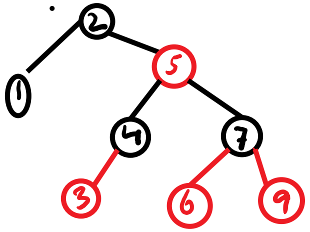


CS 201 Data Structure II (L2 / L5)

Red-Black Tree

Chapter 13 (Introduction to Algorithms, Coreman)

Muhammad Qasim Pasta

qasim.pasta@sse.habib.edu.pk

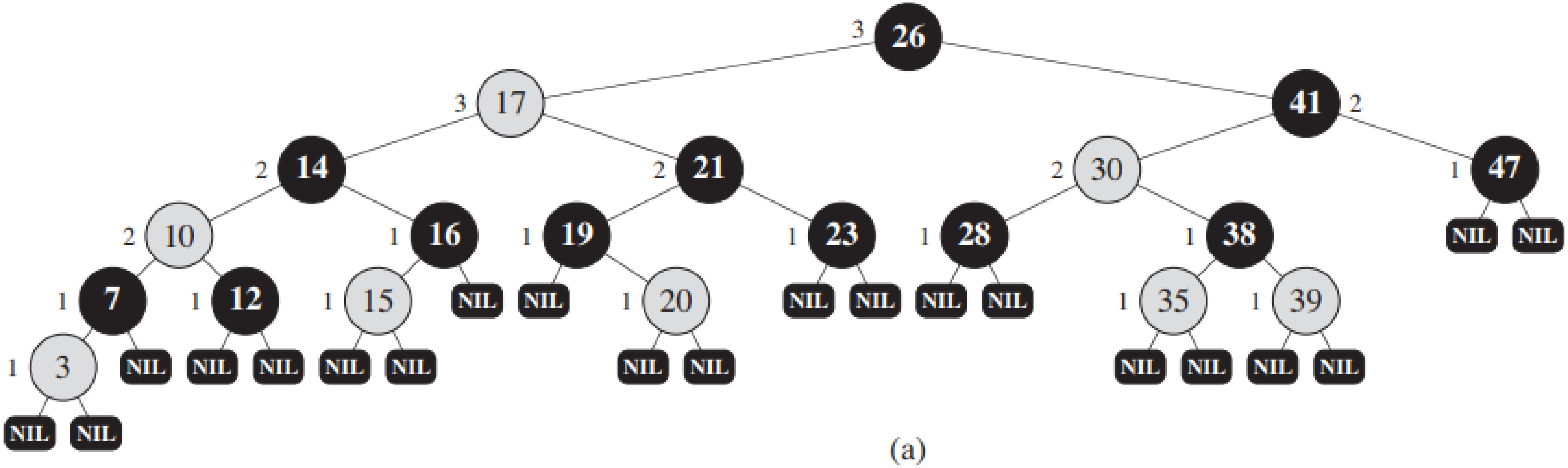
Red-Black Tree

- Self-balancing binary search tree like AVL, treaps …
- Storing **n** elements with at most height of $2\log n$
  - Skiplists and treaps have expected O(log n) [randomization]
- Add(x) and Remove(x) operations run in $O(\log n)$ ***worst-case*** time.
  - Scapegoat run in O(log n) amortized time
- Amortized number of rotations performed during in Add(x) or Remove(x) operations is constant
  - Skiplists and treaps have this property but in expected terms.
- You should find: AVL vs RedBlack

Red-Black Tree

- Each node has extra information: its color, either RED or BLACK
- Tree satisfies the following properties:
  - Every node is either red or black
  - The root is black
  - Every leaf (NIL) is black
  - If a node is red, then both its children are black (no consecutive red)
  - There are same number of black nodes on every root to leaf path (black height)
    - all simple paths from the node descendant leaves contain the same number of black node

Example:



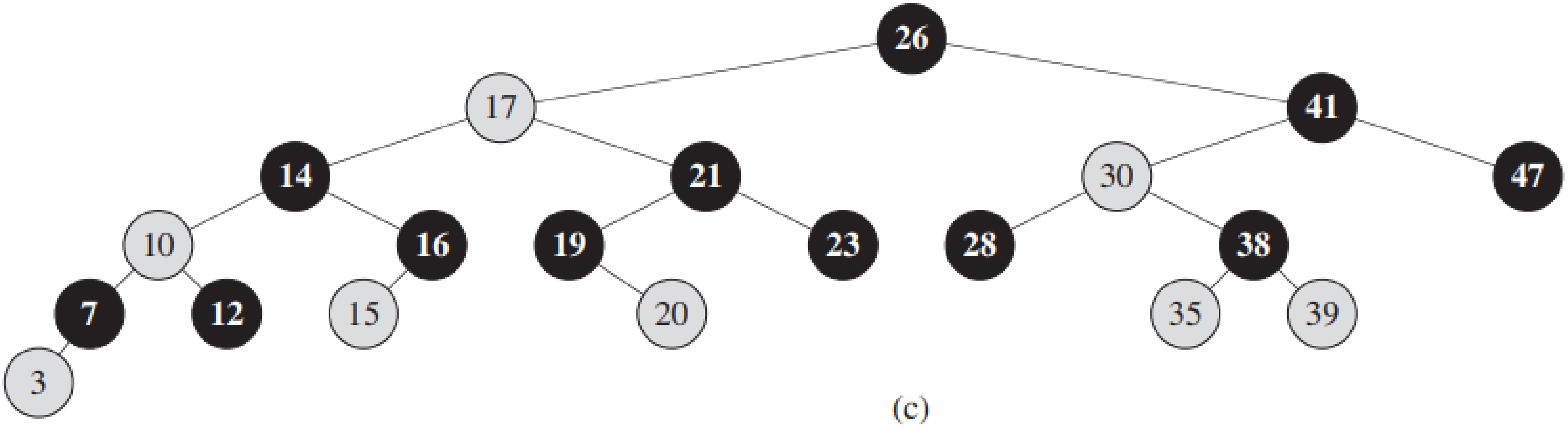
(a)

Black Height of path 26,17,14,10,7,3,Nil:
Black Height of path 26,41,47,Nil:

Black Height of 26: ?
Black Height of 41: ?
Black Height of 17: ?

Fig 13.1 from Introduction of Algorithms, Coreman

Example:



Black Height of path 26,17,14,10,7,3,Nil:
Black Height of path 26,41,47,Nil:

Black Height of 26: ?
Black Height of 41: ?
Black Height of 17: ?

Fig 13.1 from Introduction of Algorithms, Coreman

Rotations: recap from Binary trees:

- We can fix heap property by performing rotations
- Rotate Right = make the left child as a parent
- Rotate Left = make the right child as a parent
- Decrease (/increase) the depth by one

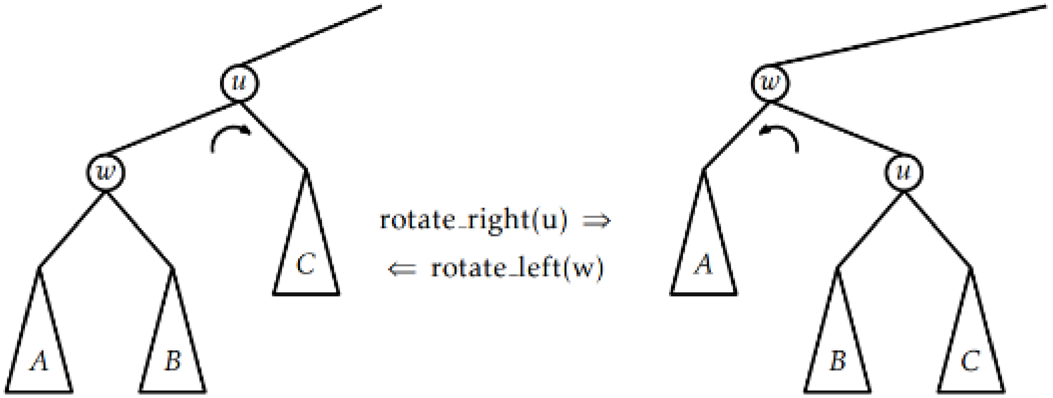
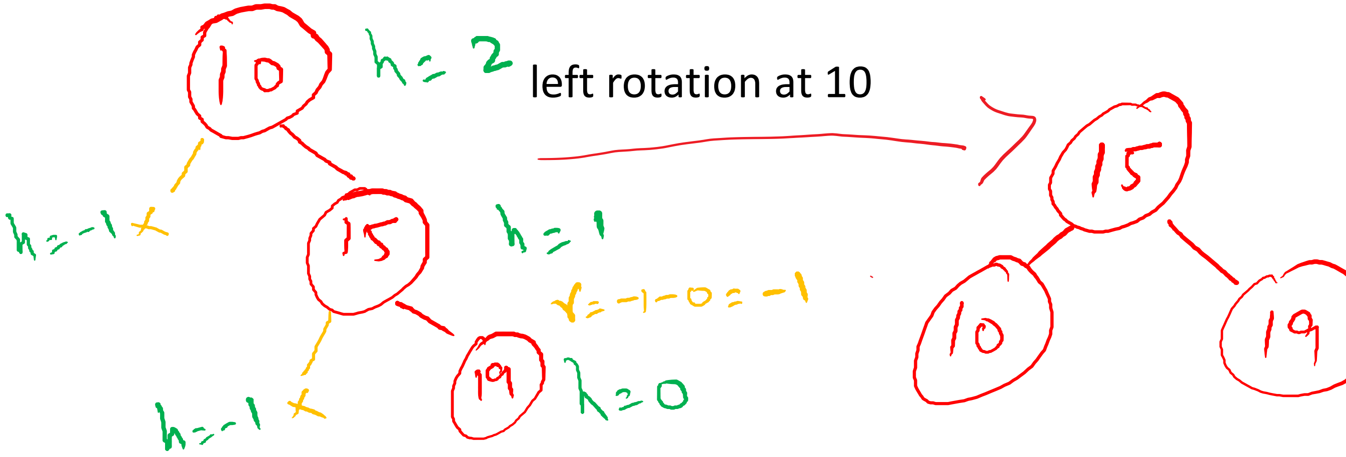


Figure 7.6: Left and right rotations in a binary search tree.


left rotation at 10

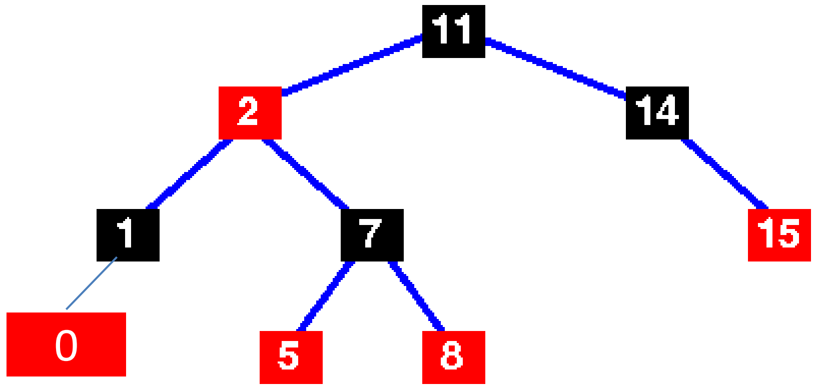
Insertion in RB-Tree:

Case #s are not in sequence as synchronized with respect to the book Introduction to Algorithms, Cormen for clarity purposes.

- Insert an element as per standard-BST insertion with color red
- If newly insert node is the root node then change color to black
- If parent of newly node is not black then:
- If uncle is RED [C1]
- If uncle is BLACK
    - Four possible cases:
        - Left – Left: newly element is LEFT element, and parent is also LEFT element of grandparent [C3]
        - Left – Right: newly element is LEFT element, and parent is RIGHT element of grandparent [C2]
        - Right – Right: newly element is RIGHT element, and parent is also RIGHT element of grandparent
        - Right – Left: newly element is RIGHT element, and parent is LEFT element of grandparent

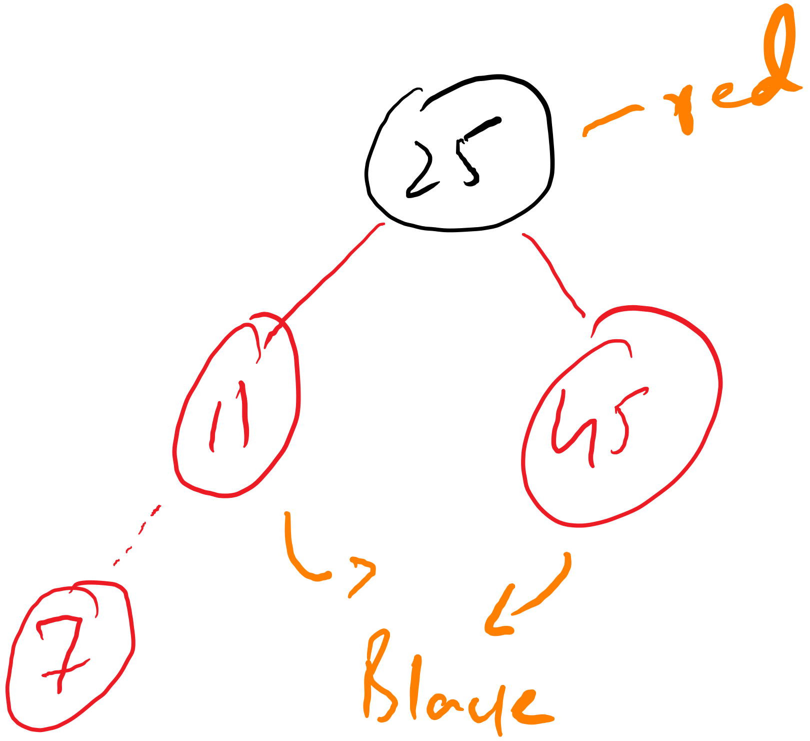
Insertion: parent is BLACK

- Add(0): no violation



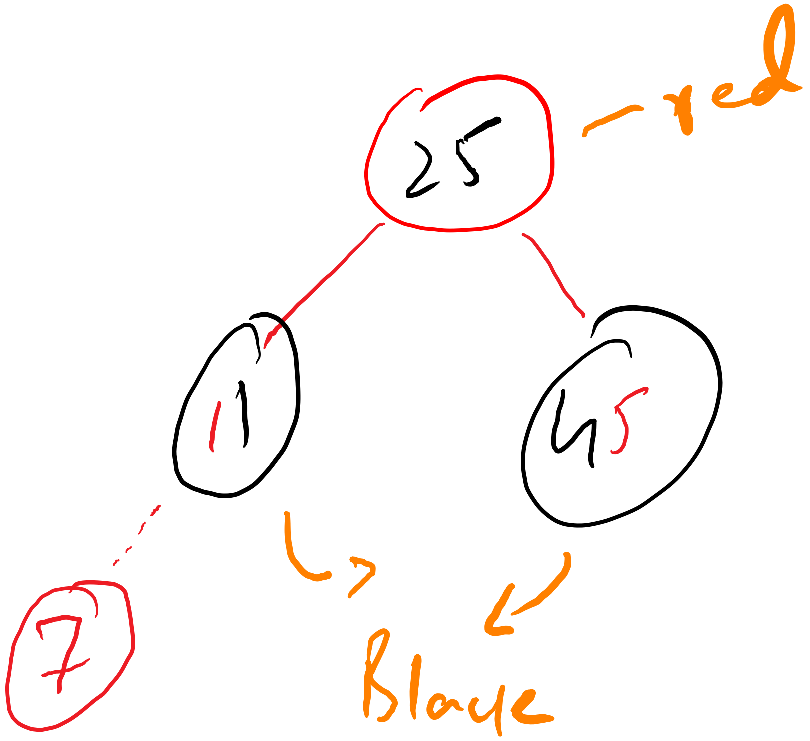
Insertion: parent is RED & uncle is RED

- Case 1: Regardless uncle is right or left child of grand parent
- Change color of parent and uncle to BLACK
- Change color of grand parent to RED



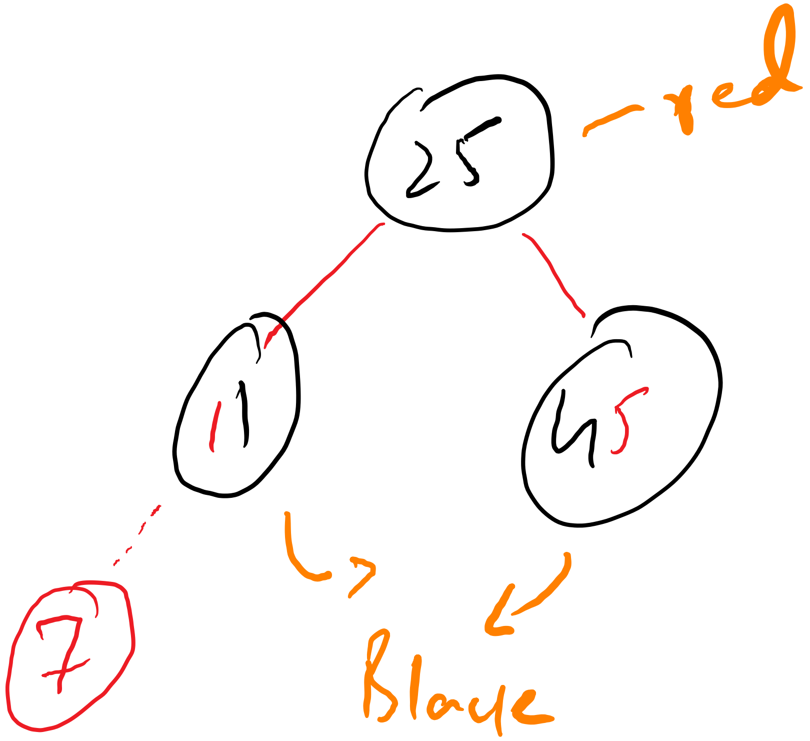
Insertion: parent is RED & uncle is RED

- Case 1: Regardless uncle is right or left child of grand parent
- Change color of parent and uncle to BLACK
- Change color of grand parent to RED



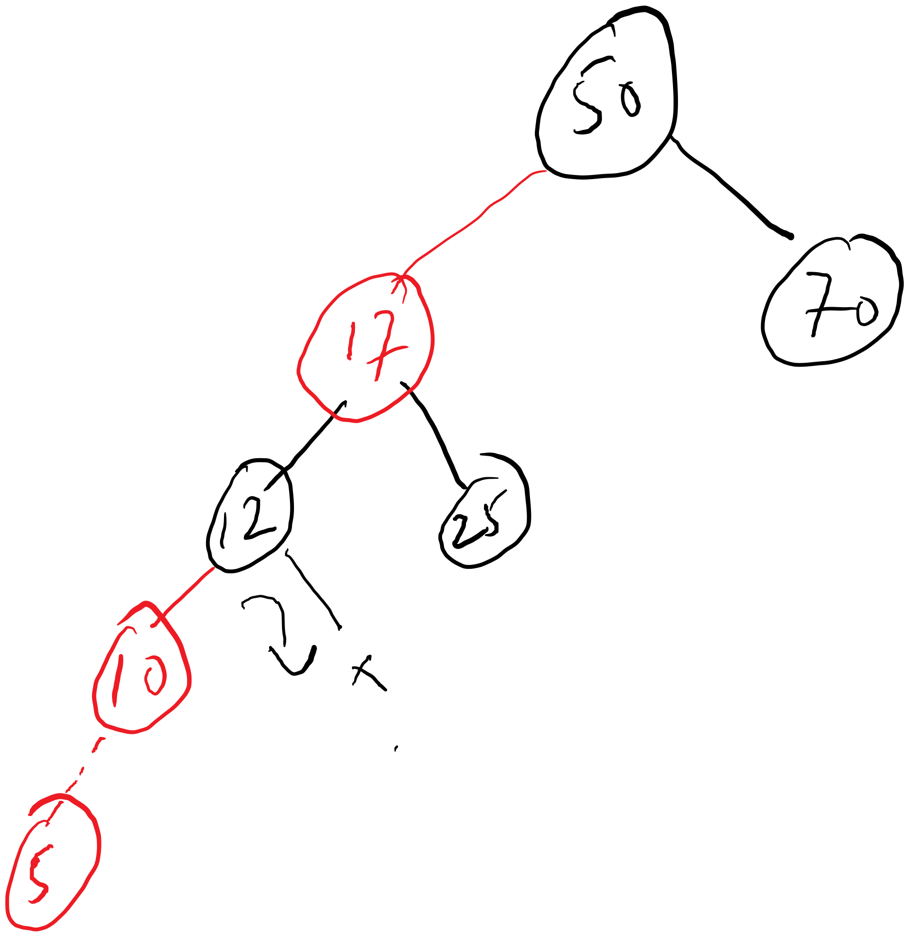
Insertion: parent is RED & uncle is RED

- Case 1: Regardless uncle is right or left child of grand parent
- Change color of parent and uncle to BLACK
- Change color of grand parent to RED



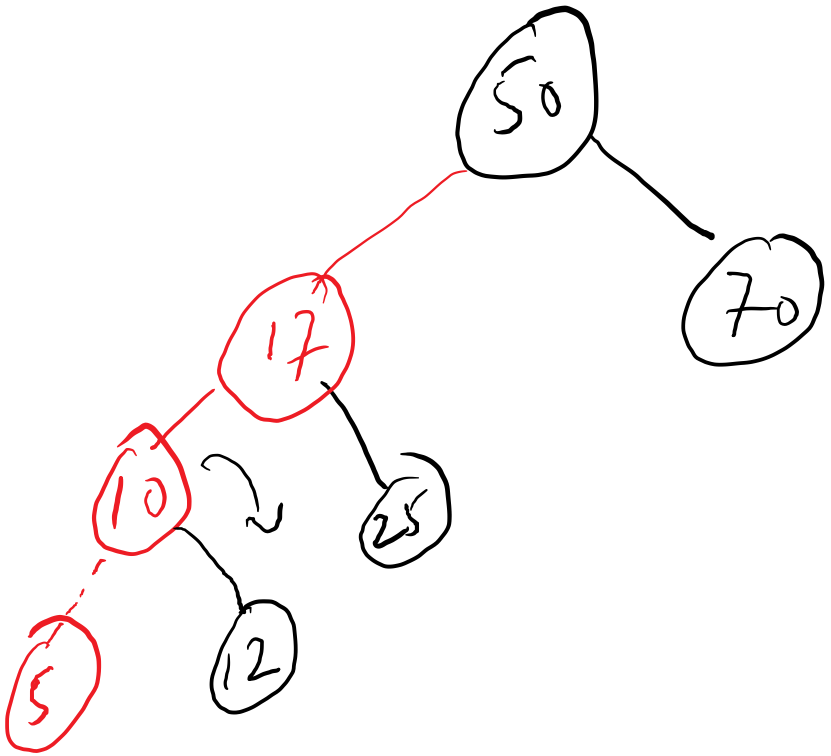
Insertion: parent is RED uncle is Black

- Case 3 (Left-Left):
  - Uncle is right child of grand parent
  - OR parent of newly added node is a left child
  - Newly node added as a left child
- Right Rotate grandparent
- Swap color of parent and grandparent
  - Parent must be red (violation) => Change to BLACK
  - Grandparent must be black (no prior violation) => Change to RED



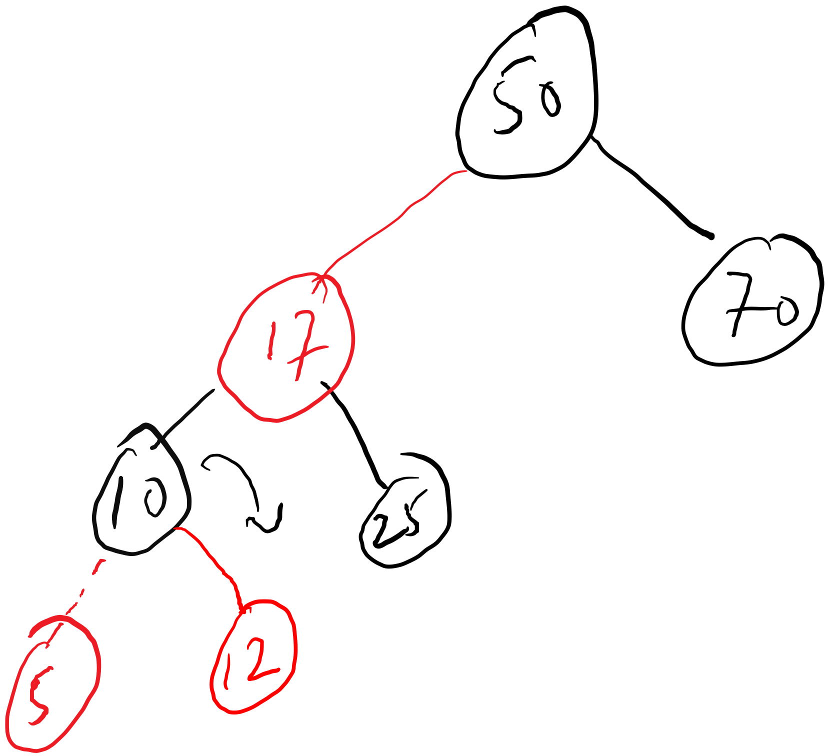
Insertion: parent is RED uncle is Black

- Case 3 (Left-Left):
  - Uncle is right child of grand parent
  - OR parent of newly added node is a left child
  - Newly node added as a left child
- Right Rotate grandparent
- Swap color of parent and grandparent
  - Parent must be red (violation) => Change to BLACK
  - Grandparent must be black (no prior violation) => Change to RED



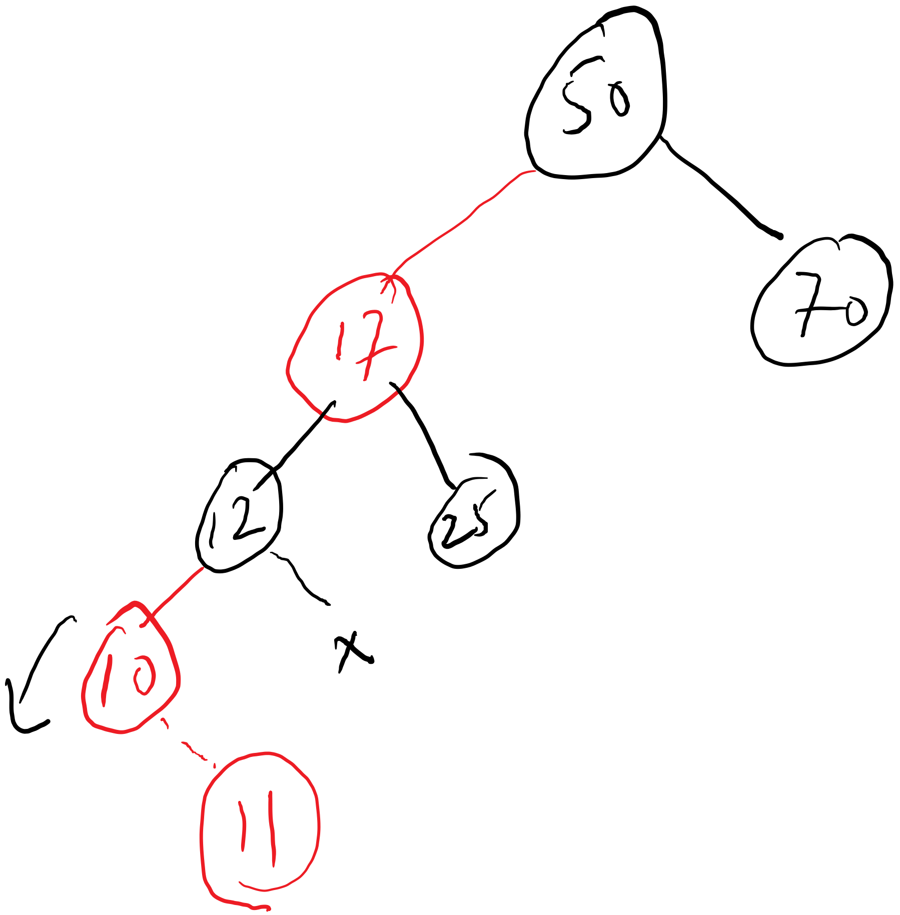
Insertion: parent is RED uncle is Black

- Case 3 (Left-Left):
  - Uncle is right child of grand parent
  - OR parent of newly added node is a left child
  - Newly node added as a left child
- Right Rotate grandparent
- Swap color of parent and grandparent
  - Parent must be red (violation) => Change to BLACK
  - Grandparent must be black (no prior violation) => Change to RED



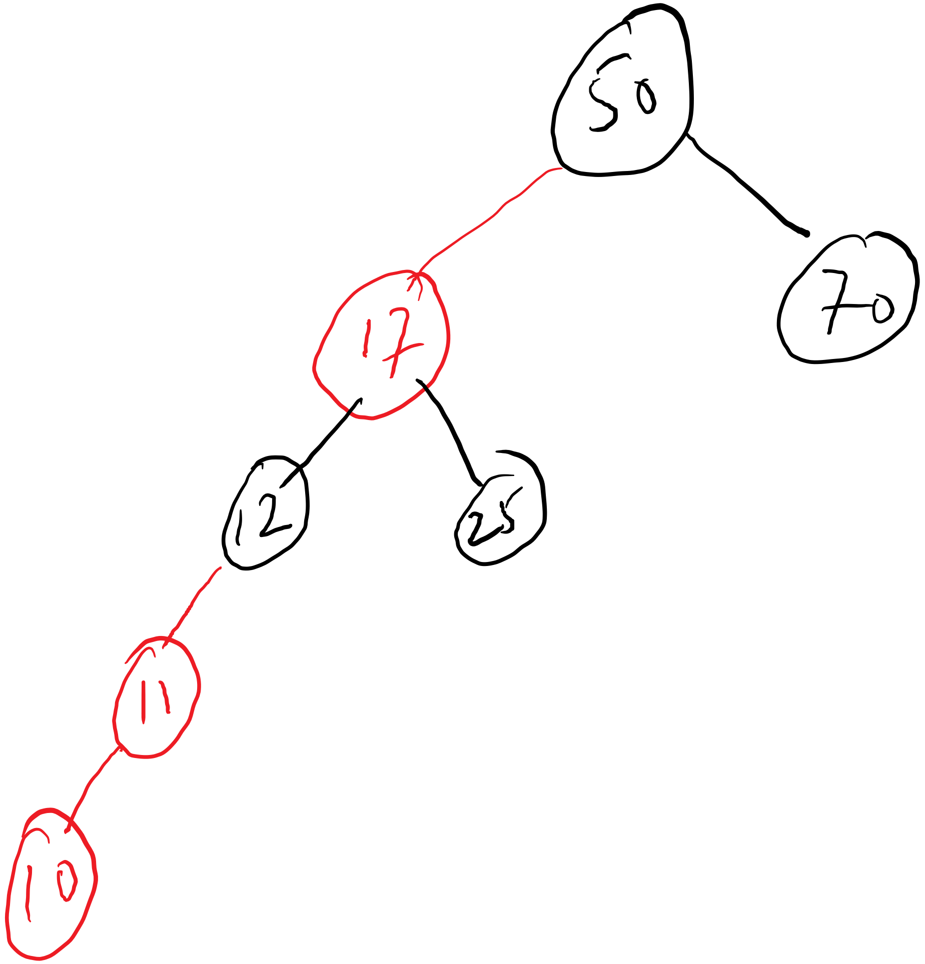
Insertion: parent is RED & uncle is Black

- Case 2 (Left-Right):
  - Uncle is right child of grand parent
  - OR parent of newly added node is a left child
  - Newly node added as a **RIGHT** child
- Left Rotate Parent
- Rest is same as Case 3
  - Right Rotate grandparent
  - Swap color of parent (new) and grandparent



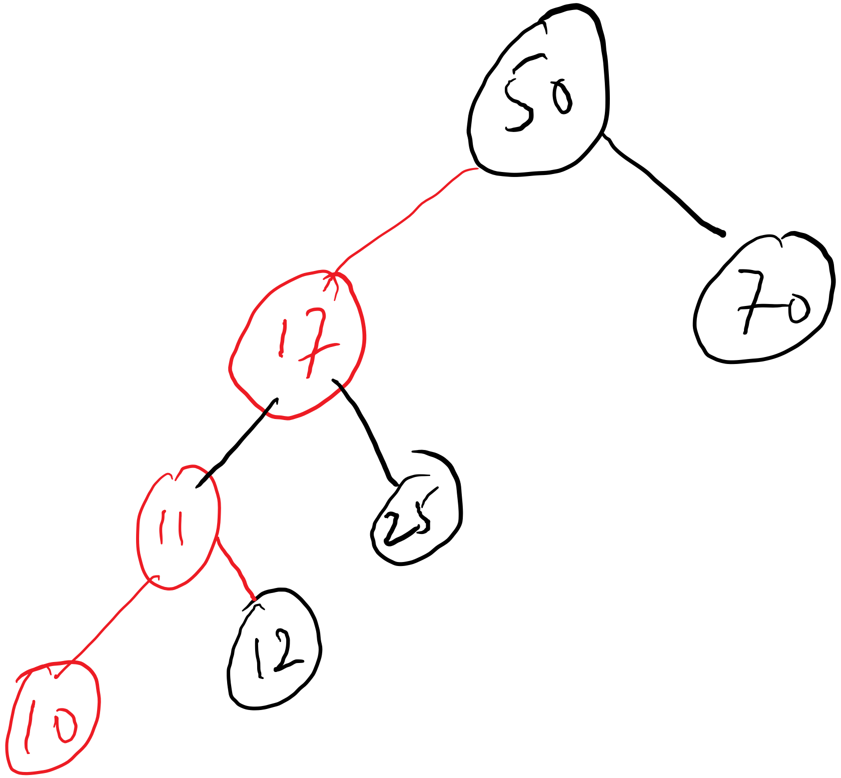
Insertion: parent is RED & uncle is Black

- Case 2 (Left-Right):
  - Uncle is right child of grand parent
  - OR parent of newly added node is a left child
  - Newly node added as a **RIGHT** child
- Left Rotate Parent
- Rest is same as Case 3
  - Right Rotate grandparent
  - Swap color of parent (new) and grandparent



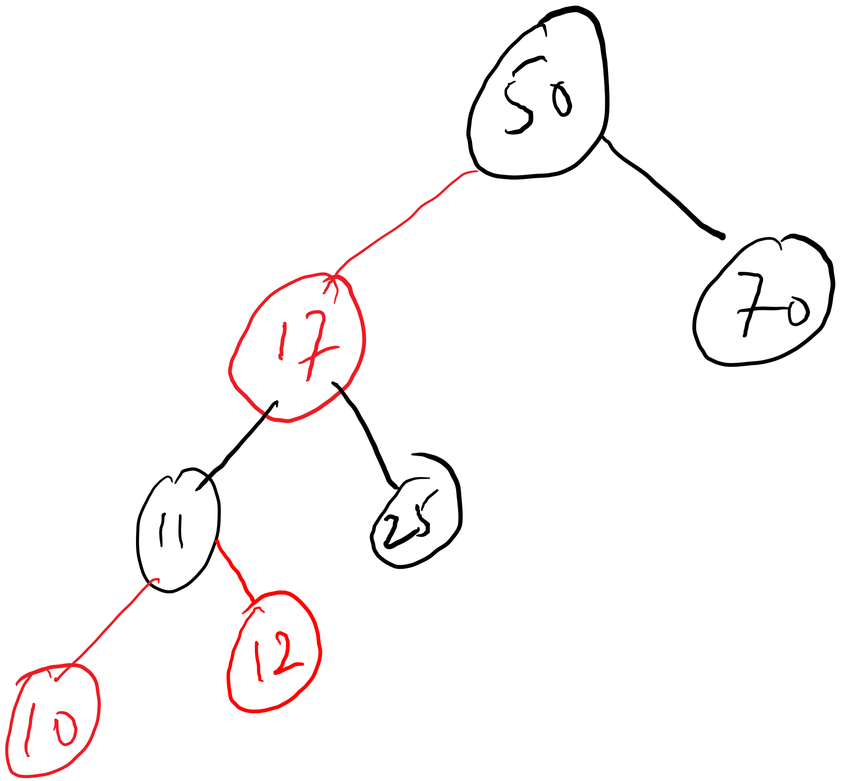
Insertion: parent is RED & uncle is Black

- Case 2 (Left-Right):
  - Uncle is right child of grand parent
  - OR parent of newly added node is a left child
  - Newly node added as a **RIGHT** child
- Left Rotate Parent
- Rest is same as Case 3
  - Right Rotate grandparent
  - Swap color of parent (new) and grandparent



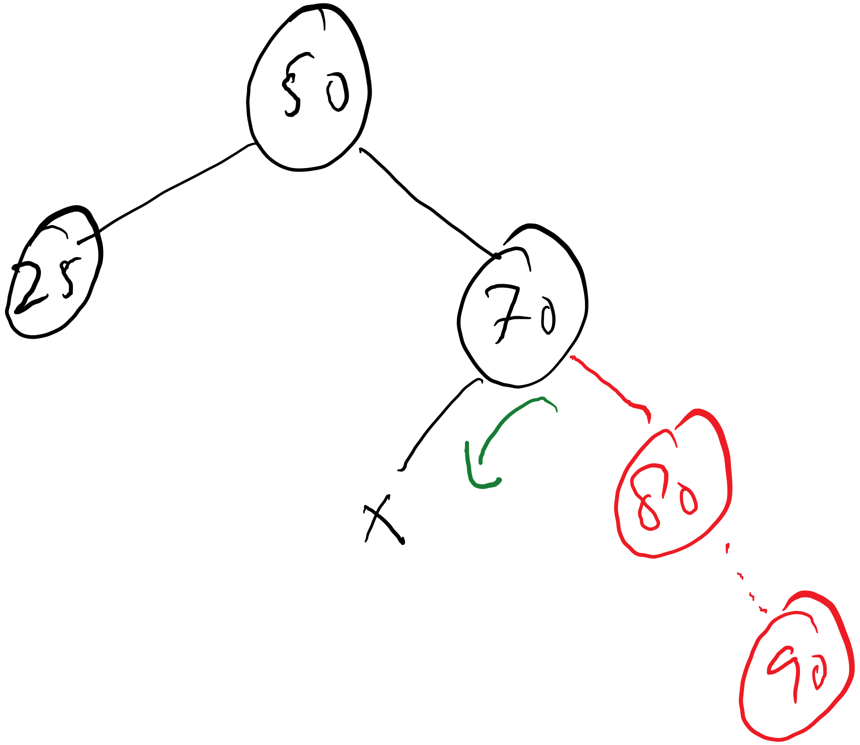
Insertion: parent is RED & uncle is Black

- Case 2 (Left-Right):
  - Uncle is right child of grand parent
  - OR parent of newly added node is a left child
  - Newly node added as a **RIGHT** child
- Left Rotate Parent
- Rest is same as Case 3
  - Right Rotate grandparent
  - Swap color of parent (new) and grandparent



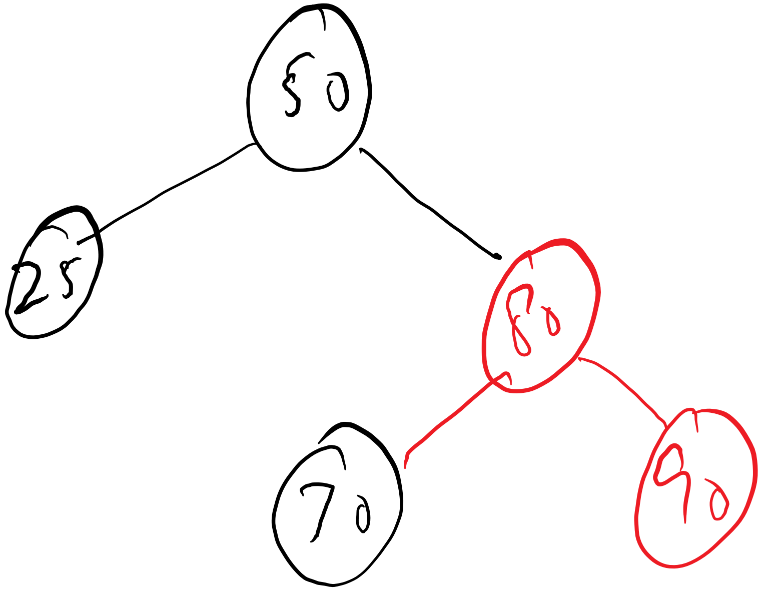
Insertion: parent is RED & uncle is Black

- Case 3’ (Right-Right):
  - Uncle is left child of grand parent
  - OR parent of newly added node is a right child
  - Newly node added as a right child
- **Left** Rotate grandparent
- Swap color of parent and grandparent
  - Parent must be red (violation) => Change to BLACK
  - Grandparent must be black (no prior violation) => Change to RED



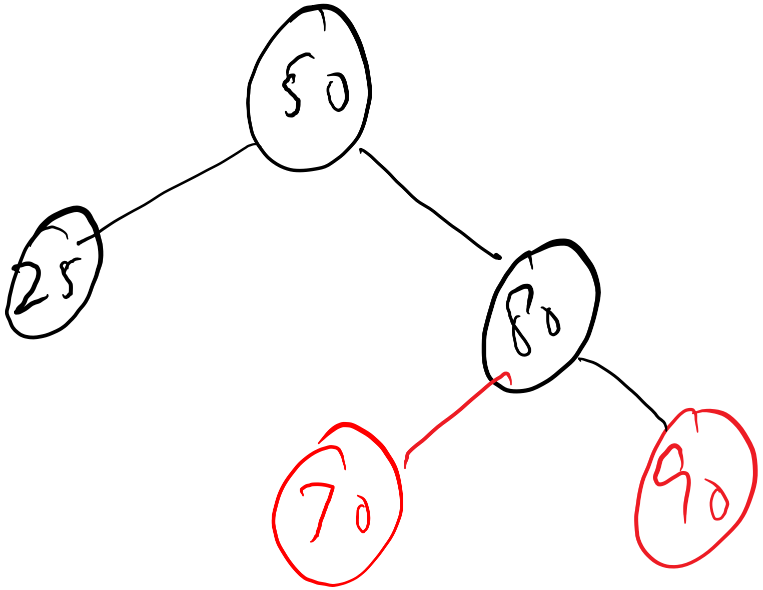
Insertion: parent is RED & uncle is Black

- Case 3’ (Right-Right):
  - Uncle is left child of grand parent
  - OR parent of newly added node is a right child
  - Newly node added as a right child
- **Left** Rotate grandparent
- Swap color of parent and grandparent
  - Parent must be red (violation) => Change to BLACK
  - Grandparent must be black (no prior violation) => Change to RED



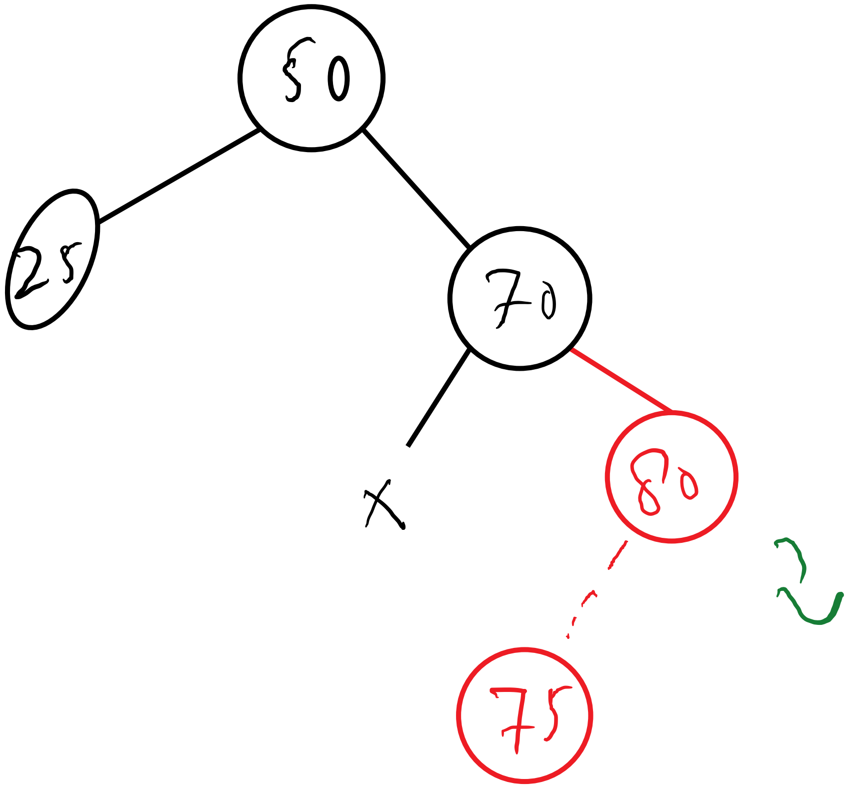
Insertion: parent is RED & uncle is Black

- Case 3’ (Right-Right):
  - Uncle is left child of grand parent
  - OR parent of newly added node is a right child
  - Newly node added as a right child
- **Left** Rotate grandparent
- Swap color of parent and grandparent
  - Parent must be red (violation) => Change to BLACK
  - Grandparent must be black (no prior violation) => Change to RED



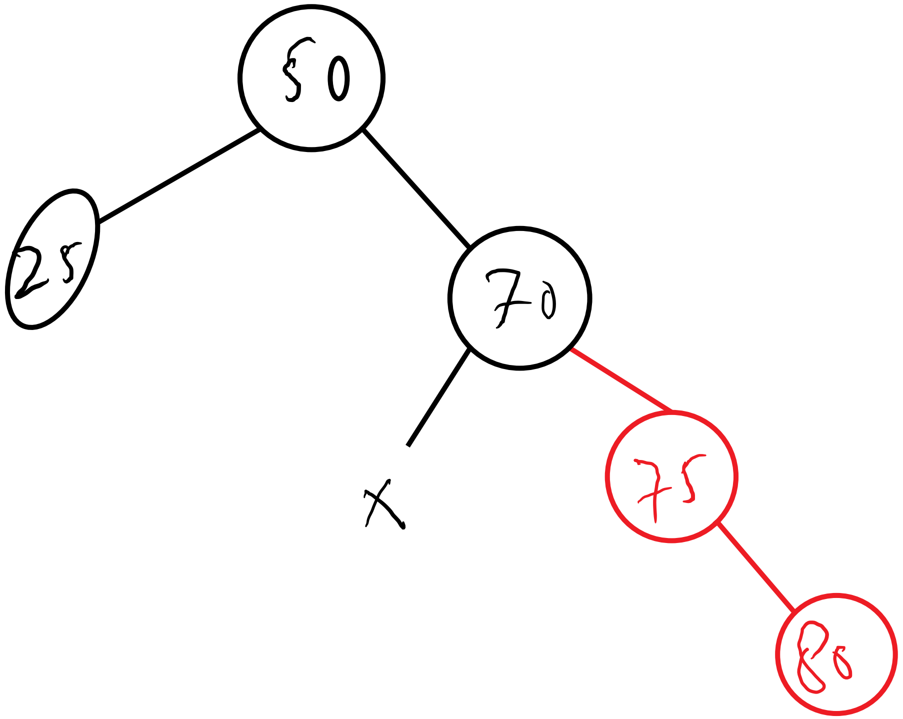
Insertion: parent is RED & uncle is Black

- Case 2’ (Right - Left):
  - Uncle is left child of grand parent
  - OR parent of newly added node is a right child
  - Newly node added as a **LEFT** child
- Right Rotate Parent
- Rest is same as Case 3’
  - Left Rotate grandparent
  - Swap color of parent (new) and grandparent



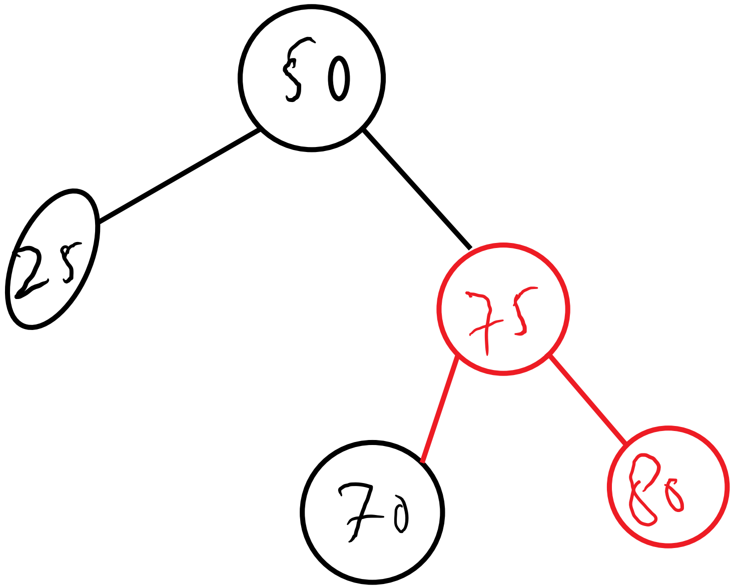
Insertion: parent is RED & uncle is Black

- Case 2’ (Right - Left):
  - Uncle is left child of grand parent
  - OR parent of newly added node is a right child
  - Newly node added as a **LEFT** child
- Right Rotate Parent ✓
- Rest is same as Case 3’
  - Left Rotate grandparent
  - Swap color of parent (new) and grandparent



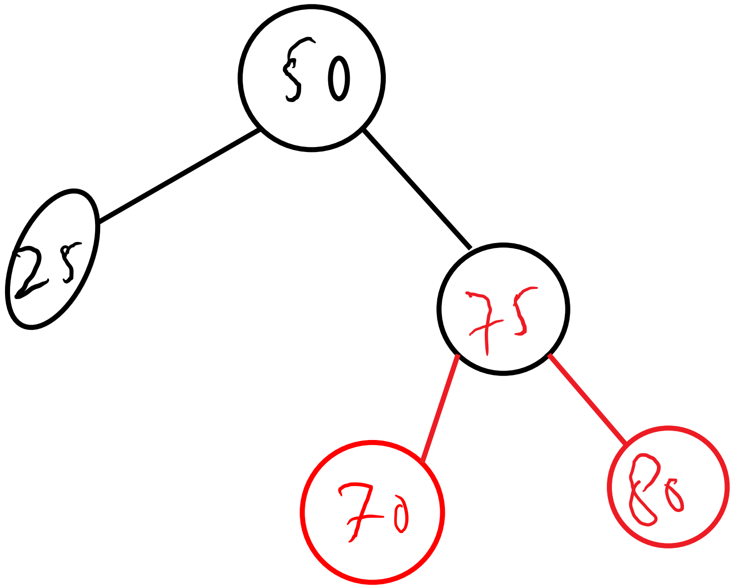
Insertion: parent is RED & uncle is Black

- Case 2' (Right - Left):
  - Uncle is left child of grand parent
  - OR parent of newly added node is a right child
  - Newly node added as a **LEFT** child
- Right Rotate Parent ✓
- Rest is same as Case 3'
  - Left Rotate grandparent ✓
  - Swap color of parent (new) and grandparent



Insertion: parent is RED & uncle is Black

- Case 2’ (Right - Left):
  - Uncle is left child of grand parent
  - OR parent of newly added node is a right child
  - Newly node added as a **LEFT** child
- Right Rotate Parent ✓
- Rest is same as Case 3’
  - Left Rotate grandparent ✓
  - Swap color of parent (new) and grandparent ✓

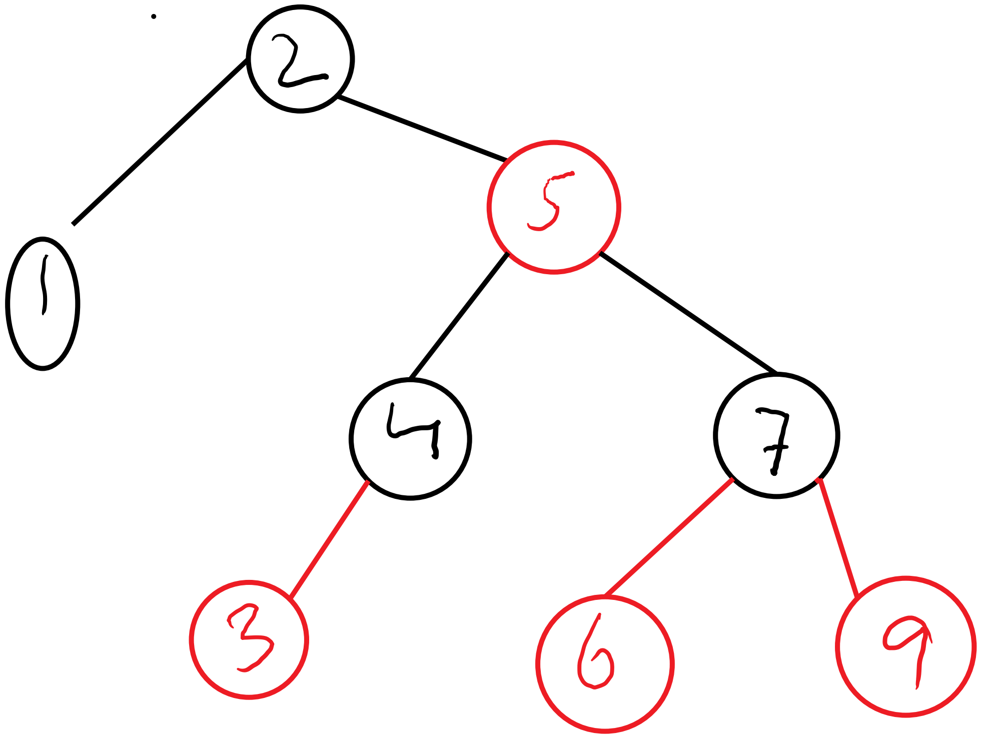


Practice!

- Create RB-Trees for the following: 2,1,4,5,9,3,6,7

Practice!

- Create RB-Trees for the following: 2,1,4,5,9,3,6,7



Insertion: Fix-up to root!

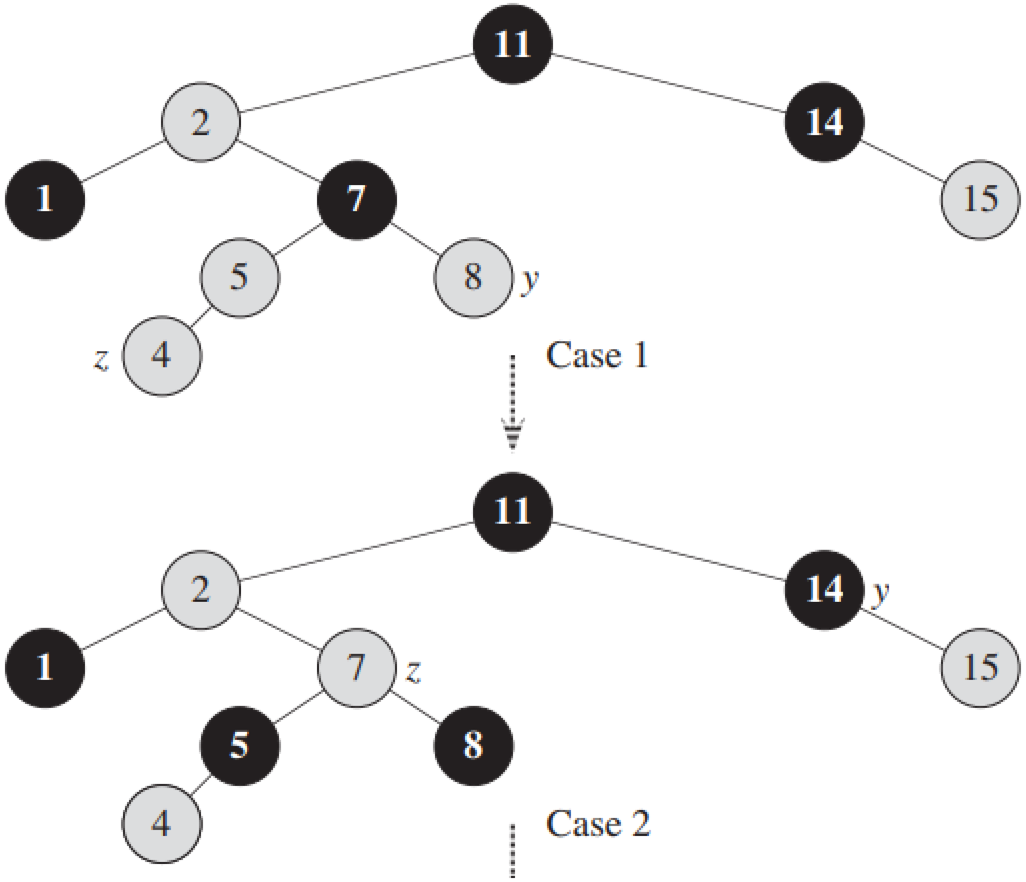
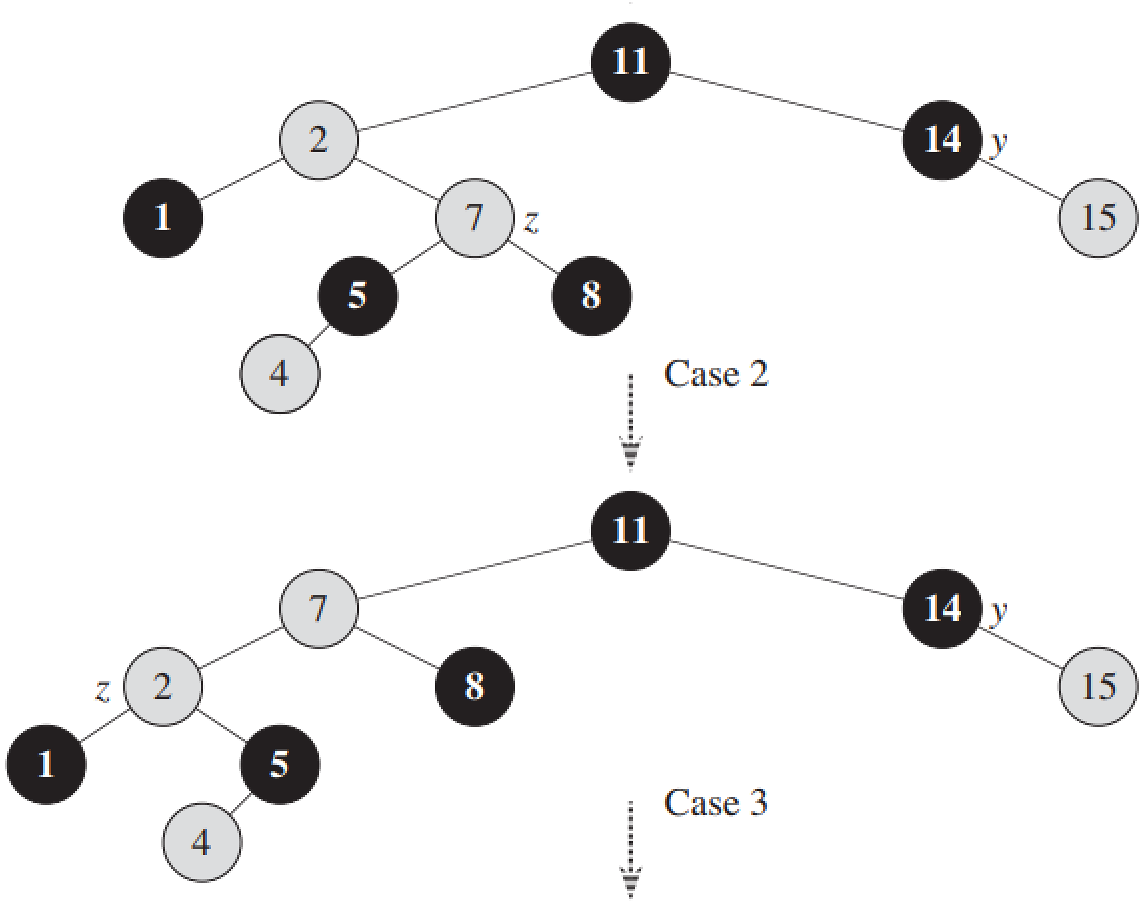
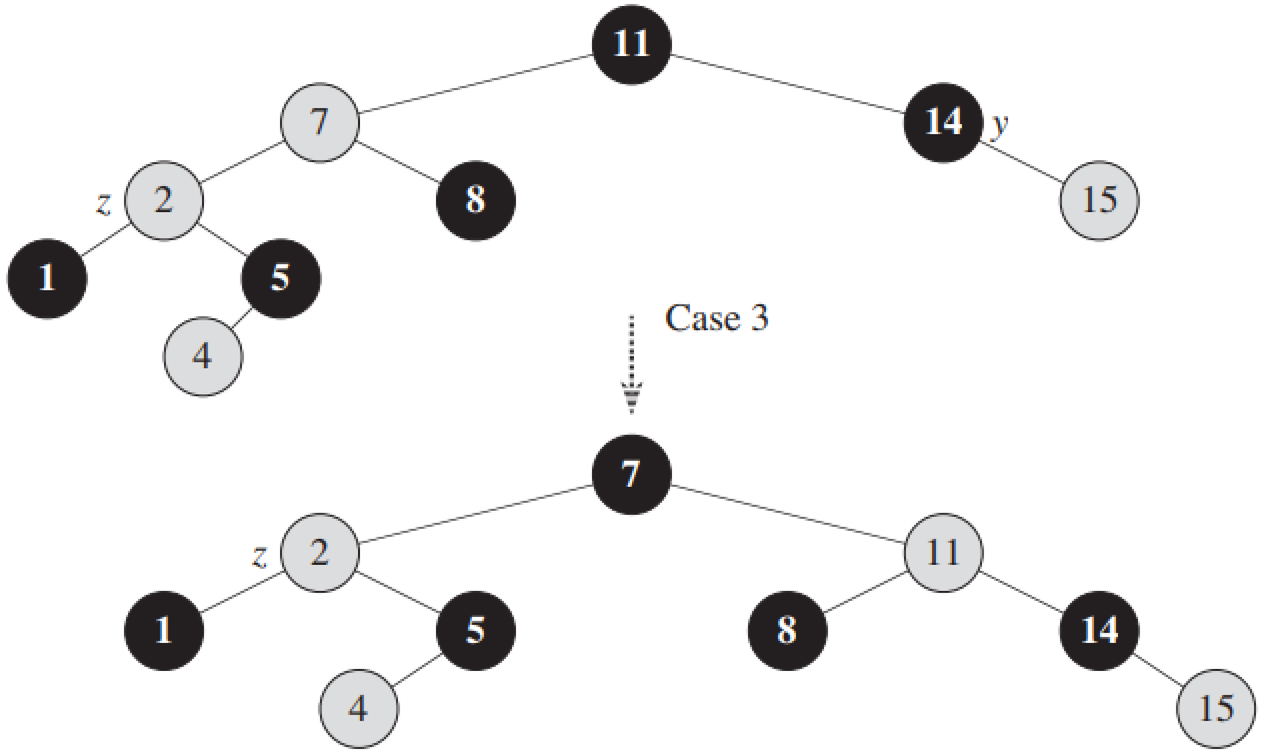


Fig 13.4 from Introduction of Algorithms, Coreman

Insertion: Fix-up to root!



Insertion: Fix-up!



Insertion: Analysis

- Standard Insertion: $O(\log n)$
- Fix-up:
  - Fix-up requires if Case 1 occurs again and again
  - Case 1: no rotations
  - Maximum Fix-up class: $O(\log n)$
  - Case 2 or Case 3 will not require further fix-up
  - Constant rotations: 1 or 2 rotations