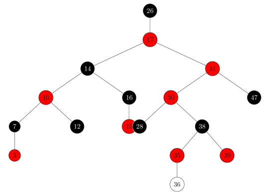


Red-Black Trees 2 - Insertion
Lecture 16

COSC 242 – Algorithms and Data Structures

UNIVERSITY of OTAGO
Te Whare Wānanga o Otāgo
NEW ZEALAND

Today’s outline

1. Rotations
2. Insertion
3. Insert Fixup
4. Cases
5. Examples

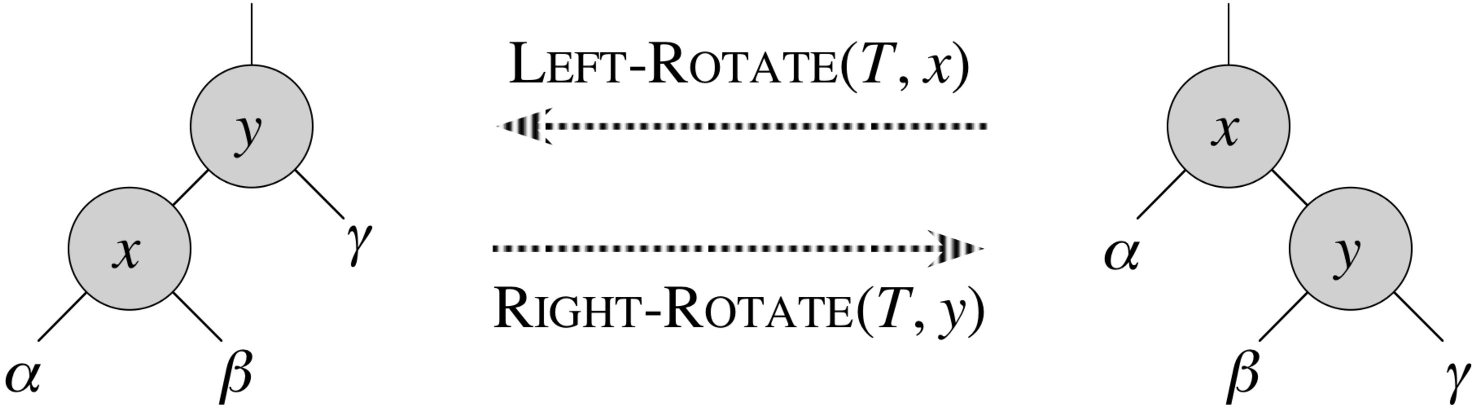
Today’s outline

1. Rotations
2. Insertion
3. Insert Fixup
4. Cases
5. Examples

Rotations

The operations insert and delete when run on an RBT with n nodes takes O(log n) time. Because these operations modify the RBT, the result may violate the RBT properties.

Rotation provides efficient rebuilding to maintain these properties.



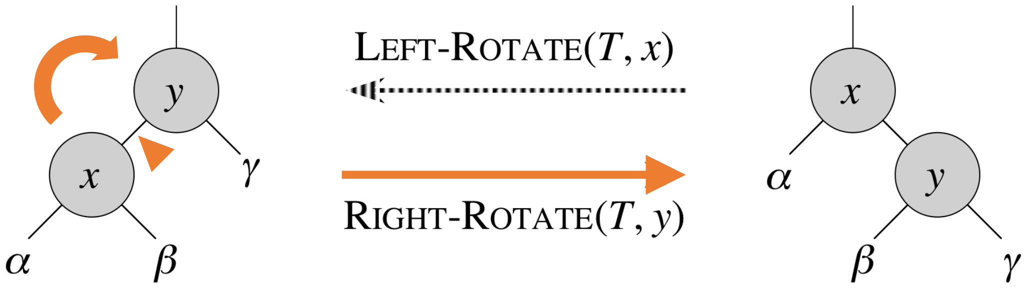
The letters $\alpha, \beta, and\ \gamma$ represent three arbitrary subtrees.

Rotations

Rotations work by updating the pointer structure of the tree. When do do a right-rotation on node *y*, we assume that its left child *x* is not *T.nil*. *y* may be any node in the tree whose left child is not *T.nil*.

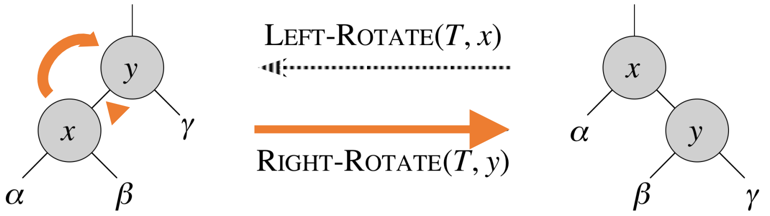
Right-rotation

- "pivots" around the edge from *x* to *y*.
- Makes *x* the new root of the subtree
- *y* becomes *x*'s right child
- *x*'s right child becomes *y*'s left child.



Right rotation


```
procedure RBT_Rotate_Right(x)
    y = x→parent                // set y
    y→left = x→right            // turn x's right subtree β into y's left subtree
    x→right = y                 // set y as x's right subtree
    x→parent = y→parent         // x's parent becomes y's parent
    if y→parent == NILL then
        root = x                // y was root, make x tree root
    else
        y→parent→[left or right] = x // update y's R or L child to point to x
    end if
    y→parent = x                // y's parent is now x
end procedure
```

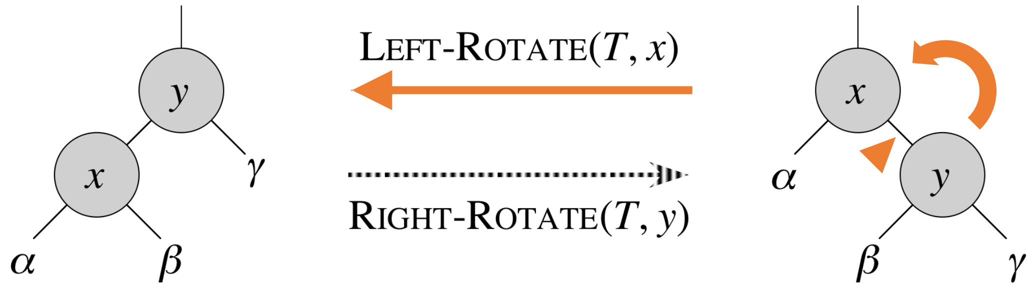


Rotations

When do do a left-rotation on node *x*, we assume that its right child *y* is not *nil*. *x* may be any node in the tree whose right child is not *nil*.

Left-rotation

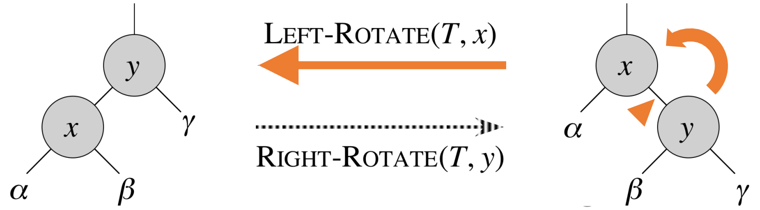
- “pivots” around the edge from *x* to *y*.
- Makes *y* the new root of the subtree
- *x* becomes *y*’s left child
- *y*’s left child becomes *x*’s right child.



The pseudocode for left rotate is symmetric: exchange right with left everywhere.

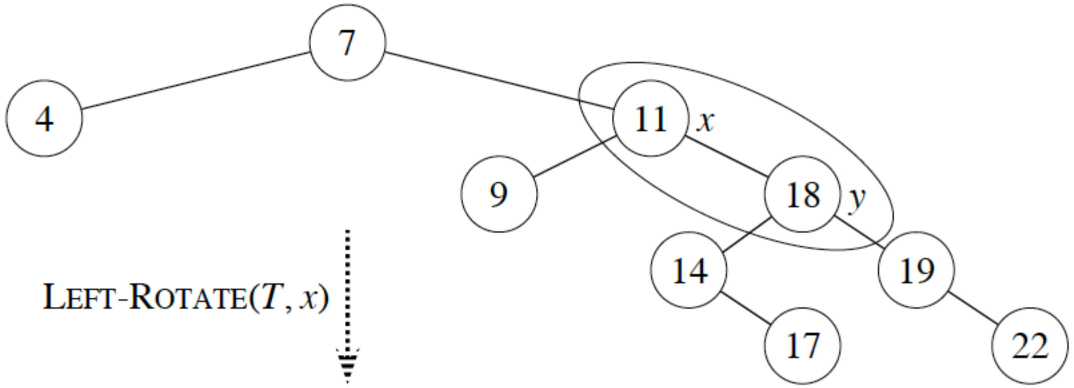
Left rotation

```
procedure RBT_Rotate_Left(x)
    y = x→right                 // set y
    x→right = y→left            // turn y’s left subtree β into x’s right subtree
    y→left = x                  // set x as y’s left subtree
    y→parent = x→parent         // y’s parent becomes x’s parent
    if x→parent == NILL then
        root = y                // x was root, make y tree root
    else
        x→parent→[left or right] = y // update x’s R or L child to point to y
    end if
    x→parent = y                // x’s parent is now y
end procedure
```



Class challenge

Perform an RBT_Rotate_left(T, x) operation:



Node colours omitted for convenience.

Today’s outline

1. Rotations
2. Insertion
3. Insert Fixup
4. Cases
5. Examples

Insertion

The basic algorithm for inserting a node into an RBT is:

```
procedure RBT_Insert(T, x)
    BST_insert(T, x)
    x.colour = RED
    if x→parent == RED then        // Violation of property 4
        RBT_Insert_Fixup(T, x)
    end if
end procedure
```

Insertion

By colouring *x* red, we may violate property 4 that says red nodes have black children. Think of *x* as the problem node. We call `RBT_Insert_Fixup` to restore red-black properties.

All fixups push the problem back up the tree, so `RBT_Insert_Fixup` needs to traverse the tree upwards until either there is no problem anymore, or we reach the root of the tree.

This can be done recursively or iteratively.

Today’s outline

1. Rotations
2. Insertion
3. Insert Fixup
4. Cases
5. Examples


```
procedure RBT_Insert_Fixup(T, z)
    while z→parent.colour == RED
        if z→parent == z→parent→parent→left      // z’s parent is a left-child
            y = z→parent→parent→right            // set y to z’s “uncle”
            if y→colour == RED
                z→parent.colour = BLACK          // case 1
                y.colour = BLACK                 // case 1
                z→parent→parent = RED            // case 1
                z = z→parent→parent              // case 1
            else
                if z = z→parent→right
                    z = z→parent                 // case 2
                    RBT_Rotate_Left(T, z)        // case 2
                end if
            z→parent.colour = BLACK              // case 3
            z→parent→parent = RED                // case 3
            RBT_Rotate_Right(T, z→parent→parent) // case 3
        else ... (same as then clause, with ”right” and “left” exchanged)
    T→root.colour = BLACK
```

Fixup procedure

To understand Fixup, we will break our investigation of the pseudocode into three major steps:

1. We will examine what violates of RBT properties are introduced by RBT_Insert.
2. We will consider the overall goal of the while loop lines 1-17.
3. We will explore each of the three cases.

Property violations

1. Every node is either red or black.
2. The root is black.
3. Every leaf (nil/null) is black.
4. If a node is red, then both its children are black*.
5. For each node, all paths from the node to leaves contain the same number of black nodes.

RBT properties upon entering Fixup

1. Satisfied, as *z* is red.
2. Violated if *z* is the root.
3. Satisfied, as both children of new node are *T.nil*
4. Violated if parent is red, as *z* is also red.
5. Satisfied, as *z* replaces black sentinel, and node *z* is red, with two black sentinel children.

While loop

The **while** loop in lines 1–15 maintains the following three-part invariant at the start of each iteration of the loop:

a) Node *z* is red

b) If *z*→`parent` is the root, then *z*→`parent` is black

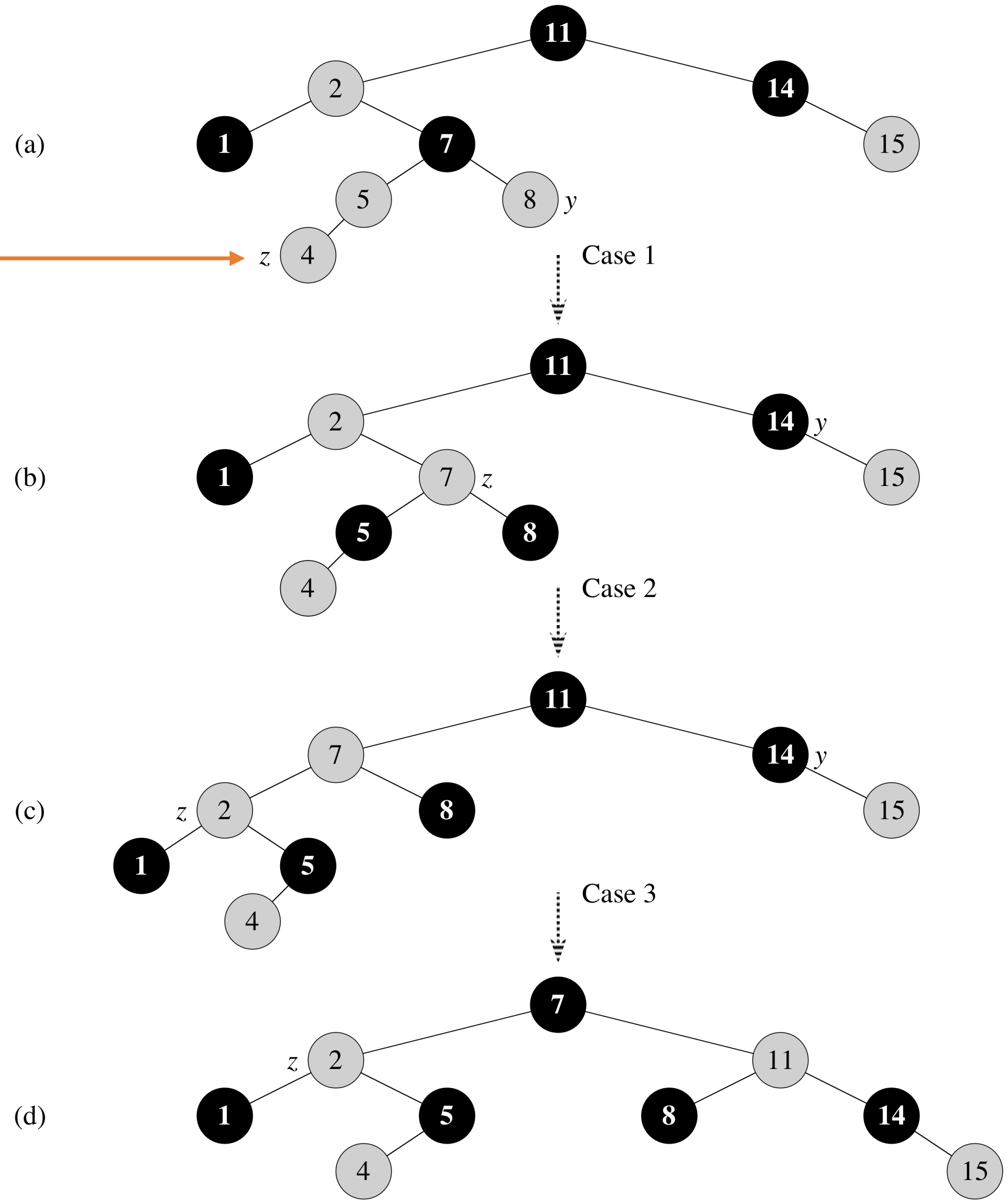
c) If the tree violates any of the RBT properties, then it violates at most one of them, which is either property 2 or property 4.

Today's outline

1. Rotations
2. Insertion
3. Insert Fixup
4. Cases
5. Examples

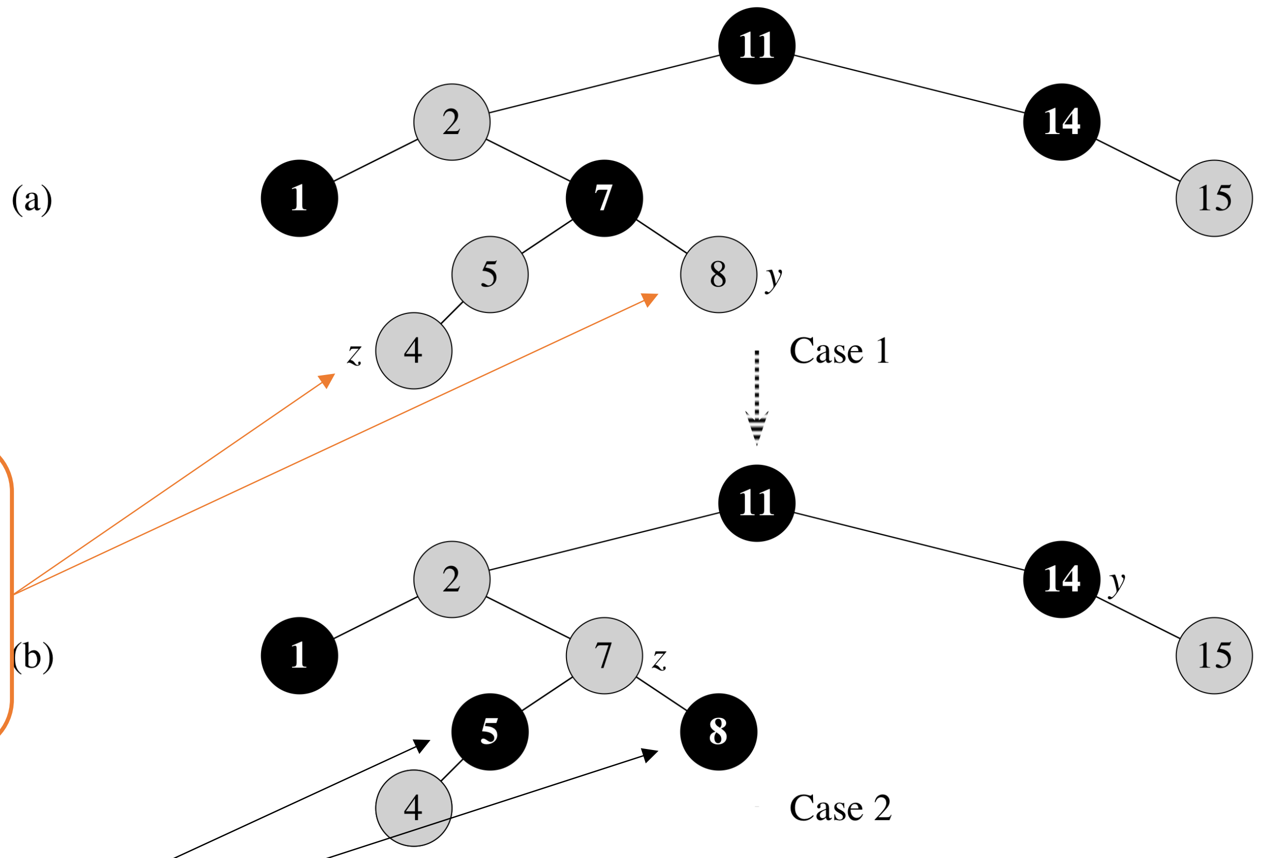
Example

Insert 4



Case 1 violation

```
procedure RBT_Insert_Fixup(T, z)
1:    while z→parent.colour == RED
2:        if z→parent == z→parent→parent→left
3:            y = z→parent→parent→right
4:            if y→colour == RED
5:                z→parent.colour = BLACK // case 1
6:                y.colour = BLACK        // case 1
7:                z→parent→parent = RED   // case 1
8:                z = z→parent→parent     // case 1
9:            else
```

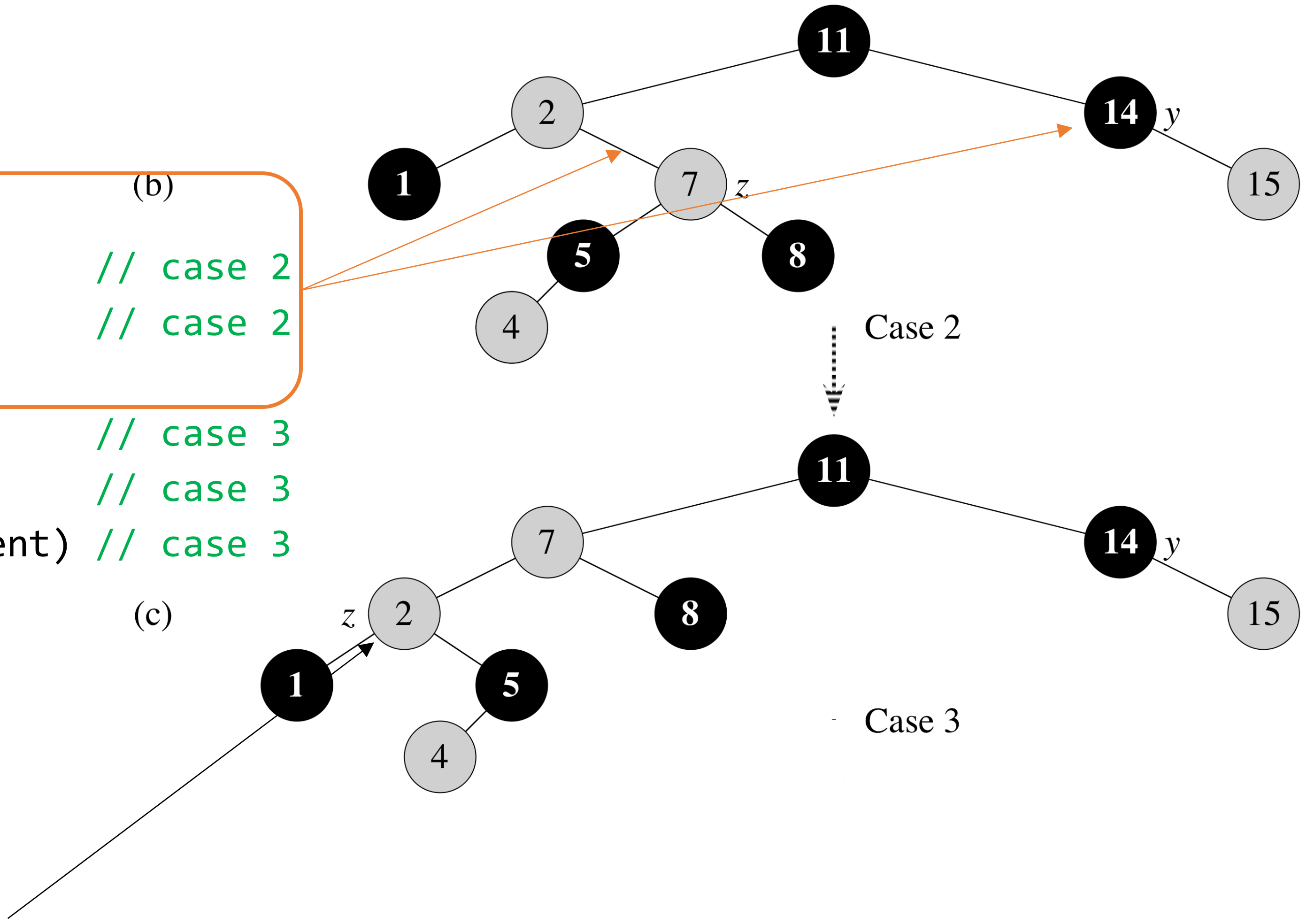


Case 1: z's uncle, y, is red.

- L5-6: Colour z's parent and uncle black
- L7: Colour z's grandparent black
- L8: Z now points to z's grandparent

Case 2 violation

```
9:          else
10:             if z = z→parent→right
11:                 z = z→parent                    // case 2
12:                 RBT_Rotate_Left(T, z)           // case 2
13:             end if
14:         z→parent.colour = BLACK                 // case 3
15:         z→parent→parent = RED                   // case 3
16:         RBT_Rotate_Right(T, z→parent→parent) // case 3
17:     else ...
18  T→root.colour = BLACK
```

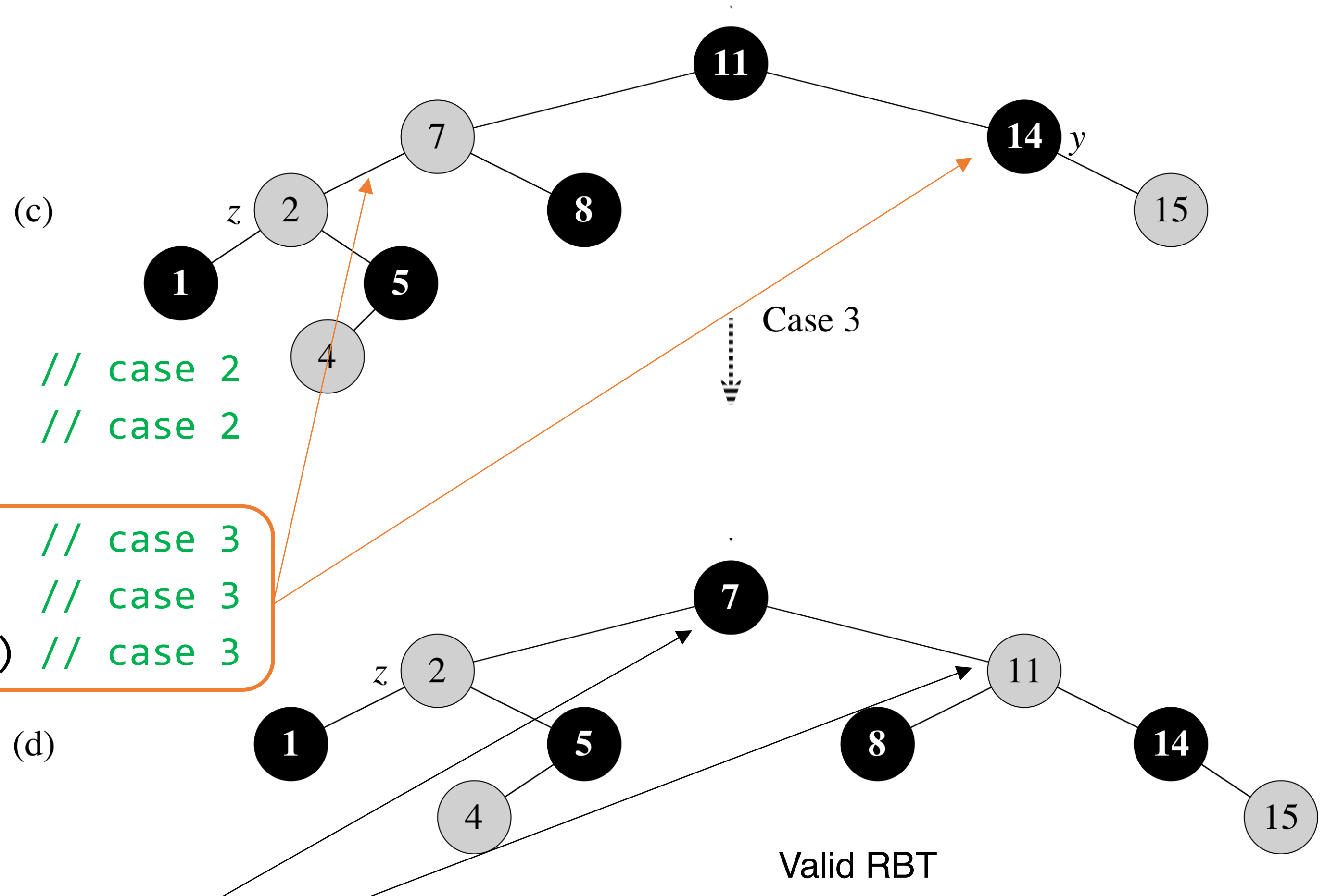


Case 2: z's uncle y is black and z is a right child.

- L11: z now points to z's parent
- L12: Left rotate on z (i.e. old z's parent)

Case 3 violation

```
9:              else
10:                 if z = z→parent→right
11:                     z = z→parent                    // case 2
12:                     RBT_Rotate_Left(T, z)           // case 2
13:                 end if
14:             z→parent.colour = BLACK                 // case 3
15:             z→parent→parent = RED                   // case 3
16:             RBT_Rotate_Right(T, z→parent→parent)    // case 3
17:         else ...
18      T→root.colour = BLACK
```



Case 3 always falls through from case 2

Case 3: z's uncle y is black and z is a left child.

- L14: Colour z's parent black
- L15: Colour z's grandparent red
- L16: Right rotate on z's grandparent

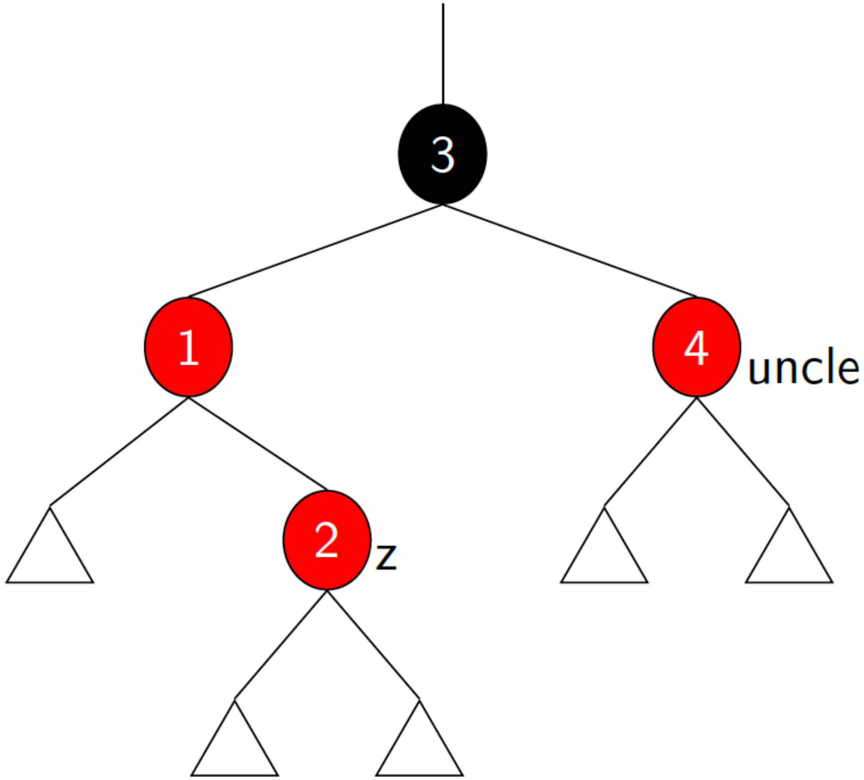
Today’s outline

1. Rotations
2. Insertion
3. Insert Fixup
4. Cases
5. Examples

Case 1 violation

Insert: 3, 1, 4, 2

Initial:

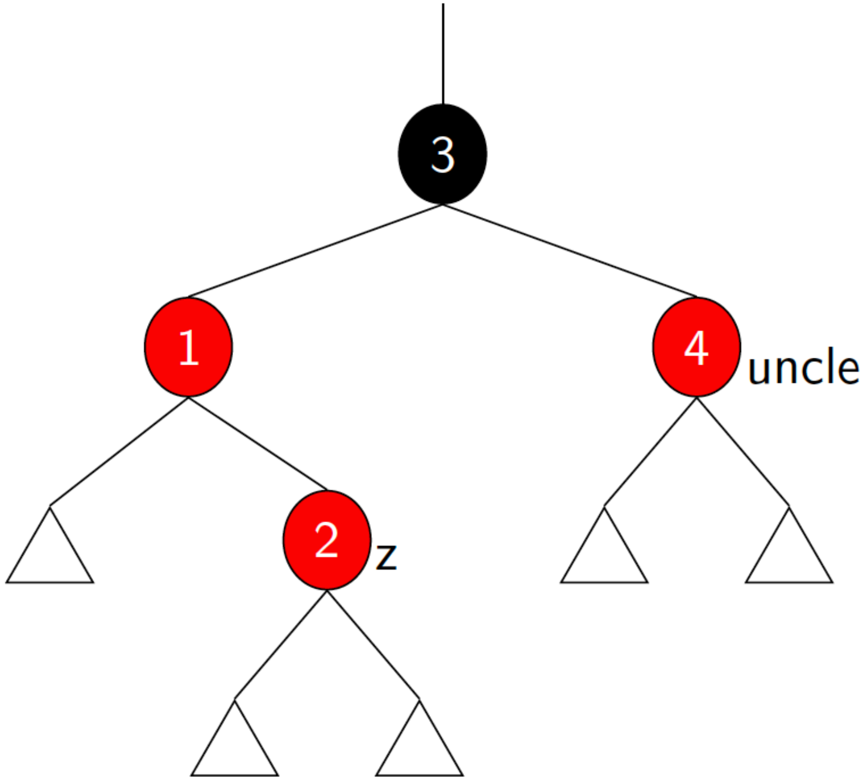


Case 1 violation

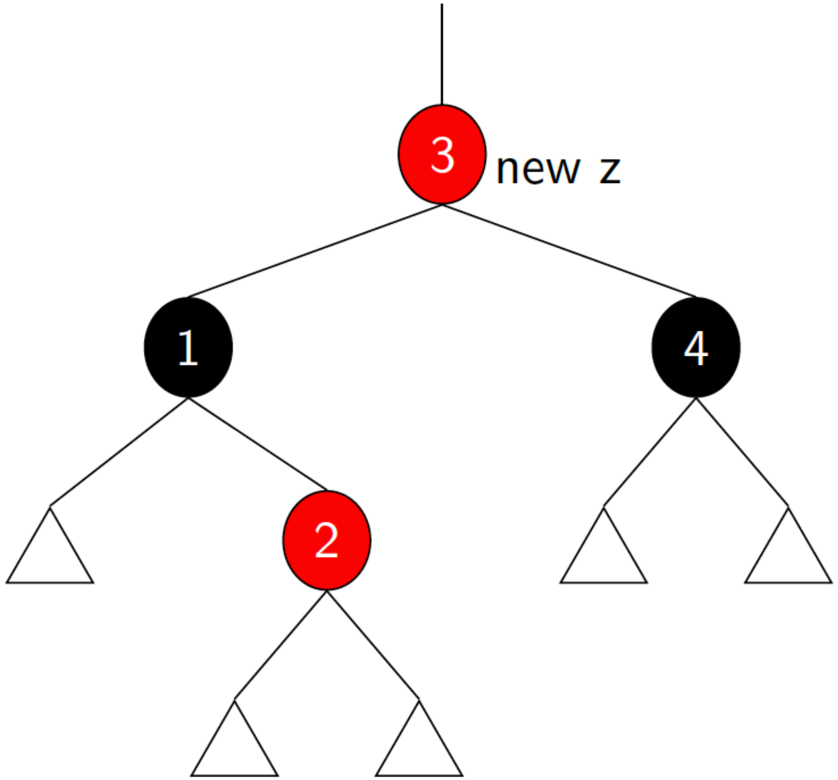
Case 1: *z*'s uncle is red.
It might cause a violation further up the tree.

Insert: 3, 1, 4, 2

Initial:

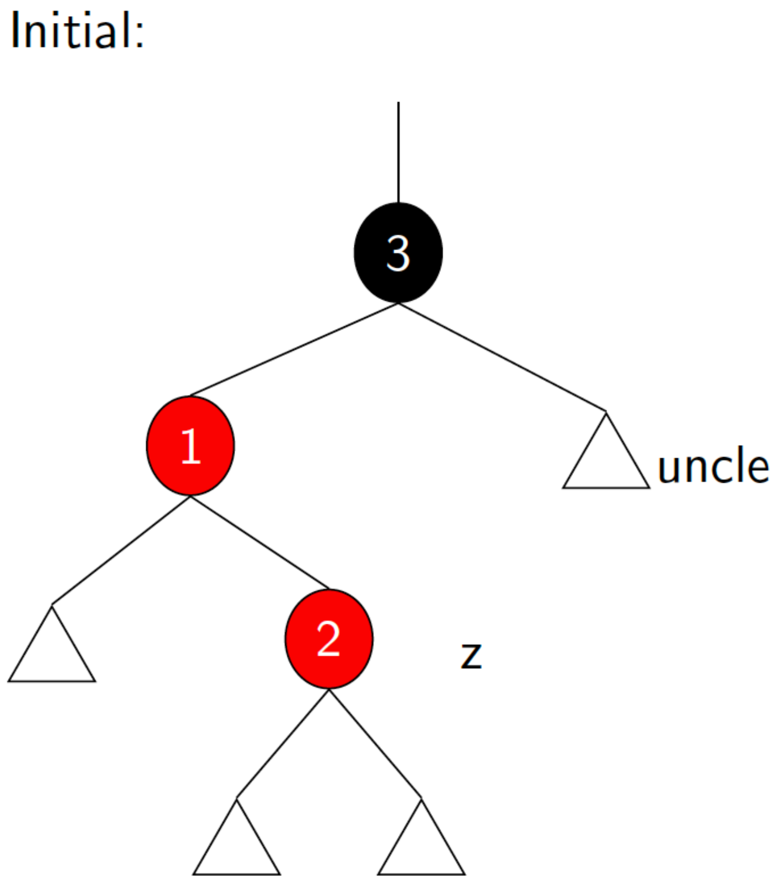


Fixup:



Case 2 violation

Insert: 3, 1, 2

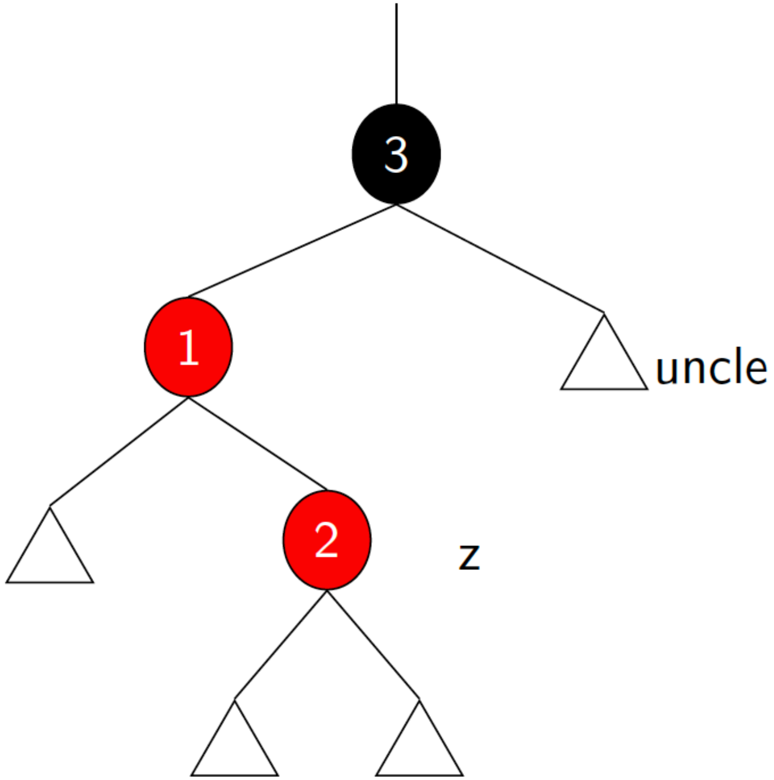


Case 2 violation

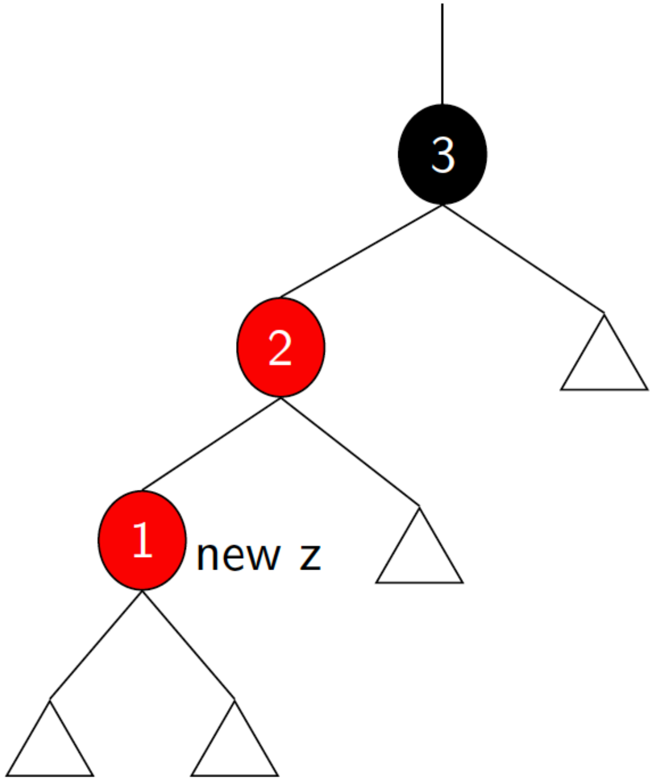
Case 2: z's uncle is black and z is a right child.

Insert: 3, 1, 2

Initial:



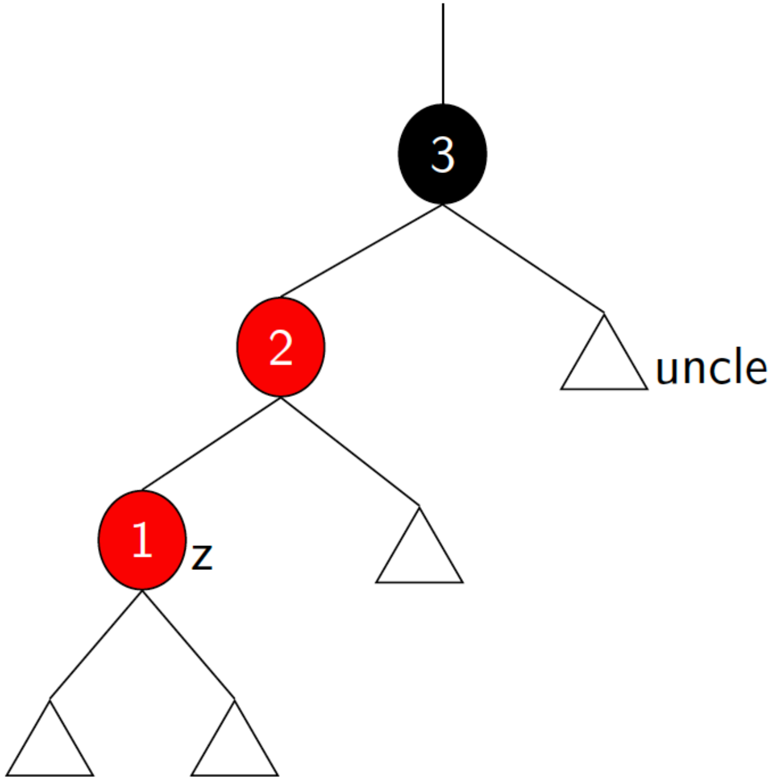
Fixup:



Case 3 violation

Following on from case 2 violation…

Initial:

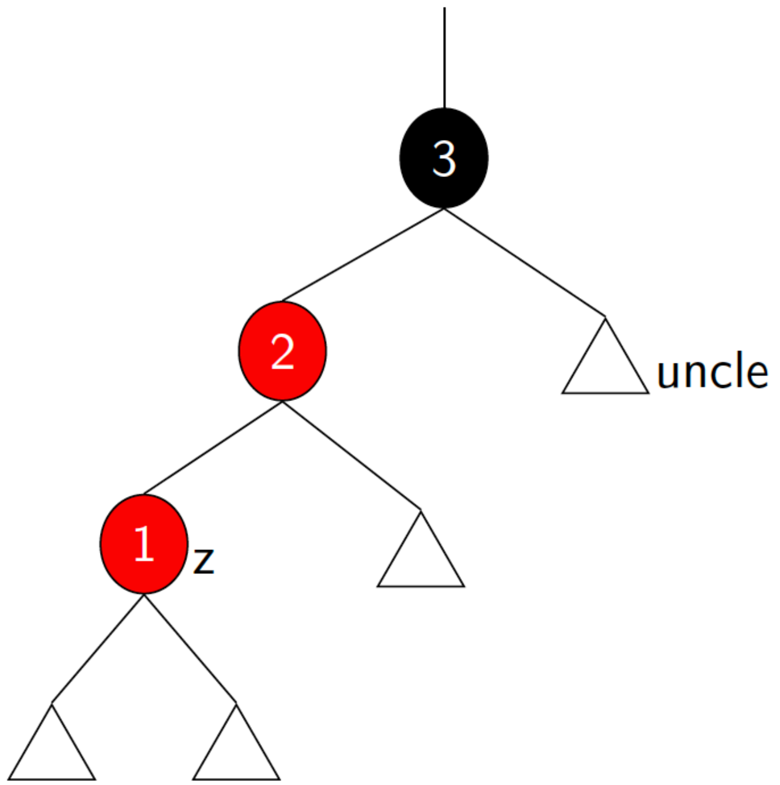


Case 3 violation

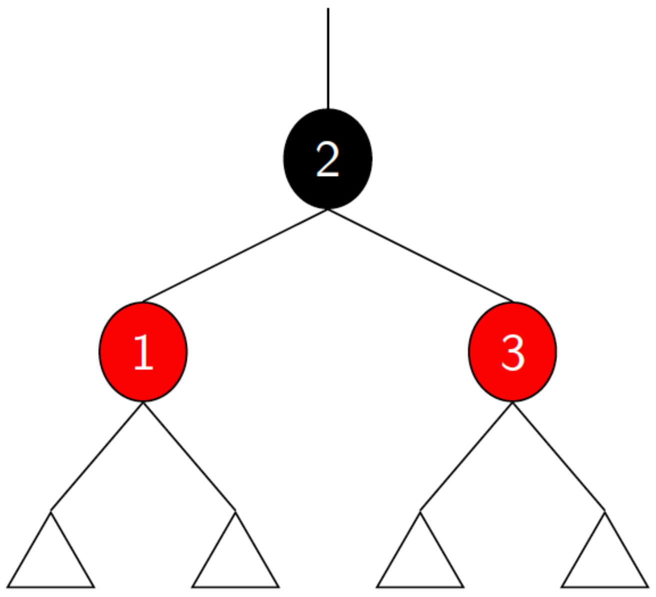
Case 3: z's uncle is black and z is a left child.

Following on from case 2 violation…

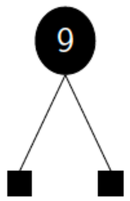
Initial:



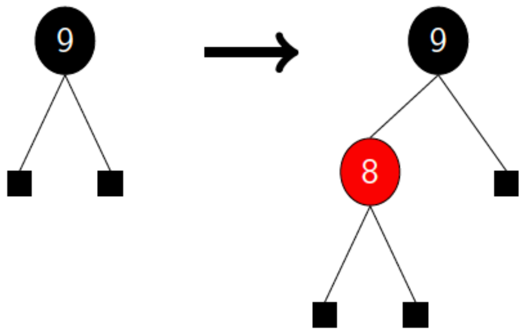
Fixup:



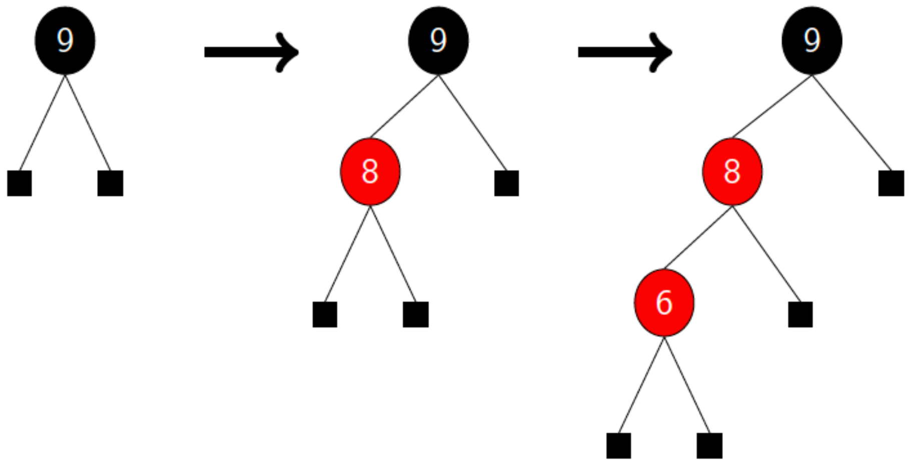
Example Insertions (9, 8, 6, 3, 5)



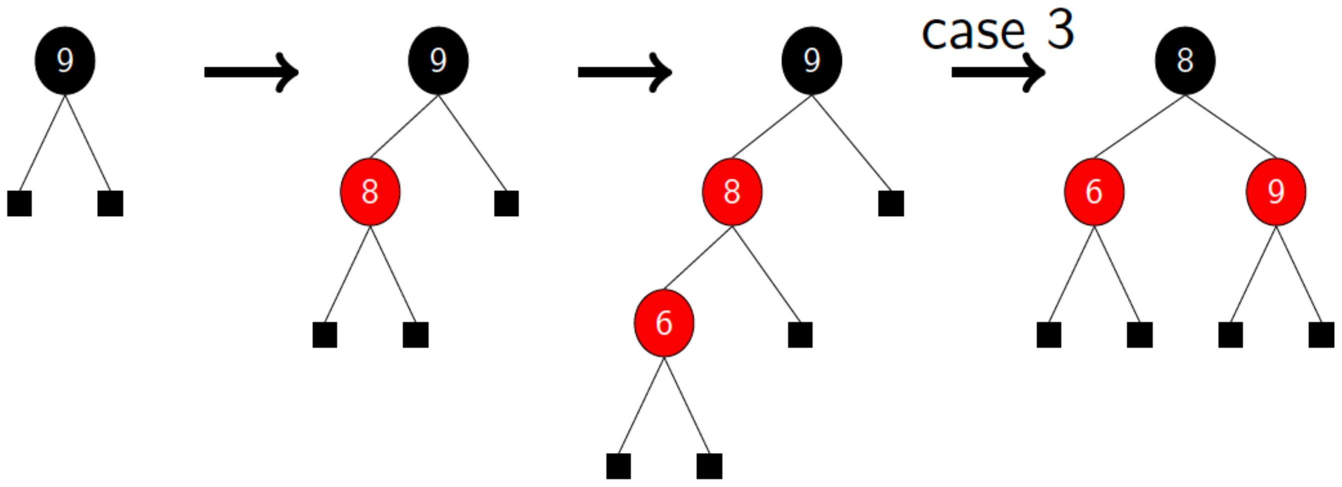
Example Insertions (9, 8, 6, 3, 5)



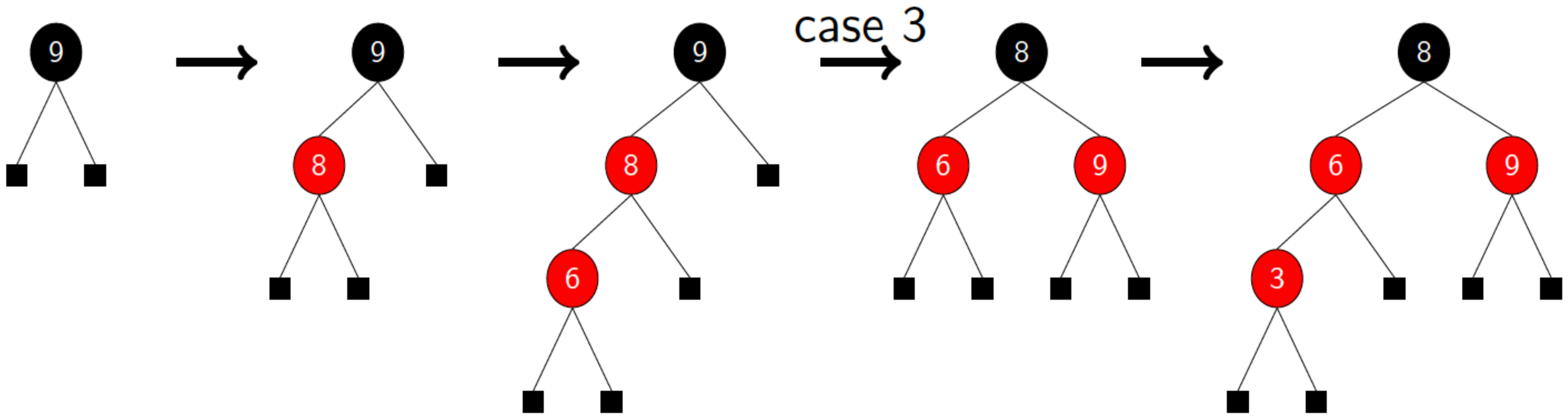
Example Insertions (9, 8, 6, 3, 5)



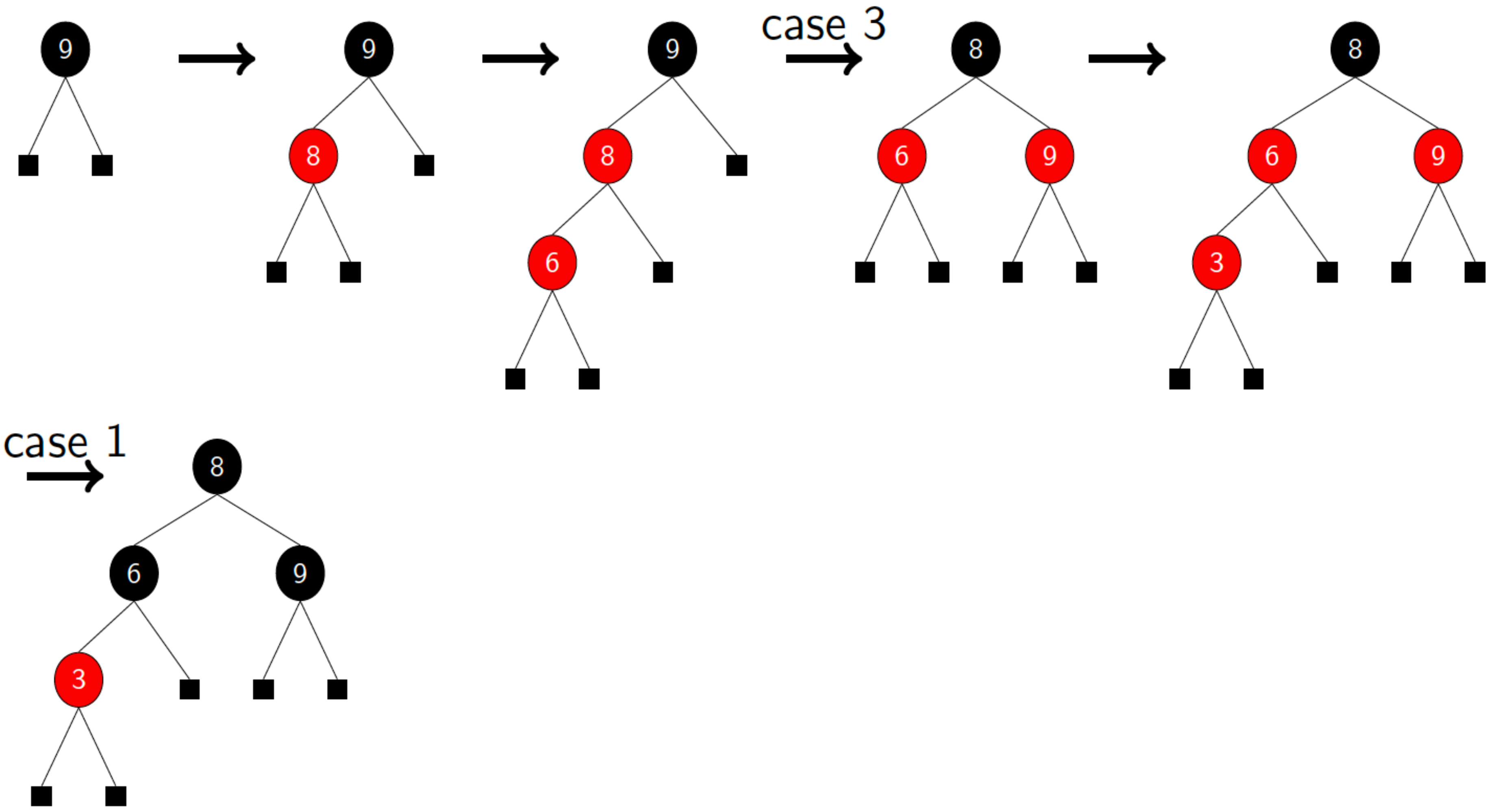
Example Insertions (9, 8, 6, 3, 5)



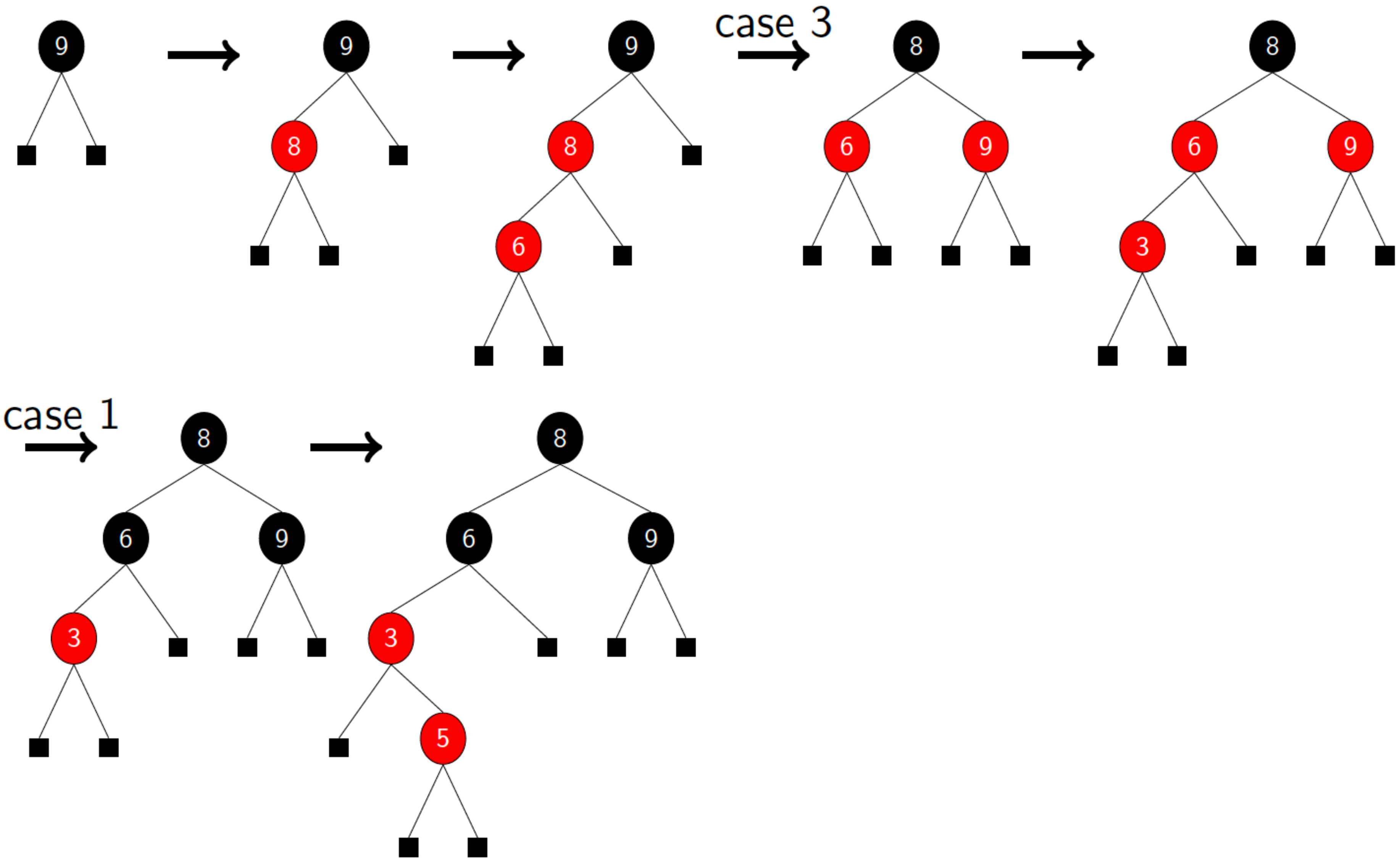
Example Insertions (9, 8, 6, 3, 5)



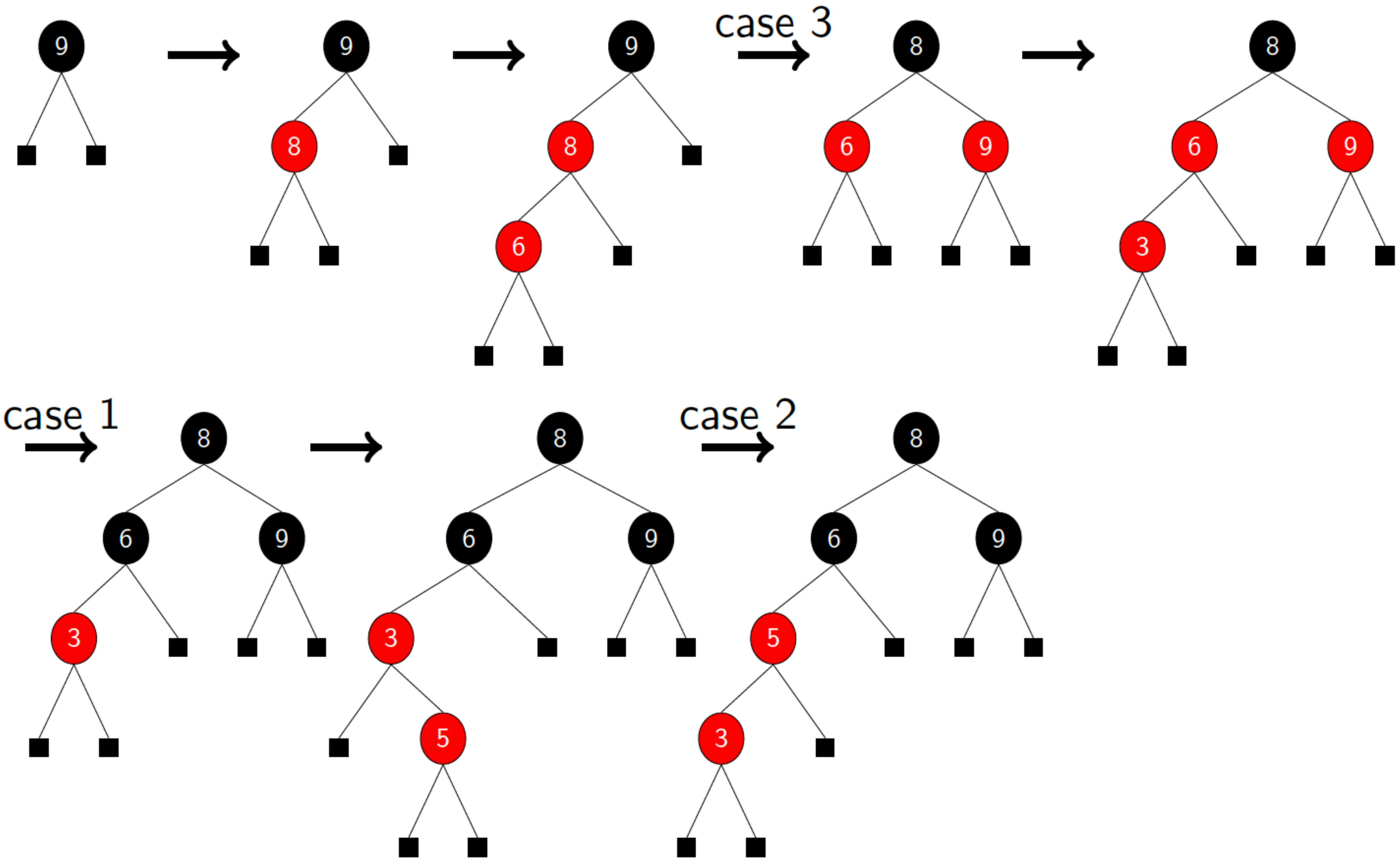
Example Insertions (9, 8, 6, 3, 5)



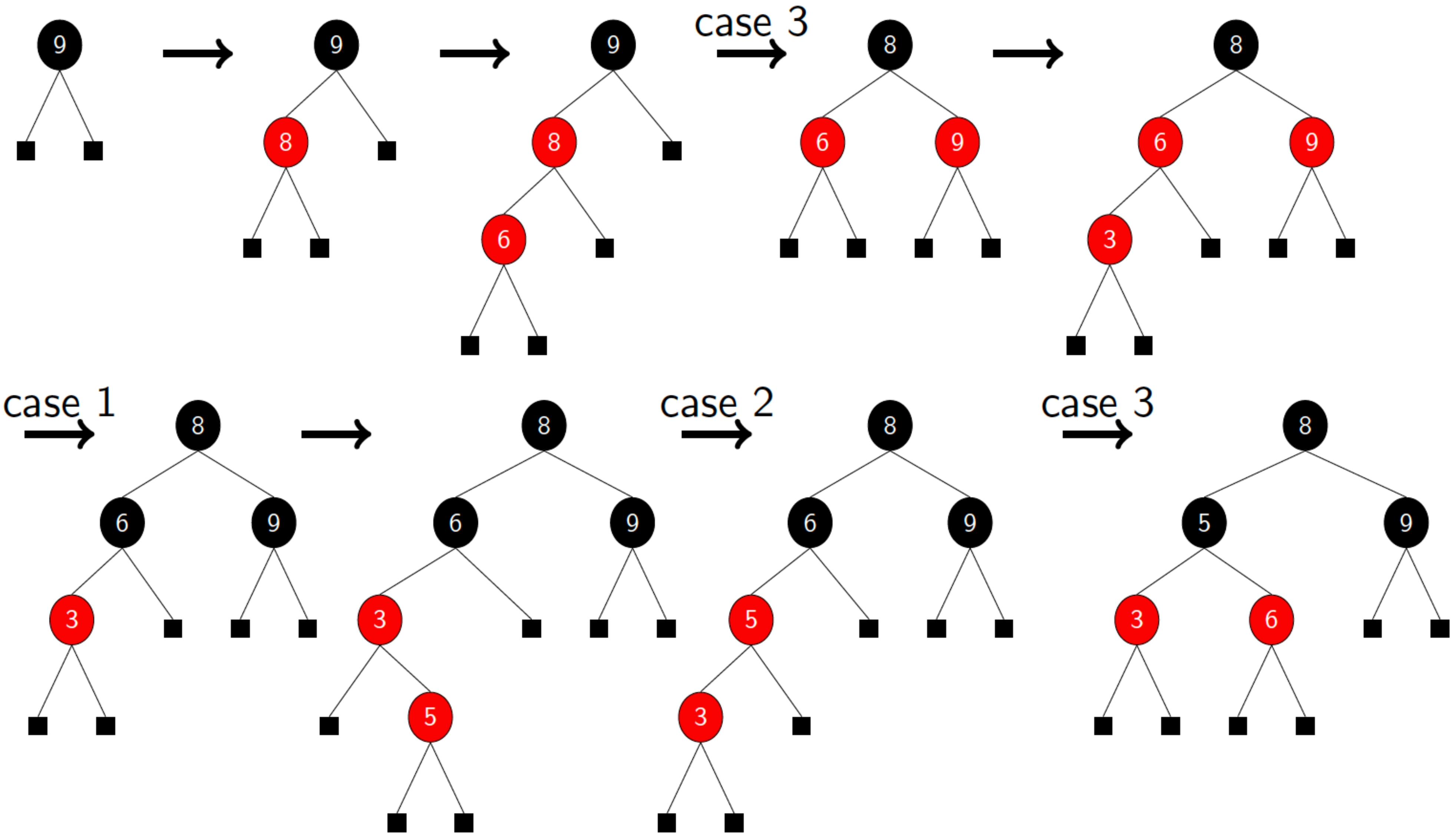
Example Insertions (9, 8, 6, 3, 5)



Example Insertions (9, 8, 6, 3, 5)



Example Insertions (9, 8, 6, 3, 5)



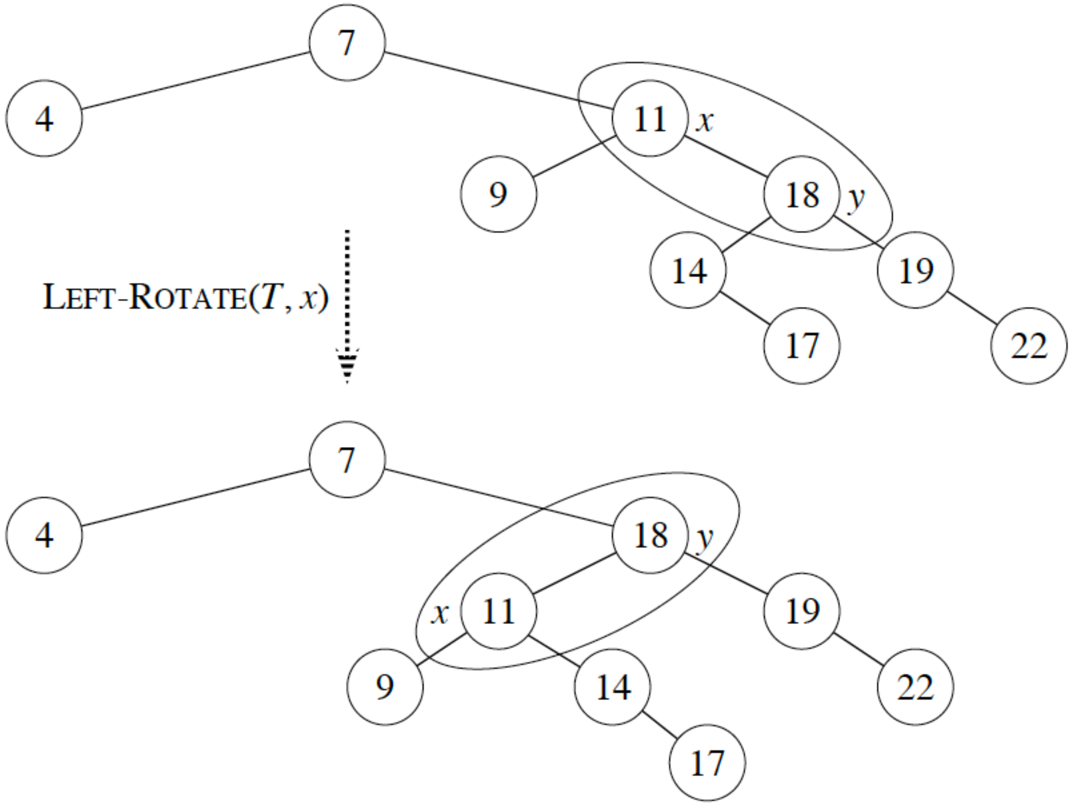
Suggested reading

Today's lecture covered sections 13.2 and 13.3.

Solutions

Class challenge

Perform an RBT_Rotate_left(T, x) operation:



Node colours omitted for convenience.