CS 590 – Algorithms

Lecture 7 – Red-Black Trees

Outlines

7. Red-black trees

Red-black trees

- A variation of binary search trees.
- Self-balancing BST:
  - ***Balanced***: height is $O(\lg n)$, where n is the number of nodes.
  - Operations will take $O(\lg n)$ time in the worst case.

7.1. Properties

Red-black trees:

- All attributes of BST will be inherited in ***red-black tree (RBT)***
- But RBT will have one extra (+1) bit per node.
    - New attribution: x.color - either **red** or **black**.
- RBT uses a single sentinel, T.nil, for all the leaves.
- T.nil.color = black
- T.root.p = T.nil

7.1. Properties

RBT Color Properties:

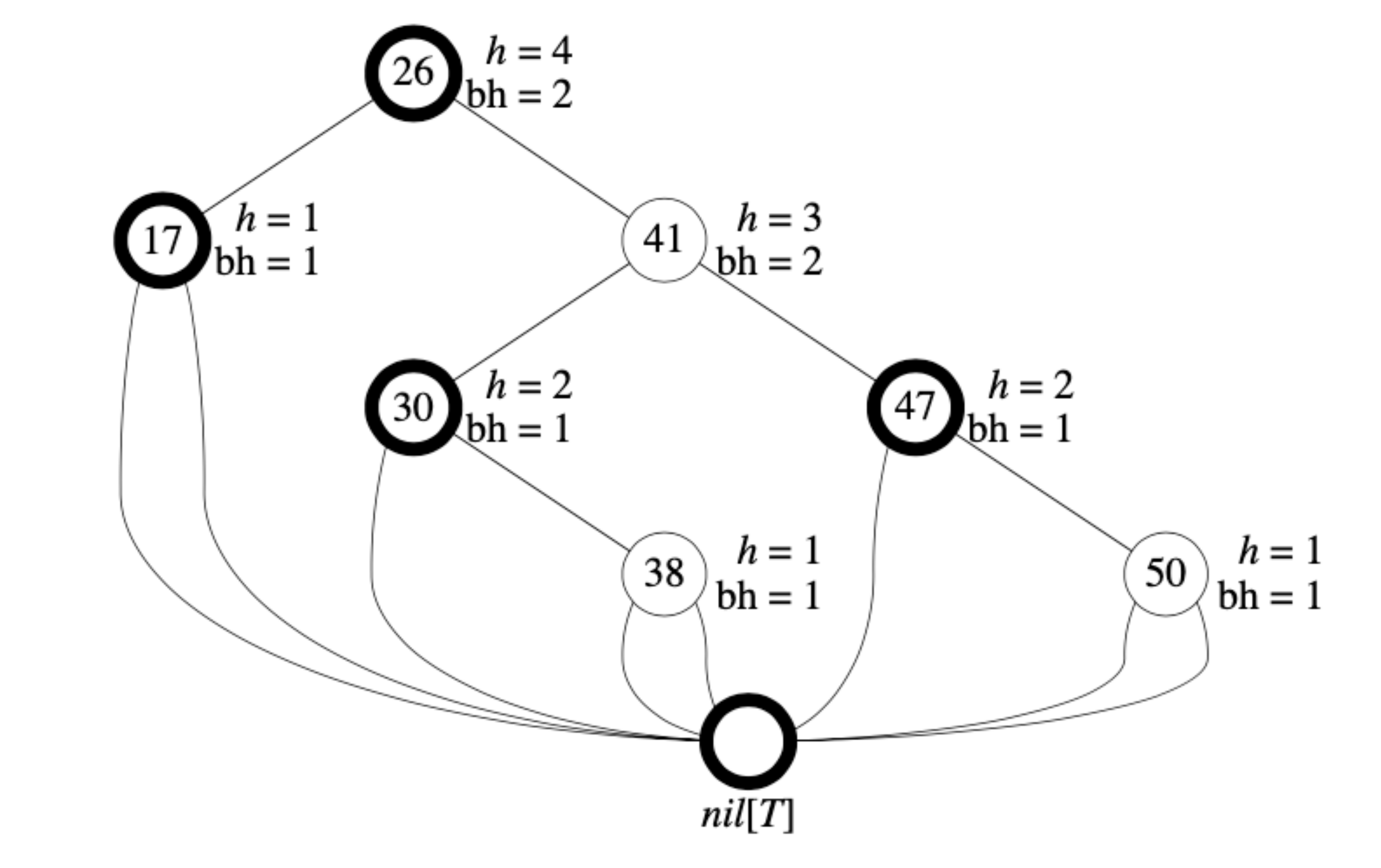
1. Every node, x.color, is either red or black.
2. T.root.color = black.
3. T.nil.color = black.
4. If x.color = red, x.left.color != red and x.right.color != red.
5. All paths from x to its descent will have the **same number of black nodes** (see black heights, bh(x)).

7.1. Properties

Black Height (bh) Properties:

- The height of the tree, h(T), is the number of edges in the longest path to a leaf.
- The black height, bh(x), is the number of black nodes from a node x to a leaf.
  - x.color is exclusive.
  - T.nil is inclusive.

7.2. Black Heights



7.2. Black Heights

- **<u>Claim 1:</u>**
  - If $h(x) = h$, $bh(x)$ is $\geq h/2$ for all x in T.
- **<u>Claim 2:</u>**
  - A subtree of x contains >= $2^{bh(x)} - 1$ internal nodes.
- **<u>Lemma 1:</u>**
  - An RBT with n-many internal nodes has h <= 2*lg(n+1).

7.3. Operations

Operations on red-black trees

- The non-modifying BST operations will be carried over.
  - Inorder-Tree-Walk, Minimum, Successor…
  - Except Insertion and Deletion.
- All searching algorithms will have the running time in O(h).

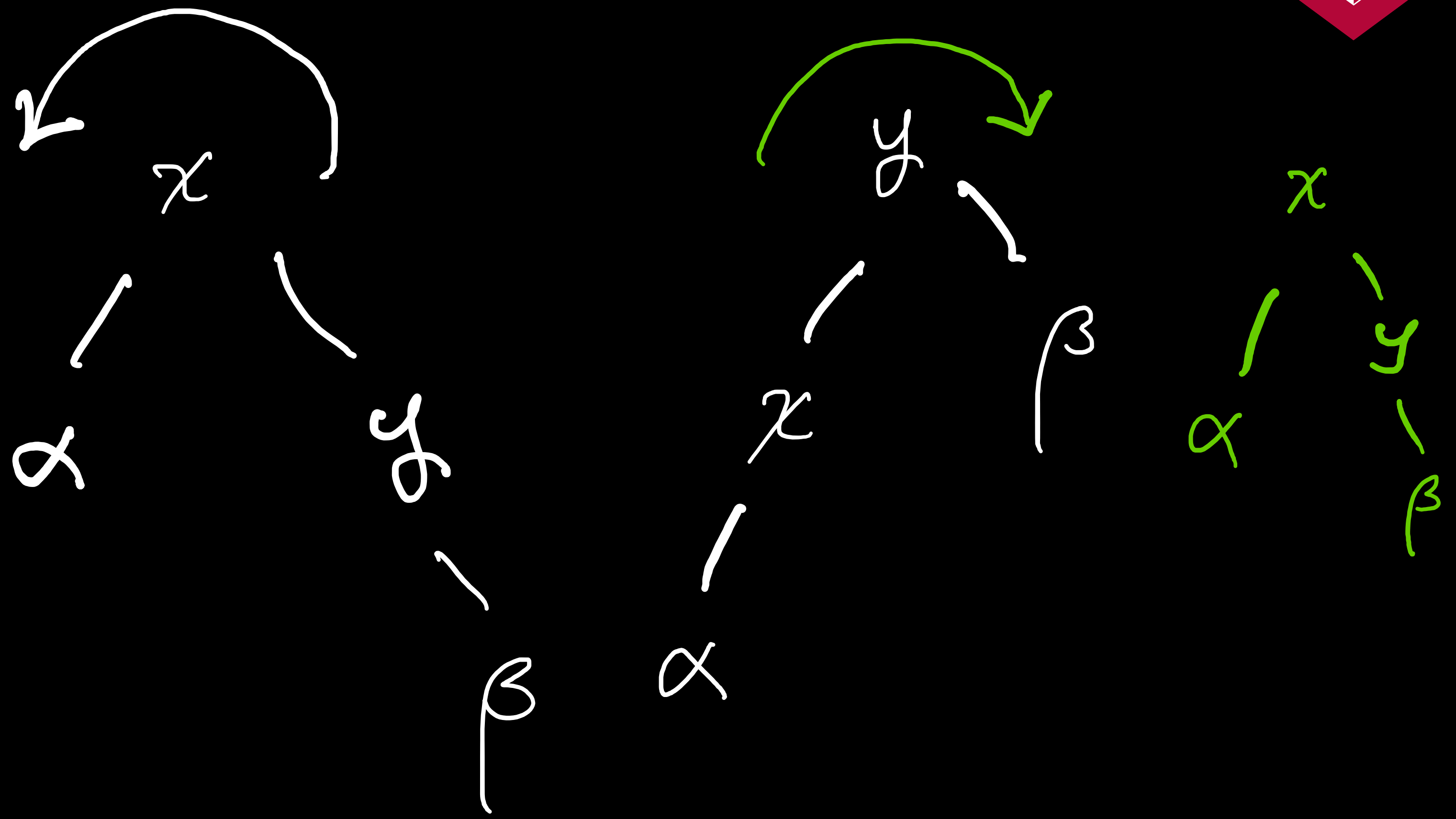
7.3.1. Rotations

Rotation Operation:

- is the basic tree-restructuring operation taking a node x within T.
- is to maintain RBTs as self-balanced BSTs.
- will use pointers to change the local pointer structure.
- must not upset the BST's property.
- operates in two directions, left and right.
- separate implementation is easier.

○ : red

○ : black



7.3.1. Rotations

LEFT-ROTATE(T, x) algorithm implementation.

1. Set y as x.right:
2. Turn the left subtree of y into the right of x:
3. What happens if y.left is not nil?
   - Link point x as y.left.p
4. Link x.p. to y.p
5. What if x.p = T.nil?
   - What becomes the root of the tree?
   - What if x is the left child?
   - What if x is the right child?

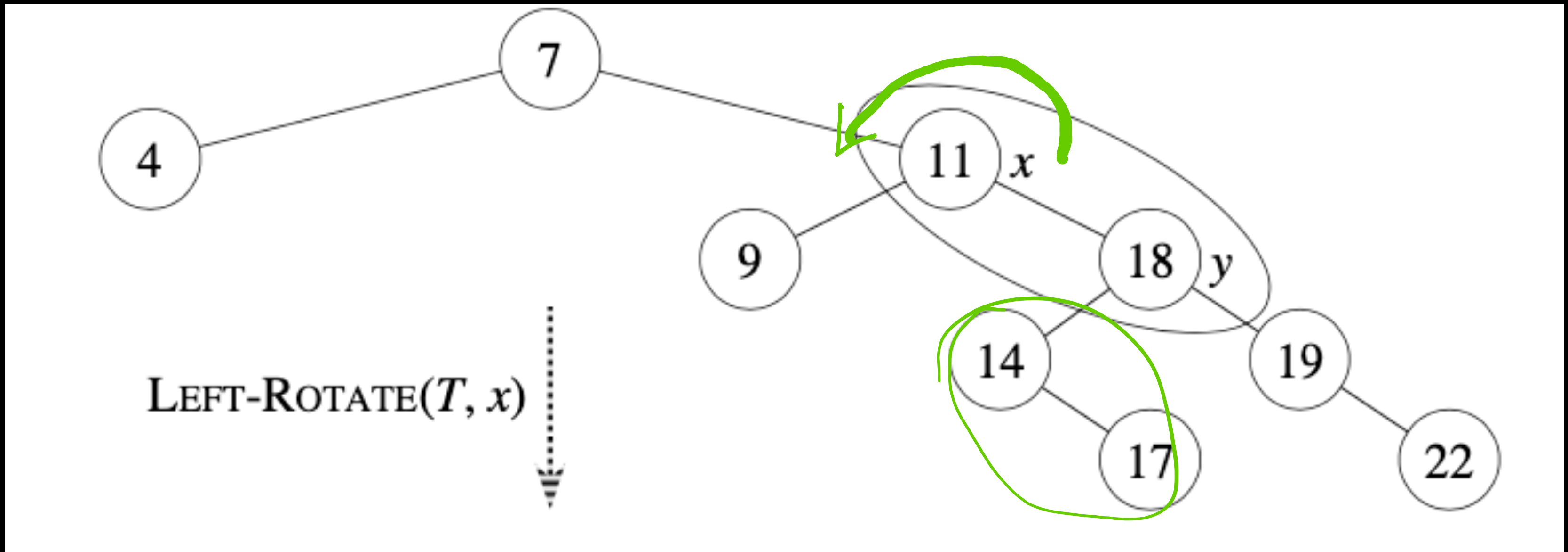
7.3.1. Rotations

```
Algorithm (LEFT-ROTATE(T,x))
    y = x.right                //set y
    x.right = y.left           //turn left subtree of y into right of x
    if (y.left ≠ T.nil) then
        y.left.p = x
    y.p = x.p                  //link parent of x to y
    if (x.p = T.nil) then
        T.root = y
    else
        if (x = x.p.left) then
            x.p.left = y
        else
            x.p.right = y
    y.left = x                 //put x of left of y
    x.p = y
```

7.3.1. Rotations

- The pseudocode for LEFT-ROTATE assumes that
  - x.right != T.nil
  - root's parent is T.nil.
- The running time is T(n) = O(1).
- The RIGHT-ROTATE algorithm is symmetric:
  - exchange left and right everywhere.

7.3.1. Rotations



7.3.2. Insertion

Insertion Consideration:

- Suppose a node z is inserted into RBT.
- If z.color = Red?
  - Property 1: ✓
  - Property 2: ✗ T.root = black
  - Property 3: ✓
  - Property 4: ✗ z.p = red ✗
  - Property 5: ✓

7.3. Operations

Insertion Consideration:

- Suppose a node z is inserted into RBT.
- If z.color = Black?
  - Property 1: ✓
  - Property 2: ✓
  - Property 3: ✓
  - Property 4: ✓
  - Property 5: ✗

7.3.2. Insertion

- Recall BST insertion:
  - Two pointers, x and y, were used to find the location.
- A new node z will be inserted as BST insertion. Why?
- z.color?
- We need an additional algorithm to fix to maintain RBT color properties.

7.3.2. Insertion

Algorithm (TREE-INSERT(T,x))

```
y = NIL, x = T.root
while (x ≠ NIL) do
    y = x
    if (z.key < x.key) then
        x = x.left
    else x = x.right
z.p = y
if (y = NIL) then
    T.root = z
else if (z.key < y.key) then
    y.left = z
else y.right = z
```

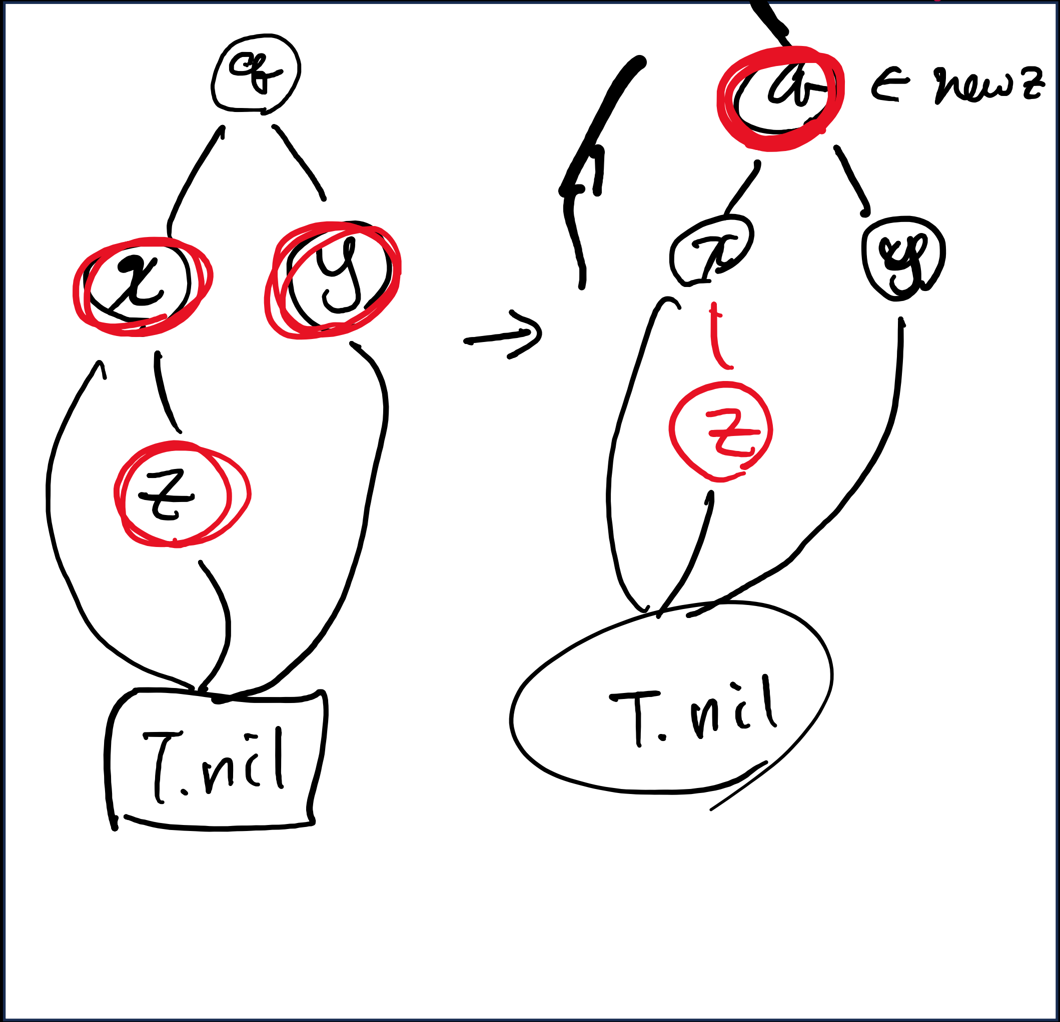
7.3.2. Insertion

- The color properties are violated when z.color = z.p.color = red.
- The new balancing considers the following cases depending on the color of z's uncle, y.color.
  1. z.p = z.p.p.left & y.color = red (case 1)
     1. z = z.p.right & y.color = black (case 2)
     2. z = z.p.left & y.color = black (case 3)
  2. Symmetric cases when z.p = z.p.p.right.

7.3.2. Insertion

Case 1: z.p = z.p.p.left and y.color = red

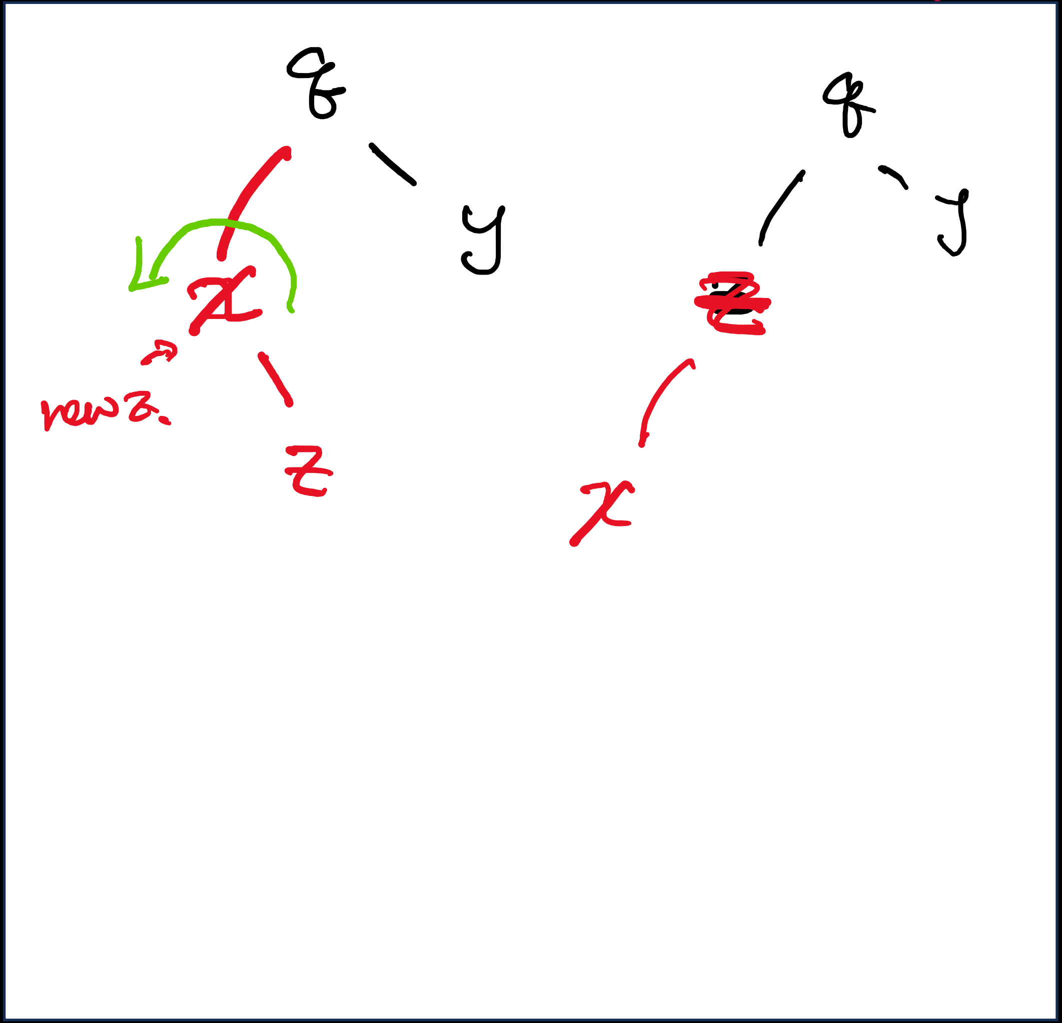
- Assume z.p.p.color = black.
- **Make z.p.color = y.color = black.**
    - Property 4:
    - Property 5:
- **Make z.p.p.color = red**.
- Let new z be z.p.p.



7.3.2. Insertion

Case 2: z = z.p.right and y.color = black

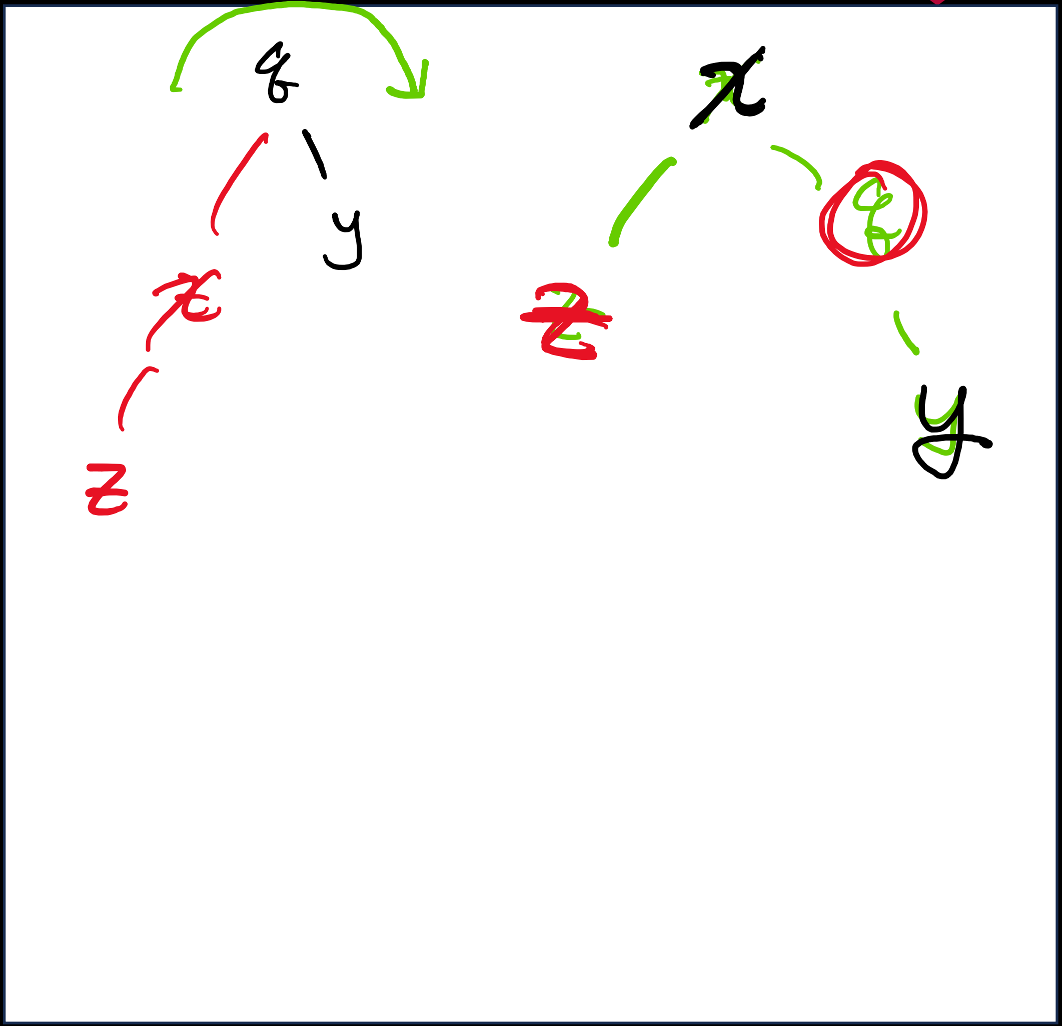
- **Let z = z.p then**
- **LEFT-ROTATE(T, z)**
  - Property 4: ✗
  - Property 5: ✗
- Transforms to case 3: z = z.p.left



7.3.2. Insertion

Case 3: z = z.p.left and y.color = black

- **Make z.p.color = red and z.p.p.color = balck**
- **RIGHT-ROTATE(T, z.p.p)**
  - Property 4:
  - Property 5:
- Transforms to case 3: z = z.p.left

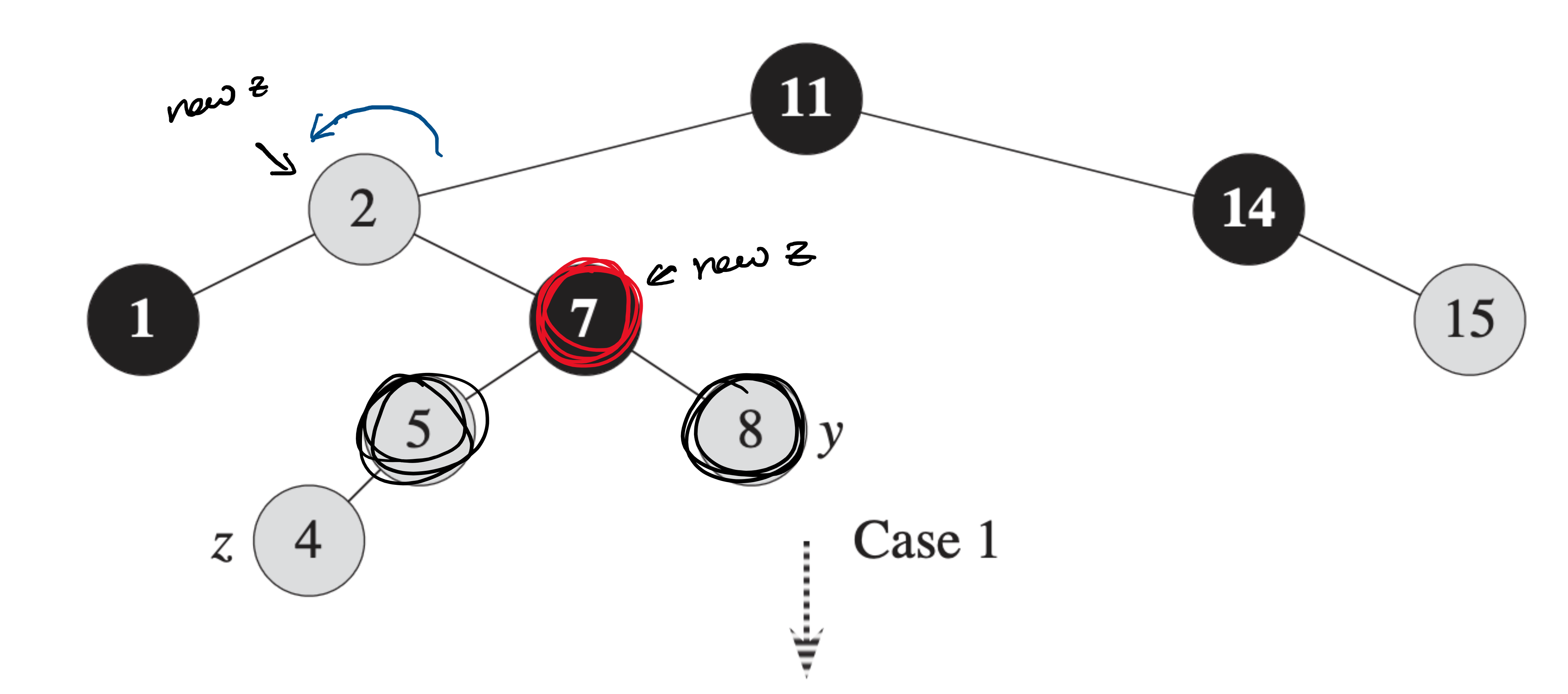


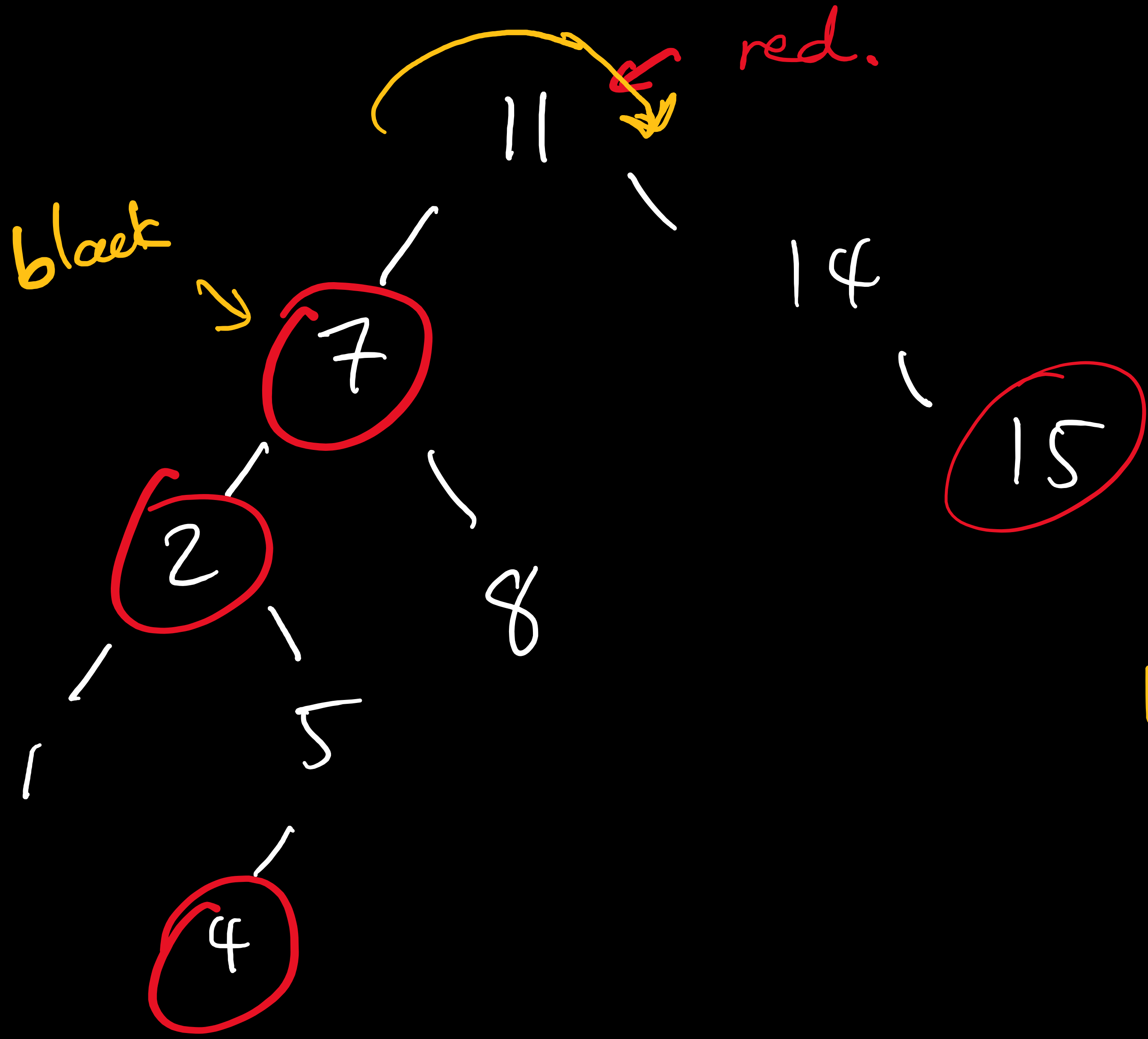
7.3.2. Insertion

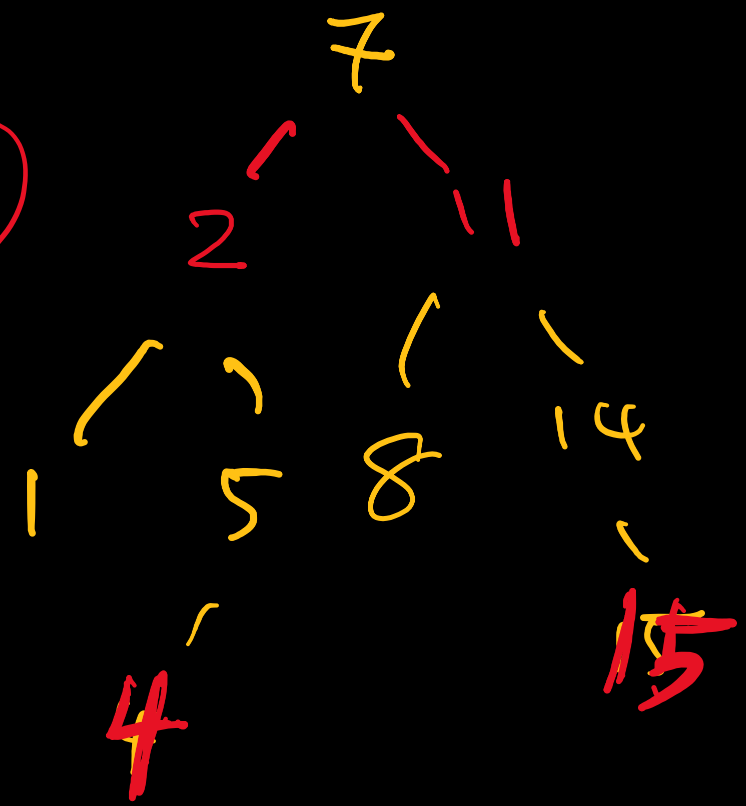
Algorithm (RB-**INSERT-FIXUP(T,z)**)

```
while (z.p.color = RED)
   if (z.p = z.p.p.left)
      y = z.p.p.right
      if (y.color = RED)
         z.p.color = BLACK, y.color = BLACK    //case 1
         z.p.p.color = RED
         z = z.p.p
      else
         if (z = z.p.right) then
            z = z.p                            //case 2
            LEFT-ROTATE(T,z)
         z.p.color = BLACK, z.p.p.color = RED  //case 3
         RIGHT-ROTATE(T, z.p.p)
   else
      (do same with right and left exchange)
T.root.color = BLACK
```

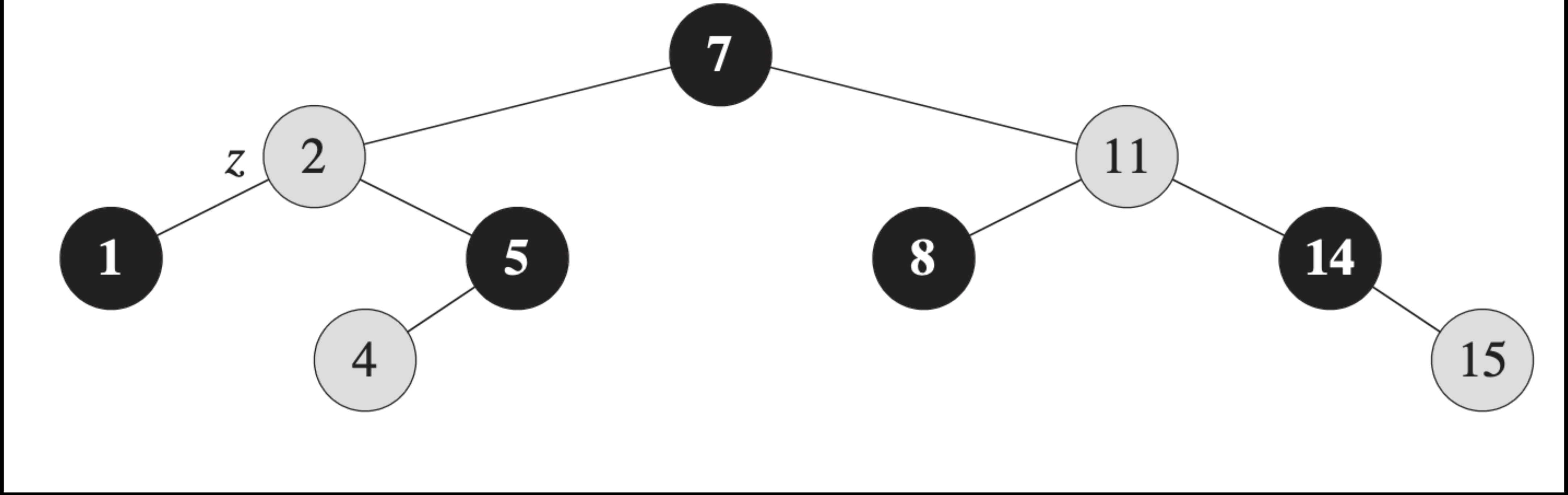
7.3.2. Insertion




red.
black
11
7
14
2
8
15
1
5
4


7
2
11
1
5
8
14
4
15

7.3.2. Insertion



7.3.2. Insertion

Analysis

$O(\lg n)$ time to get through RB-INSERT up to the call of RB-INSERT-FIXUP.

Within RB-INSERT-FIXUP:

- Each iteration takes $O(1)$ time.
- Each iteration is either the last one or it moves z up 2 levels.
- $O(\lg n)$ levels $\Rightarrow O(\lg n)$ time.
- Also note that there are at most 2 rotations overall.

Thus, insertion into a red-black tree takes $O(\lg n)$ time.

7.3. Operations

Deletion Consideration:

- Suppose a node z is removed.
  - If z.color = red?
    - Property 1:
    - Property 2:
    - Property 3:
    - Property 4:
    - Property 5:

7.3. Operations

Deletion Consideration:

- Suppose a node z is removed.
  - If z.color = black?
    - Property 1: ✓
    - Property 2: ✗ ✓
    - Property 3:
    - Property 4: ✗
    - Property 5: ✗