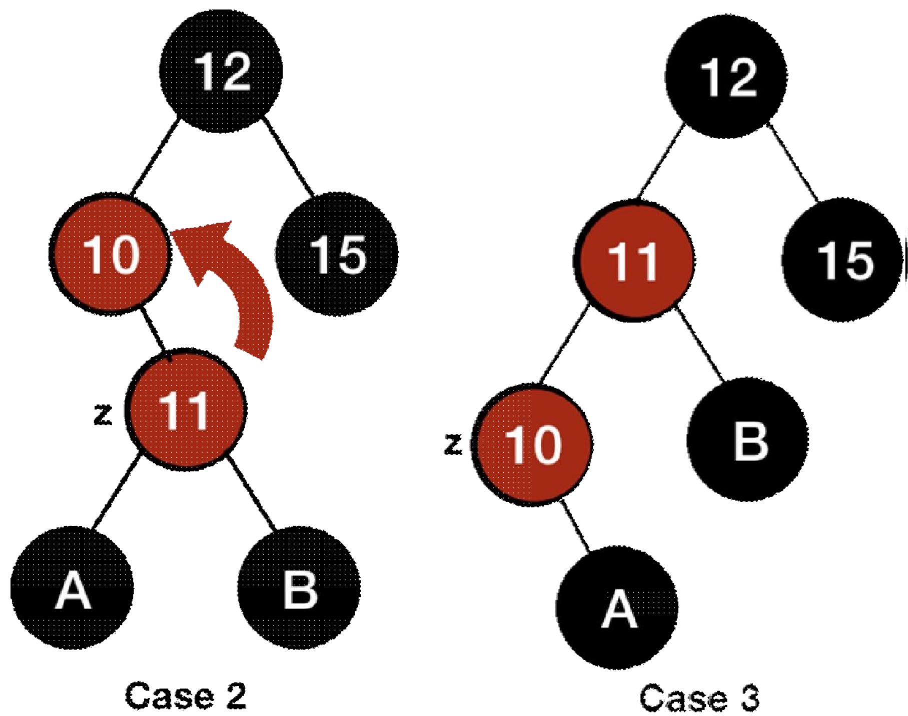
Insertion Fixup – Case 3

- In the third case, the uncle of the node z is **black** and the node z is the left child.



Insertion Fixup – Case 2 & 3

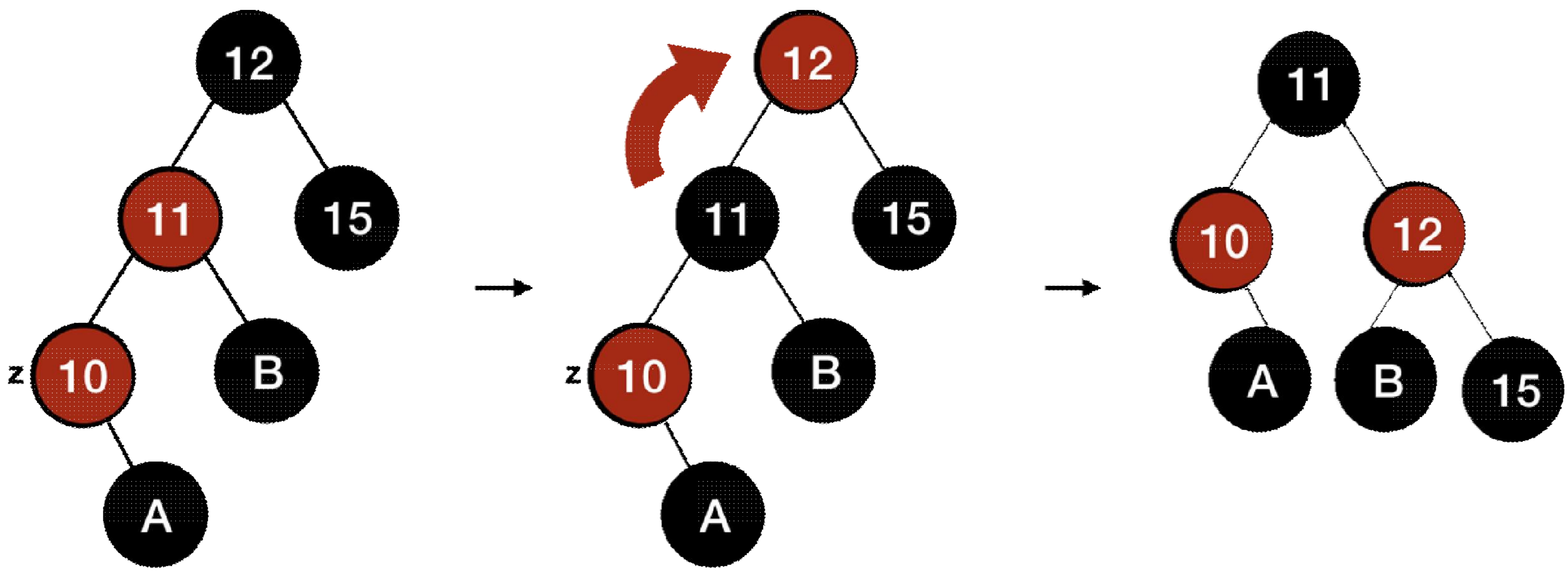
- We can transform the **second case** into the **third** one by performing the left rotation on the parent of the node z.
- Since both z and its parent are **red**, so rotation won't affect the **black height**.



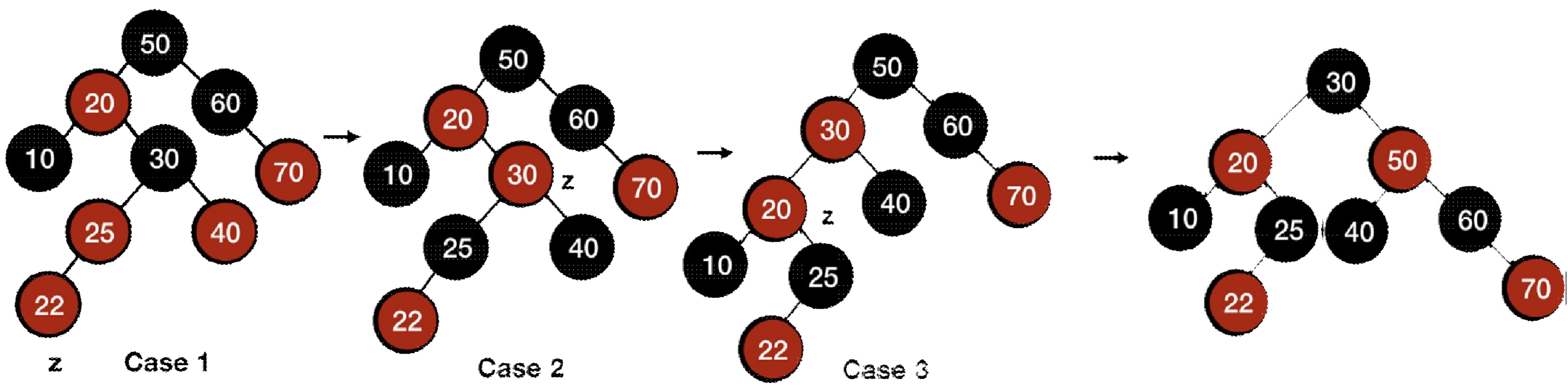
Insertion Fixup – Case 3

- In case 3, we first color the parent of the node z **black** and its grandparent **red** and then do a right rotation on the grandparent of the node z.
- This fixes the violation of properties completely.

Insertion Fixup – Case 3 (cont.)



Insertion Fixup – Example



Insertion Fixup

- Similarly, there will be three cases when the parent of z will be the right child but those cases will be symmetric to the above cases only with <u>left and right exchanged</u>.

Code for Fixup

```
INSERT_FIXUP(T, z)
  while z.parent.color == red

      if z.parent == z.parent.parent.left //z.parent is left child
          y = z.parent.parent.right //uncle of z

          if y.color == red //case 1
              z.parent.color = black
              y.color = black
              z.parent.parent.color = red
              z = z.parent.parent

          else //case 2 or 3
              if z == z.parent.right //case 2
                  z = z.parent //marked z.parent as new z
                  LEFT_ROTATE(T, z) //rotated parent of original z

              //case 3
              z.parent.color = black // made parent black
              z.parent.parent.color = red // made grandparent red
              RIGHT_ROTATE(T, z.parent.parent) // right rotation on grandparent

      else // z.parent is right child
          code will be symmetric //DIY (Do It Yourself!)
  T.root.color = black
```

RBT Code in C

```c
enum COLOR {Red, Black};

typedef struct tree_node {
    int data;
    struct tree_node *right;
    struct tree_node *left;
    struct tree_node *parent;
    enum COLOR color;
} tree_node;

typedef struct red_black_tree {
    tree_node *root;
    tree_node *NIL;
} red_black_tree;
```

RBT Code in C – New Node

```
tree_node* new_tree_node(int data) {
    tree_node* n = malloc(sizeof(tree_node));
    n->left = NULL;
    n->right = NULL;
    n->parent = NULL;
    n->data = data;
    n->color = Red;

    return n;
}

red_black_tree* new_red_black_tree() {
    red_black_tree *t = malloc(sizeof(red_black_tree));
    tree_node *nil_node = malloc(sizeof(tree_node));
    nil_node->left = NULL;
    nil_node->right = NULL;
    nil_node->parent = NULL;
    nil_node->color = Black;
    nil_node->data = 0;
    t->NIL = nil_node;
    t->root = t->NIL;

    return t;
}
```

RBT Code in C – Left Rotation

```c
void left_rotate(red_black_tree *t, tree_node *x) {
    tree_node *y = x->right;
    x->right = y->left;
    if(y->left != t->NIL) {
        y->left->parent = x;
    }
    y->parent = x->parent;
    if(x->parent == t->NIL) { //x is root
        t->root = y;
    }
    else if(x == x->parent->left) { //x is left child
        x->parent->left = y;
    }
    else { //x is right child
        x->parent->right = y;
    }
    y->left = x;
    x->parent = y;
}
```

RBT Code in C – Right Rotation

```c
void right_rotate(red_black_tree *t, tree_node *x) {
    tree_node *y = x->left;
    x->left = y->right;
    if(y->right != t->NIL) {
        y->right->parent = x;
    }
    y->parent = x->parent;
    if(x->parent == t->NIL) { //x is root
        t->root = y;
    }
    else if(x == x->parent->right) { //x is left child
        x->parent->right = y;
    }
    else { //x is right child
        x->parent->left = y;
    }
    y->right = x;
    x->parent = y;
}
```

RBT Code in C – Insertion Fixup

```c
void insertion_fixup(red_black_tree *t, tree_node *z) {
    while(z->parent->color == Red) {
        if(z->parent == z->parent->parent->left) { //z.parent is the left child

        tree_node *y = z->parent->parent->right; //uncle of z

        if(y->color == Red) { //case 1
            z->parent->color = Black;
            y->color = Black;
            z->parent->parent->color = Red;
            z = z->parent->parent;
        } else { //case2 or case3
            if(z == z->parent->right) { //case2
                z = z->parent; //marked z.parent as new z
                left_rotate(t, z);
            }
            //case3
            z->parent->color = Black; //made parent black
            z->parent->parent->color = Red; //made parent red
            right_rotate(t, z->parent->parent);
        }
    } //end of z.parent is the left child
```

RBT Code in C – Insertion Fixup (cont.)

```
        else { //z.parent is the right child
            tree_node *y = z->parent->parent->left; //uncle of z

            if(y->color == Red) {
                z->parent->color = Black;
                y->color = Black;
                z->parent->parent->color = Red;
                z = z->parent->parent;
            }
            else {
                if(z == z->parent->left) {
                    z = z->parent; //marked z.parent as new z
                    right_rotate(t, z);
              }
              z->parent->color = Black; //made parent black
              z->parent->parent->color = Red; //made parent red
              left_rotate(t, z->parent->parent);
            }
        } //end of z.parent is the right child
    } //end of while
    t->root->color = Black;
} //end of function
```

RBT Code in C – Insert

```c
void insert(red_black_tree *t, tree_node *z) {
    tree_node* y = t->NIL; //variable for the parent of the added node
    tree_node* temp = t->root;

    while(temp != t->NIL) {
        y = temp;
        if(z->data < temp->data) temp = temp->left;
        else temp = temp->right;
    }
    z->parent = y;

    if(y == t->NIL) { //newly added node is root
        t->root = z;
    } else if(z->data < y->data) //data of child is less than its parent, left child
        y->left = z;
    else
        y->right = z;

    z->right = t->NIL;
    z->left = t->NIL;

    insertion_fixup(t, z);
}
```

RBT Code in C – Inroder

```c
void inorder(red_black_tree *t, tree_node *n) {
    if(n != t->NIL) {
        inorder(t, n->left);
        printf("%d\n", n->data);
        inorder(t, n->right);
    }
}
```