National University of Computer & Emerging Sciences

Red Black Trees

Red-Black Trees (Intro)

- BSTs perform dynamic set operations such as SEARCH, INSERT, DELETE etc in time related to tree height.
- If the tree height is large, performance may be no better than a linked list, for large n.
- RBTs are “balanced” in order to guarantee *better* worst case time for set dynamic operations

Red-Black Trees (Intro)

- What is an AVL-Tree?
- Both AVL trees and red-black trees are self-balancing binary search trees
- AVL trees are more rigidly balanced than red-black trees
  - leading to slower insertion and removal but faster retrieval. HOW?

Red-Black Trees (Intro)

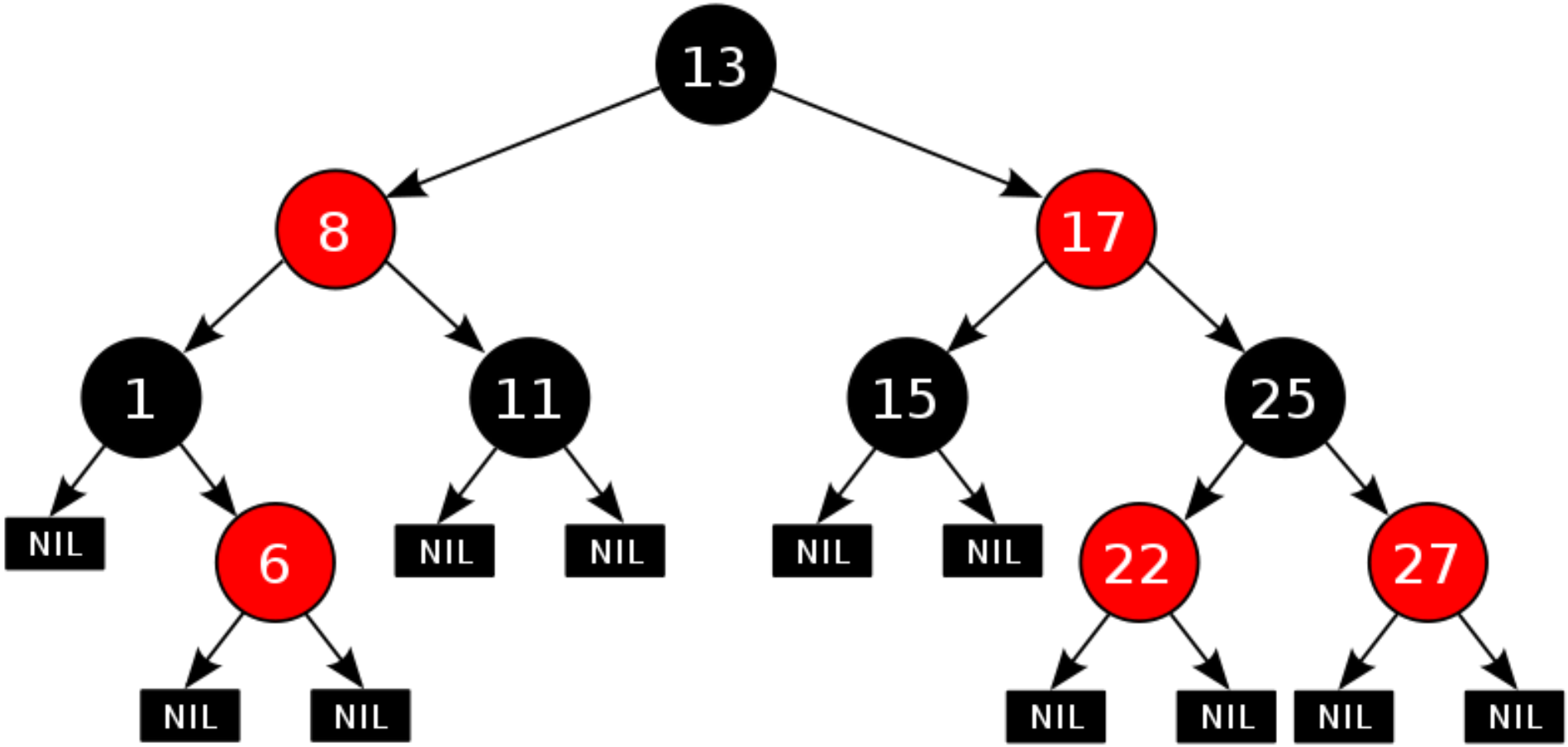
- A RBT is a binary search tree where each node is colored red or black
- No path from root to leaf is twice as long as any other?
- A node contains: *color, value, left, right*
- A NIL is a leaf, all other nodes are internal nodes.

Red-Black Trees (Properties)

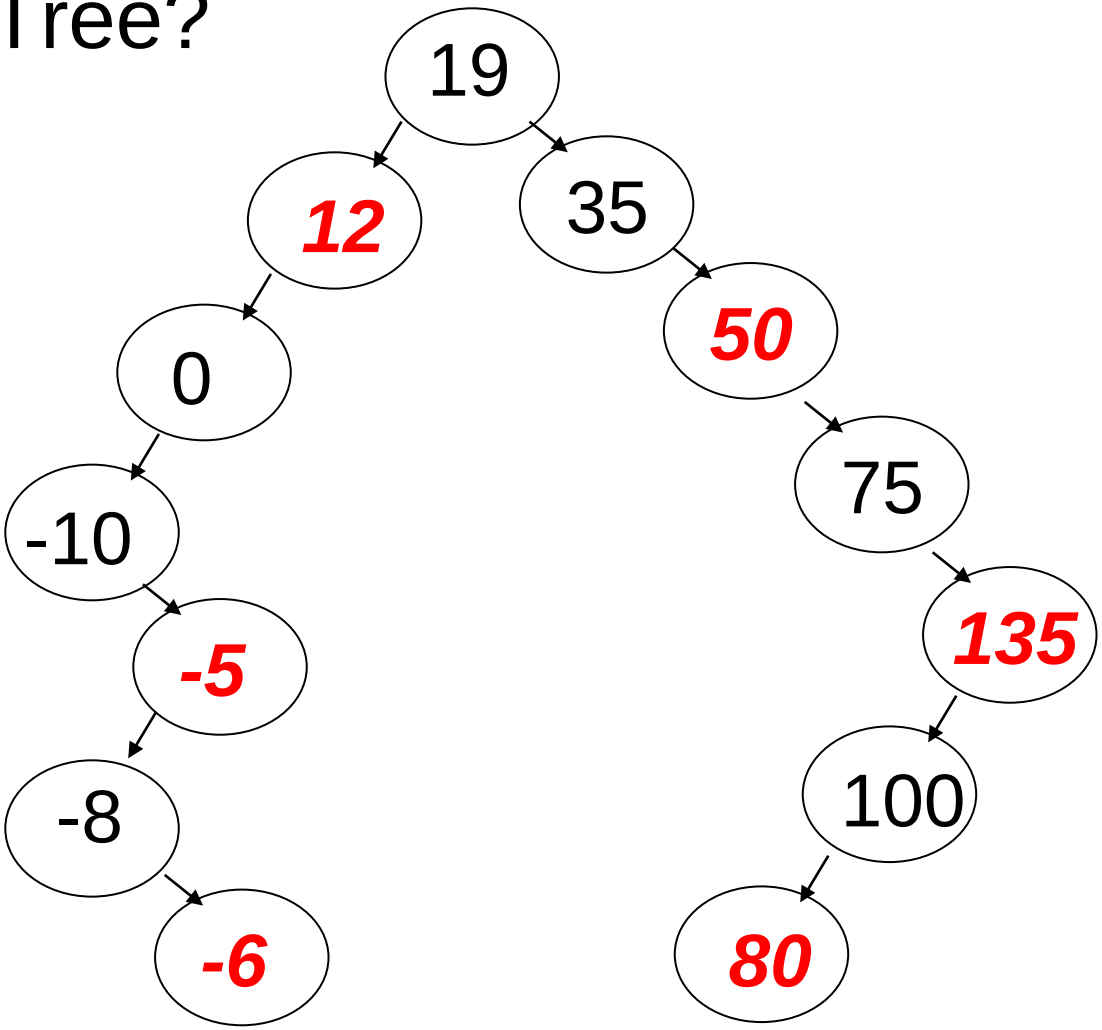
A binary search tree is a red-black tree if:

1. Every node is either red or black
2. The root is black
3. Every leaf (NIL) is black
4. A red node's children are black
5. For each node, all paths from a node to descendant leaves contain the same number of black nodes (balance condition)

Red Black Tree?



Red Black Tree?



Implications of the Rules

- If a Red node has any children, it must have two children and they must be Black. (Why?)

- If a Black node has only one child that child must be a Red leaf. (Why?)
  - Yes, all the three underlined points must be true!

NULL Leaf nodes?

- Their explicit representation is not required
  - a null child pointer can be used
  - but it simplifies some algorithms if the leaves are explicit nodes

Rotations

- The operations TREE-INSERT and TREE-DELETE modify the tree, they may violate the red-black properties.

- To restore these, we
  - Change color of nodes
  - Change pointer structure (rotations, same as AVL tree)
    - Single left rotation and single right rotation only

Rotations

- Left-Rotation on a node *x*:
  - Right child *y* is not *nil[T]*
  - *Left rotation "pivots" around the link from x to y*
  - i.e., it makes *y* the new root of the subtree, with *x* as *y's* left child and *y's* left child as *x's* right child.

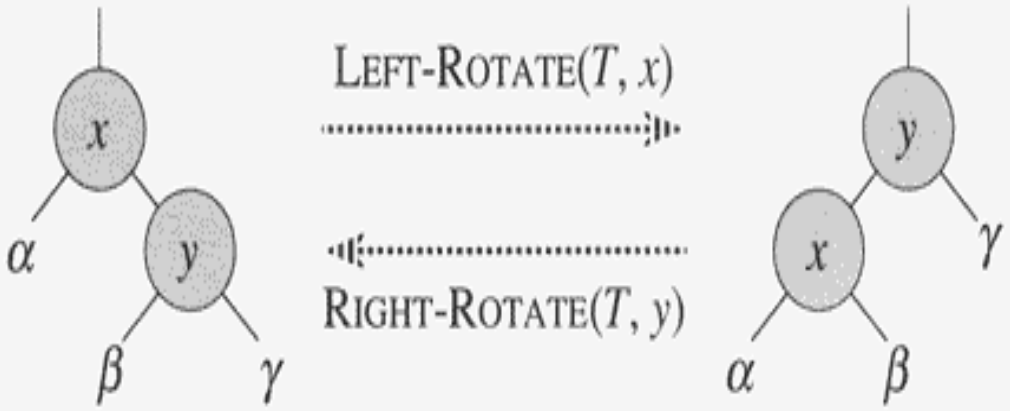


Figure 13.2 The rotation operations on a binary search tree. The operation LEFT-ROTATE(T, x) transforms the configuration of the two nodes on the left into the configuration on the right by changing a constant number of pointers. The configuration on the right can be transformed into the configuration on the left by the inverse operation RIGHT-ROTATE(T, y). The letters α, β, and γ represent arbitrary subtrees. A rotation operation preserves the binary-search-tree property: the keys in α precede $key[x]$, which precedes the keys in β, which precede $key[y]$, which precedes the keys in γ.

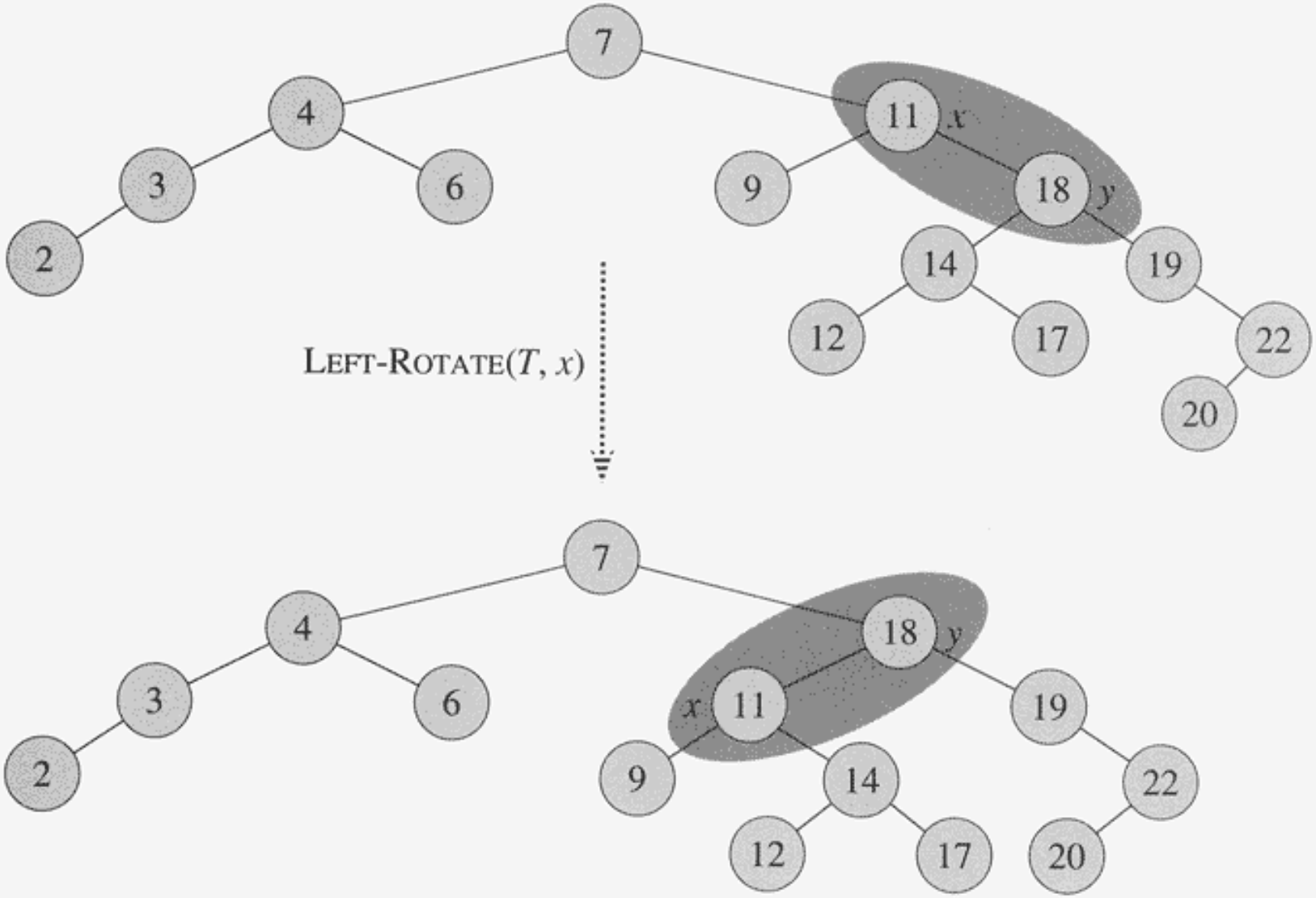


Figure 13.3 An example of how the procedure LEFT-ROTATE(T, x) modifies a binary search tree. Inorder tree walks of the input tree and the modified tree produce the same listing of key values.

```
LEFT-ROTATE(T, x)
 1  y ← right[x]            ▷ Set y.
 2  right[x] ← left[y]      ▷ Turn y's left subtree into x's right subtree
 3  p[left[y]] ← x
 4  p[y] ← p[x]             ▷ Link x's parent to y.
 5  if p[x] = nil[T]
 6     then root[T] ← y
 7     else if x = left[p[x]]
 8             then left[p[x]] ← y
 9             else right[p[x]] ← y
10  left[y] ← x                    ▷ Put x on y's left.
11  p[x] ← y
```

Inserting Nodes

- A node is inserted into the tree just as in BST.
- When a node is inserted, it is colored red initially.
- A procedure is called at the end of the insertion to correct the tree structure

Inserting Nodes

- Suppose we are inserting a node *z* in the tree.
- The following red-black properties can be violated:

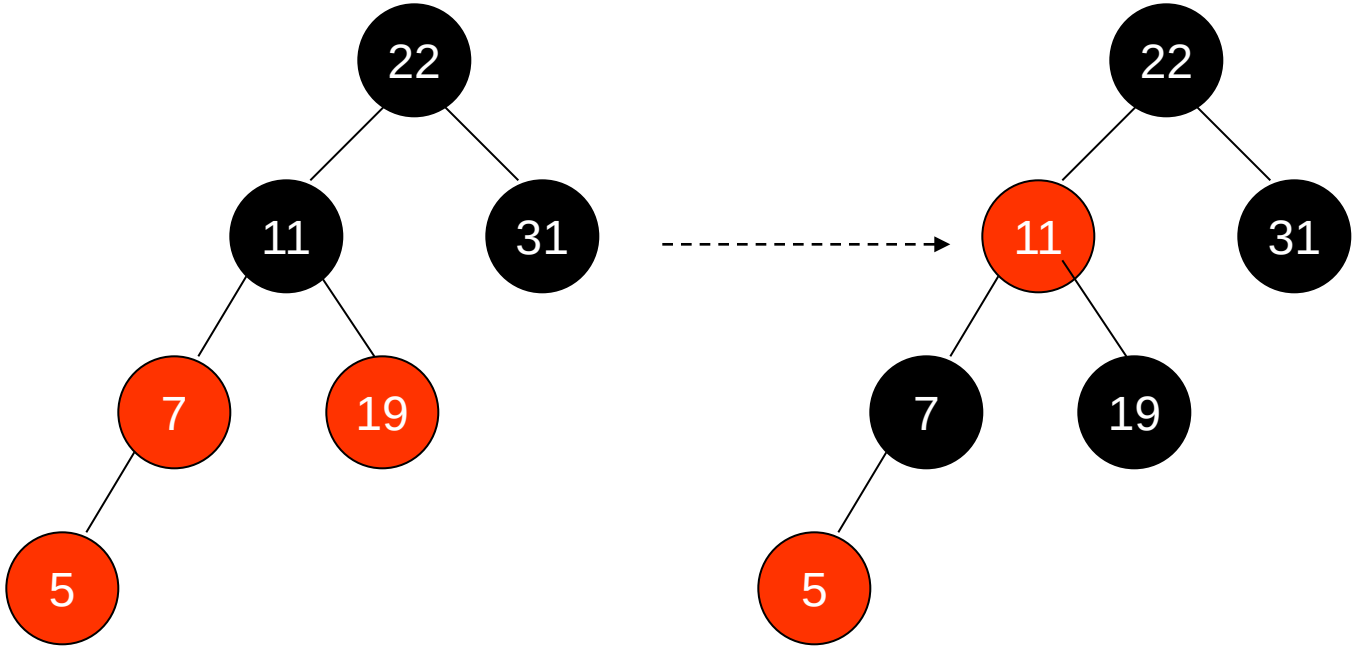
2. The root is black (when the first red node is inserted)

4. A red node has black children (when the parent of inserted node is red)

- A violation of property 4 leads to three cases:
  - Case 1: *z's* uncle *y* is red
  - Case 2: *z's* uncle *y* is black and *z* is a right child
  - Case 3: *z's* uncle *y* is black and *z* is a left child

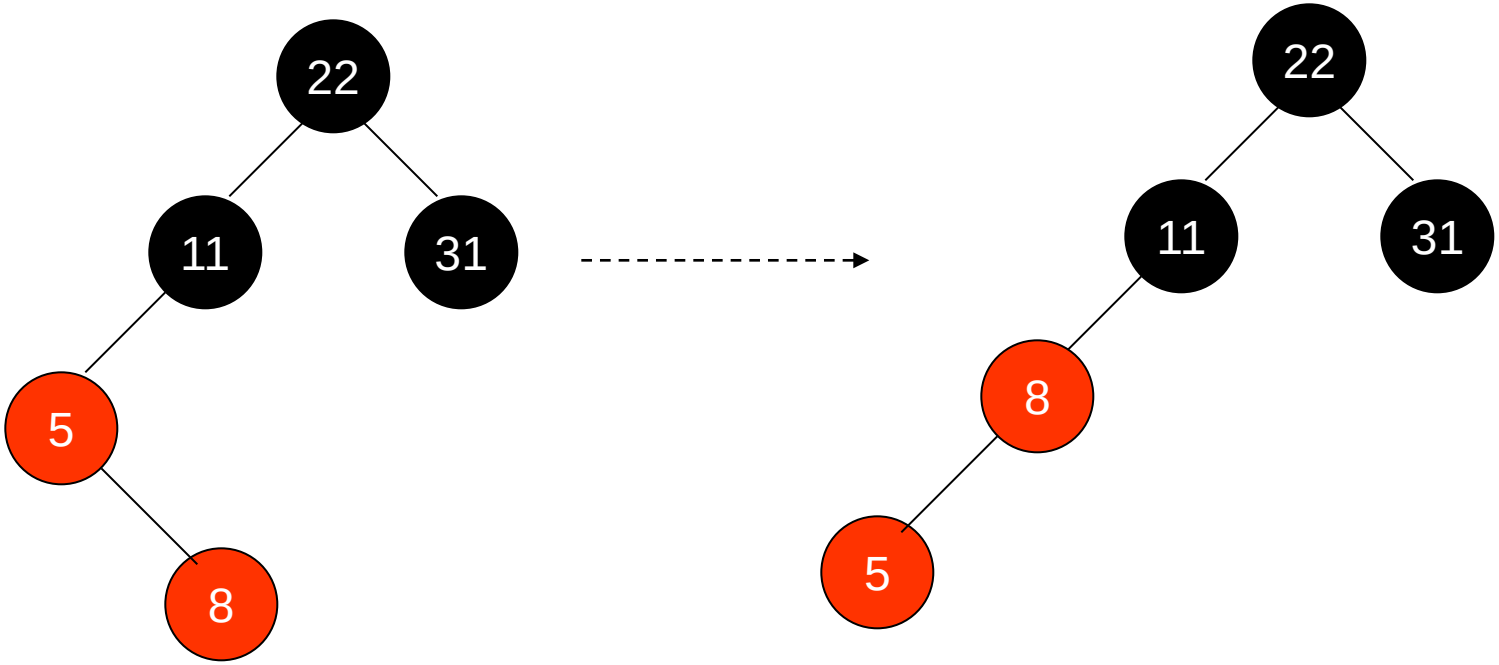
Inserting a node *z*

- Case 1: *z, p[z]* and *z's* uncle *y* all are red.
  - Since *p[p[z]]* is black, color it red, color p[z] and y black.
  - Check for the three cases from *p[p[z]]*, i.e., now *z* is the node *p[p[z]]*
    - i.e. recursively check violations till root node


22
11
31
7
19
5
22
11
31
7
19
5

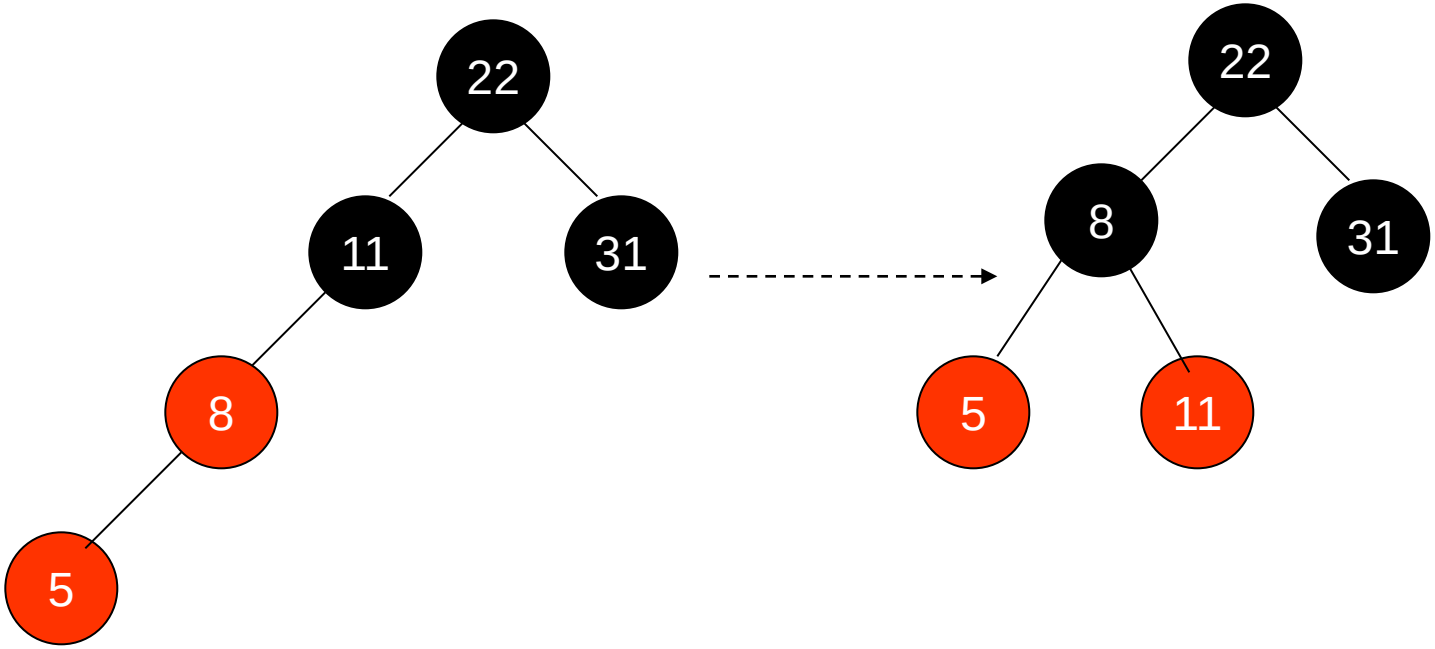
Inserting a node *z*

- Case 2: *z, p[z]* are red, *z's* uncle *y* is black, *z* is right child
  - Use a left rotation, we get to case 3.


22
11
31
5
8
22
11
31
8
5

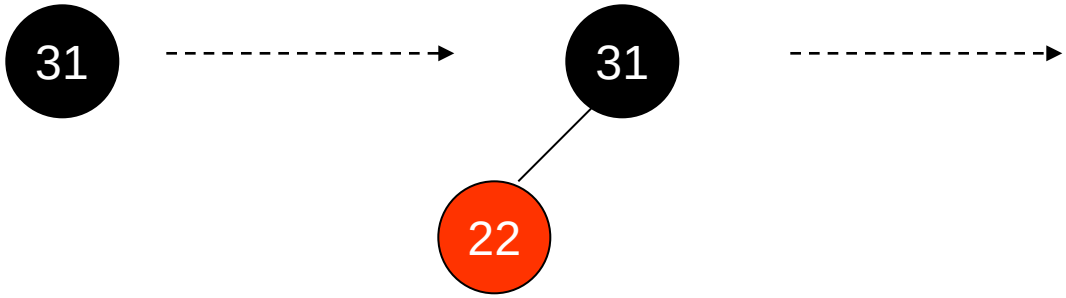
Inserting a node *z*

- Case 3: *z, p[z]* are red, *z's* uncle *y* is black, *z* is left child
  - Change color of *p[p[z]]* to red, *p[z]* to black and perform a right rotation on *p[p[z]]*


22
11
31
8
5
22
8
31
5
11

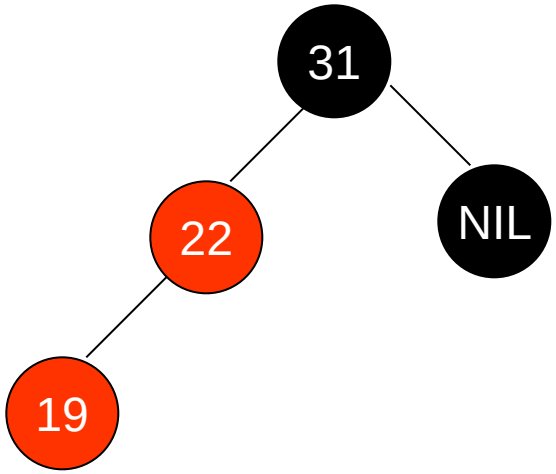
Example

- Input : { 31, 22, 19, 11, 7, 5, 18, 34, 44, 59, 80, 85, 60, 32}



Example

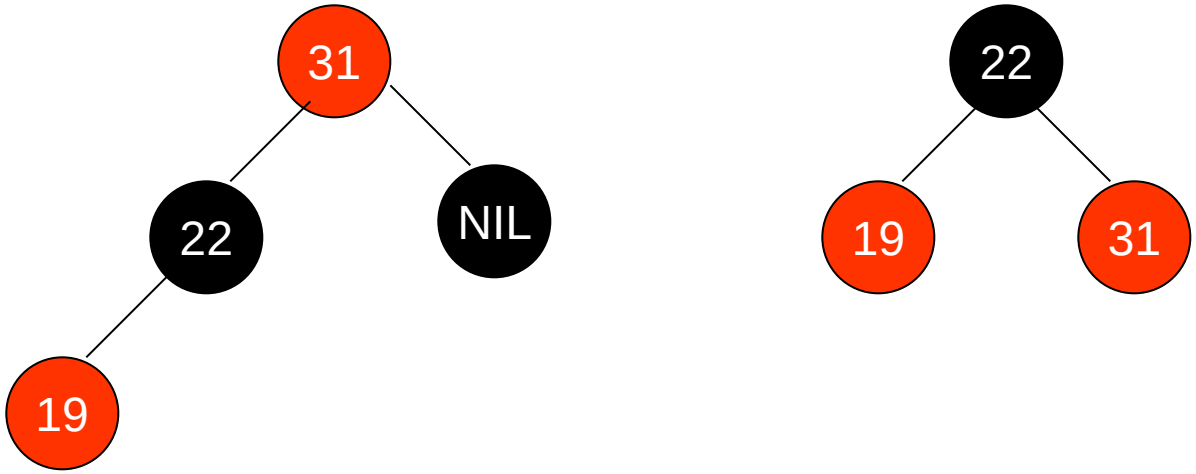
- Input : { 31, 22, 19, 11, 7, 5, 18, 34, 44, 59, 80, 85, 60, 32}



- z, p[z] – red , y is black, z is left child (case-3)
- Change color of *p[p[z]]* to red, *p[z]* to black and perform a right rotation on *p[p[z]]*

Example

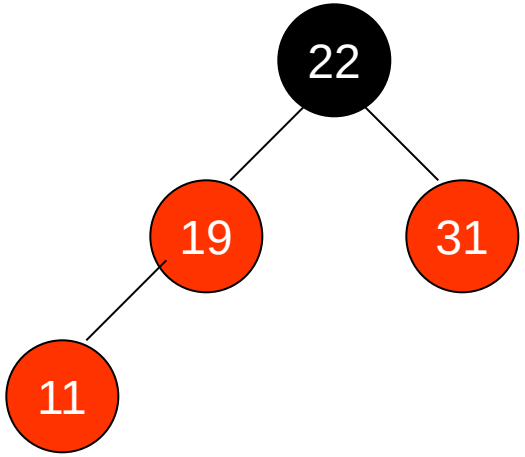
- Input : { 31, 22, 19, 11, 7, 5, 18, 34, 44, 59, 80, 85, 60, 32}



- z, p[z] – red , y is black, z is left child (case-3)
- Change color of *p[p[z]]* to red, *p[z]* to black and perform a right rotation on *p[p[z]]*

Example

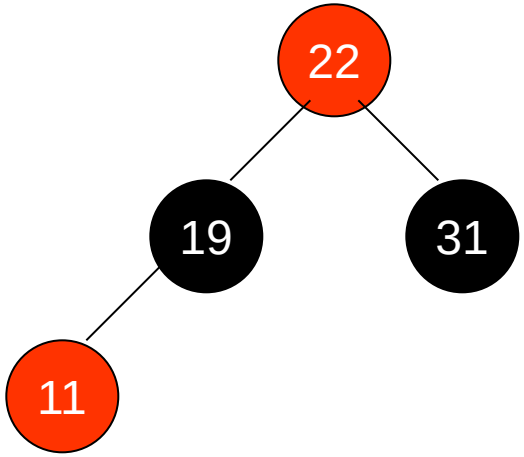
- Input : { 31, 22, 19, 11, 7, 5, 18, 34, 44, 59, 80, 85, 60, 32}



- z, p[z], y – red (case-1)
  - Since *p[p[z]]* is black, color it red, color p[z] and y black.
  - Check for the three cases from *p[p[z]]*, i.e., now *z* is the node *p[p[z]]*

Example

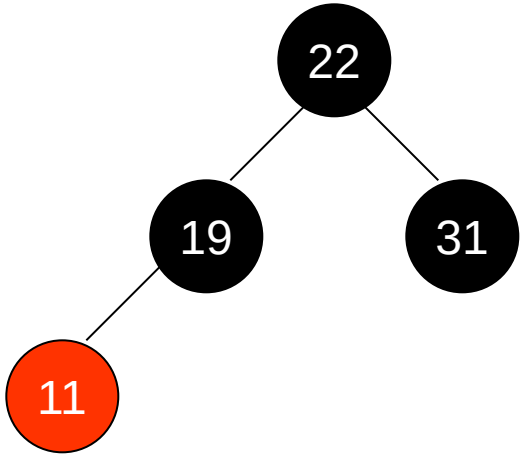
- Input : { 31, 22, 19, 11, 7, 5, 18, 34, 44, 59, 80, 85, 60, 32}



- z, p[z], y – red (case-1)
  - Since *p[p[z]]* is black, color it red, color p[z] and y black.
  - Check for the three cases from *p[p[z]]*, i.e., now *z* is the node *p[p[z]]*

Example

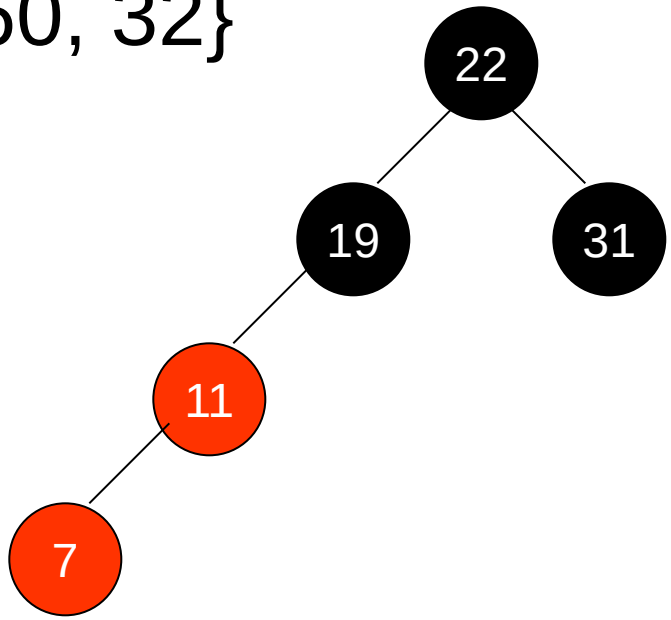
- Input : { 31, 22, 19, 11, 7, 5, 18, 34, 44, 59, 80, 85, 60, 32}



- Root cannot be Red, color it black

Example

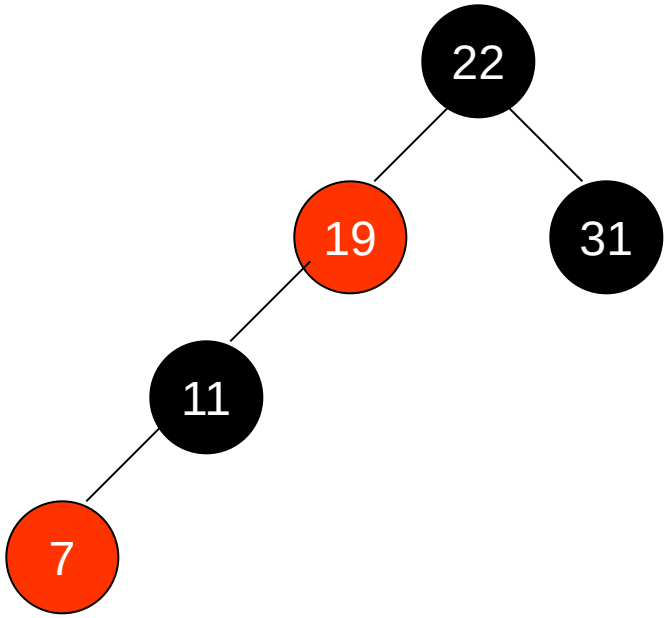
- Input : { 31, 22, 19, 11, 7, 5, 18, 34, 44, 59, 80, 85, 60, 32}

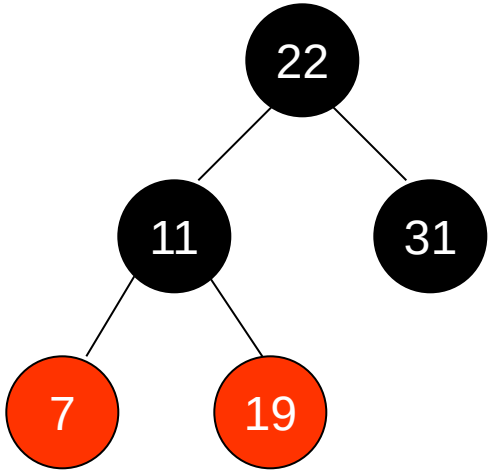


- z, p[z] – red , y is black, z is left child (case-3)
- Change color of *p[p[z]]* to red, *p[z]* to black and perform a right rotation on *p[p[z]]*

Example

- Input : { 31, 22, 19, 11, 7, 5, 18, 34, 44, 59, 80, 85, 60, 32}

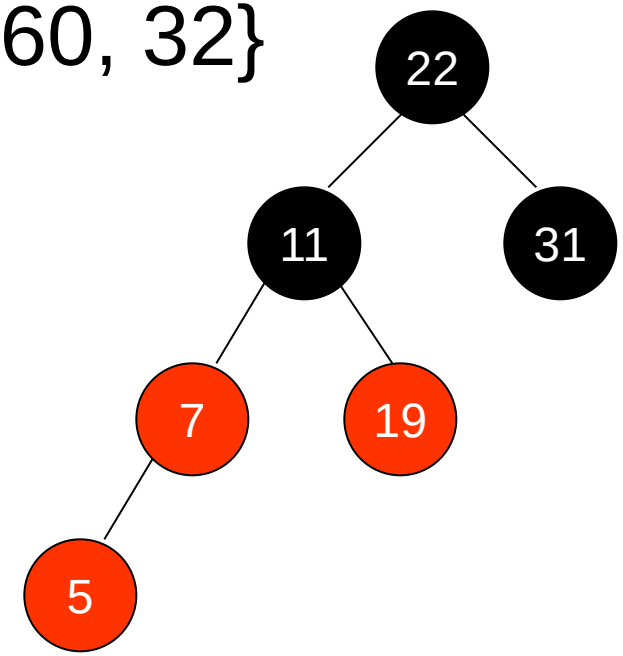




- z, p[z] – red , y is black, z is left child (case-3)
- Change color of *p[p[z]]* to red, *p[z]* to black and perform a right rotation on *p[p[z]]*

Example

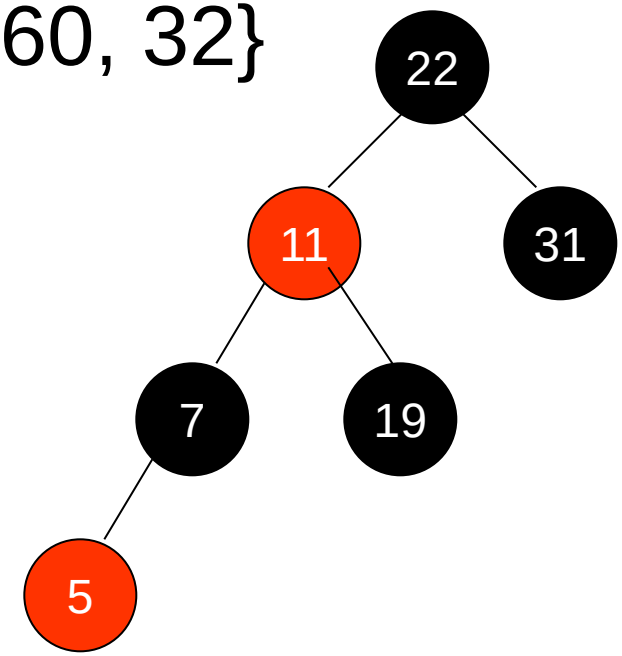
- Input : { 31, 22, 19, 11, 7, 5, 18, 34, 44, 59, 80, 85, 60, 32}



- z, p[z], y – red (case-1)
  - Since *p[p[z]]* is black, color it red, color p[z] and y black.
  - Check for the three cases from *p[p[z]]*, i.e., now *z* is the node *p[p[z]]*

Example

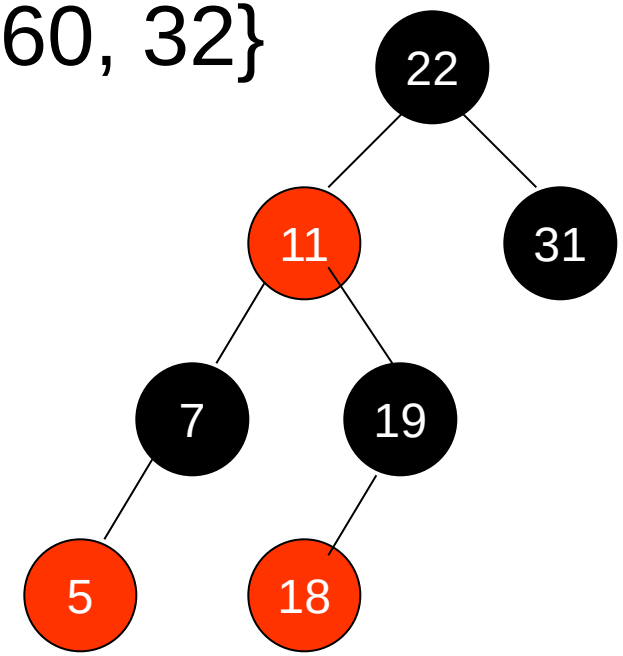
- Input : { 31, 22, 19, 11, 7, 5, 18, 34, 44, 59, 80, 85, 60, 32}



- z, p[z], y – red (case-1)
  - Since *p[p[z]]* is black, color it red, color p[z] and y black.
  - Check for the three cases from *p[p[z]]*, i.e., now *z* is the node *p[p[z]]*

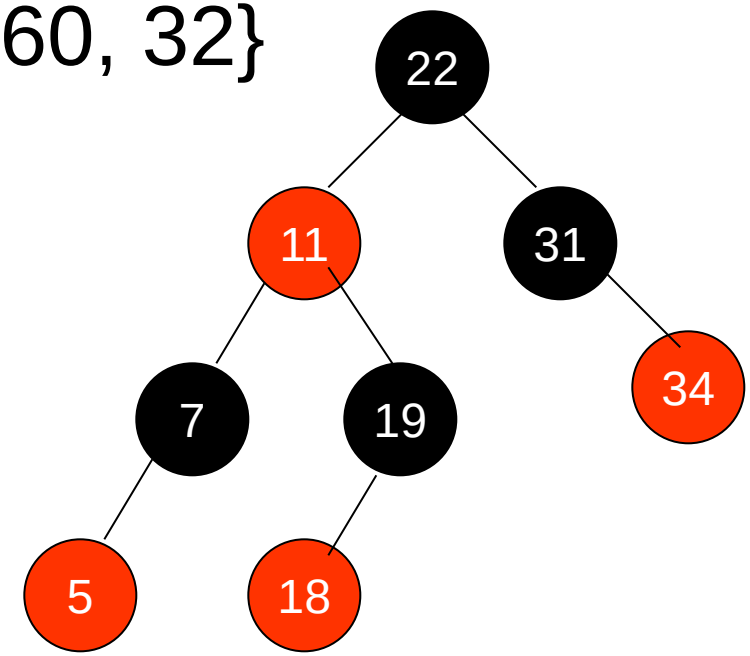
Example

- Input : { 31, 22, 19, 11, 7, 5, 18, 34, 44, 59, 80, 85, 60, 32}



Example

- Input : { 31, 22, 19, 11, 7, 5, 18, 34, 44, 59, 80, 85, 60, 32}



Inserting Nodes

- Suppose we are inserting a node *z* in the tree.
- The following red-black properties can be violated:

2. The root is black (when the first red node is inserted)

4. A red node has black children (when the parent of inserted node is red)

- A violation of property 4 leads to three cases:
  - Case 1: *z's* uncle *y* is red
  - Case 2: *z's* uncle *y* is black and *z* is a right child
  - Case 3: *z's* uncle *y* is black and *z* is a left child

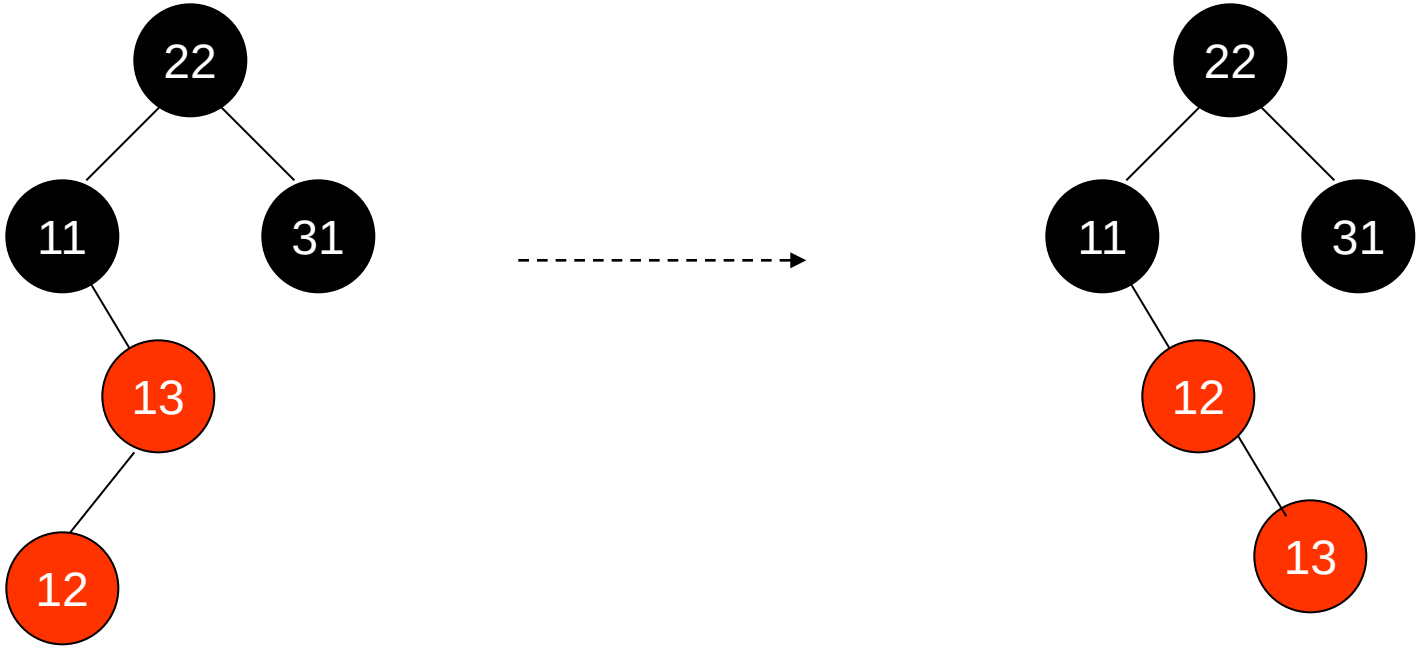
Some variations for case 2 and 3

- For both cases we have assumed that the p[z] is left child of p[p[z]]
- If p[z] is right child of p[p[z]] we have different rules …

Inserting a node *z*

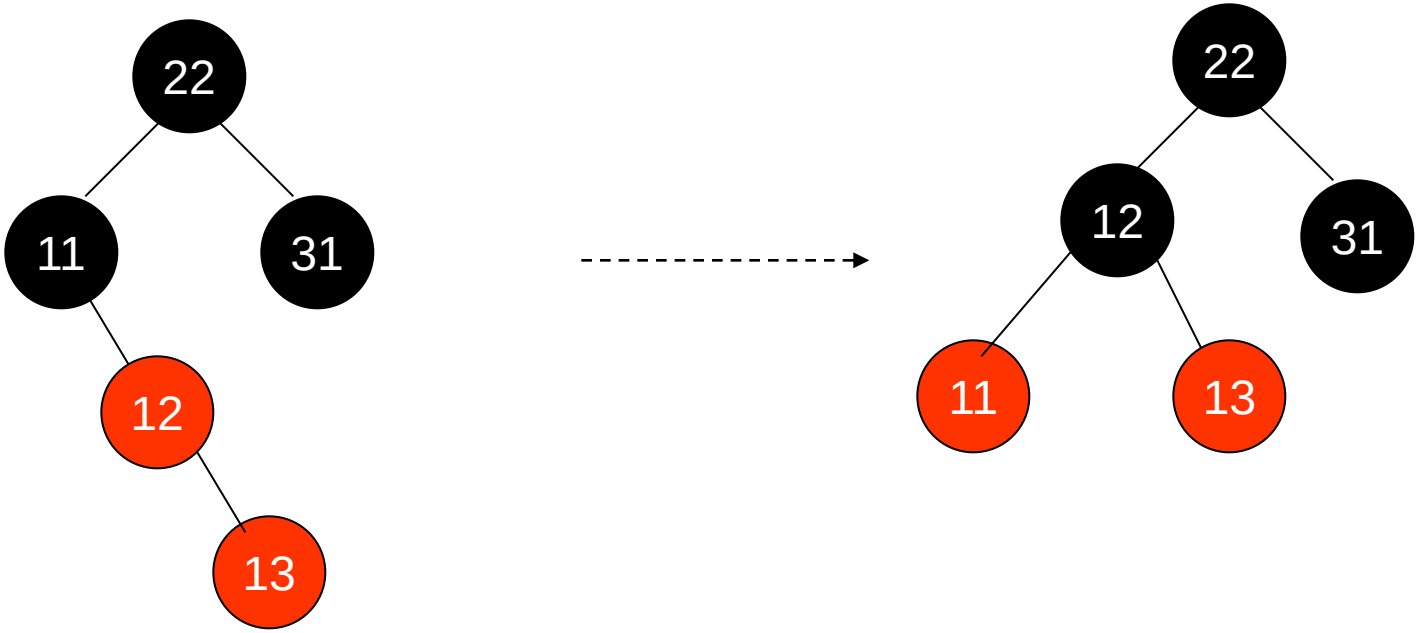
- Case 2': *z, p[z]* are red, *z's* uncle *y* is black, *z* is left child
- And p[z] is **right** child of p[p[z]]

  – Use a **right** rotation, we get to case 3'.


22
11
31
13
12
22
11
31
12
13

Inserting a node *z*

- Case 3': *z, p[z]* are red, *z's* uncle *y* is black, *z* is right child
- And p[z] is **right** child of p[p[z]]
  - Change color of *p[p[z]]* to red, *p[z]* to black and perform a **left** rotation on *p[p[z]]*

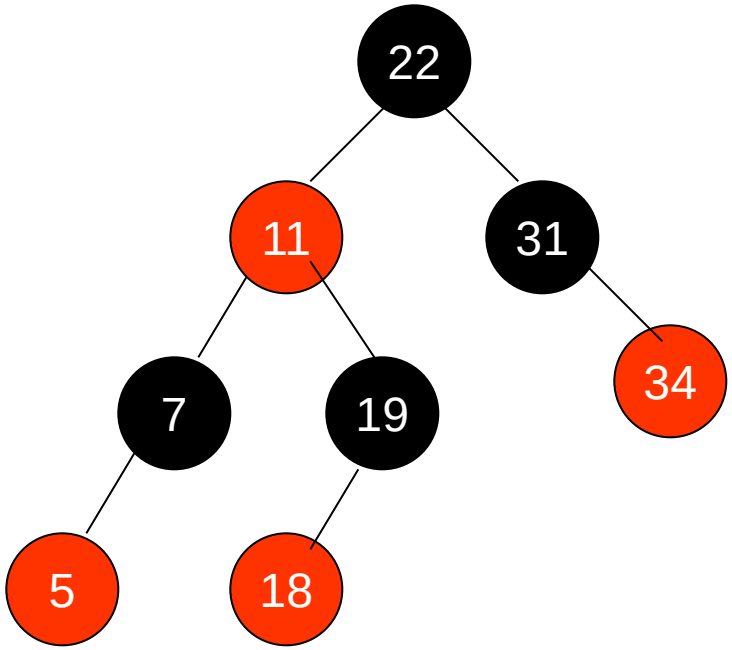

22
11
31
12
13
22
12
31
11
13

A general algorithm

- *If z's uncle y is Red*
  - *Case-1*
- *else if p(z) = left child of p(p(z))*
  - *case 2 or case 3*
- *Else*
  - *case 2' or case 3'*

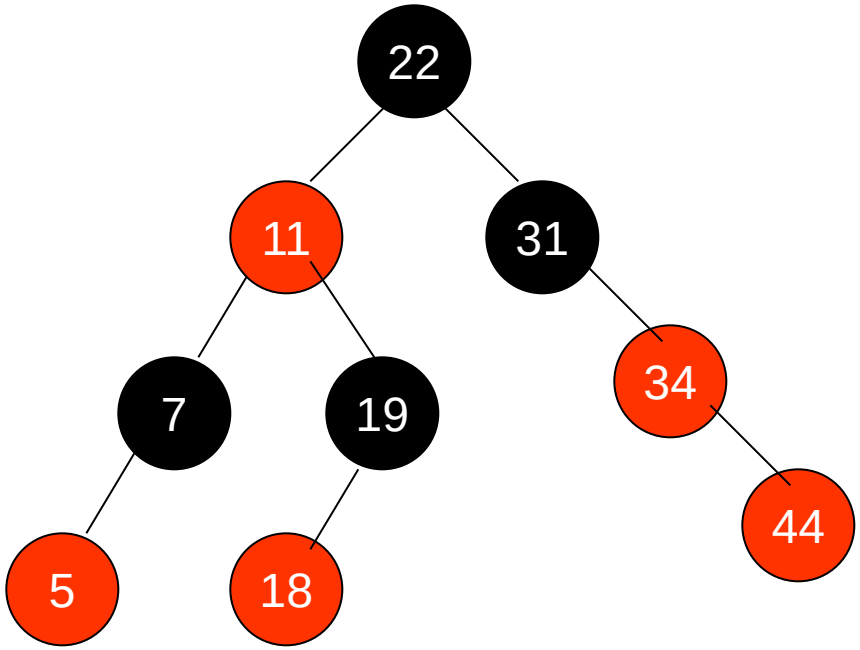
Example (continued from last lecture)

- Input : { 31, 22, 19, 11, 7, 5, 18, 34, 44, 59, 80, 85, 60, 32}



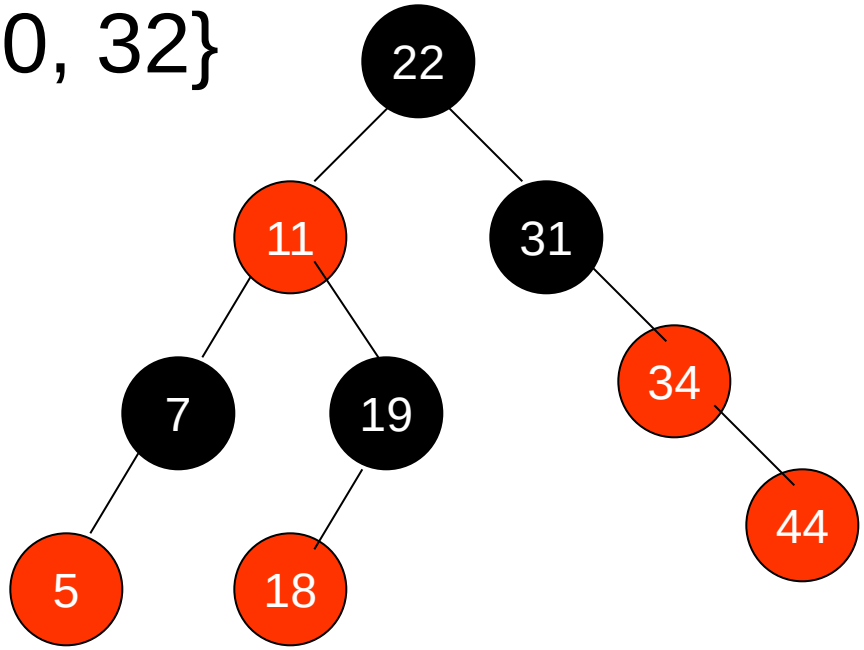
Example (continued from last lecture)

- Input : { 31, 22, 19, 11, 7, 5, 18, 34, 44, 59, 80, 85, 60, 32}



Example

- Input : { 31, 22, 19, 11, 7, 5, 18, 34, 44, 59, 80, 85, 60, 32}



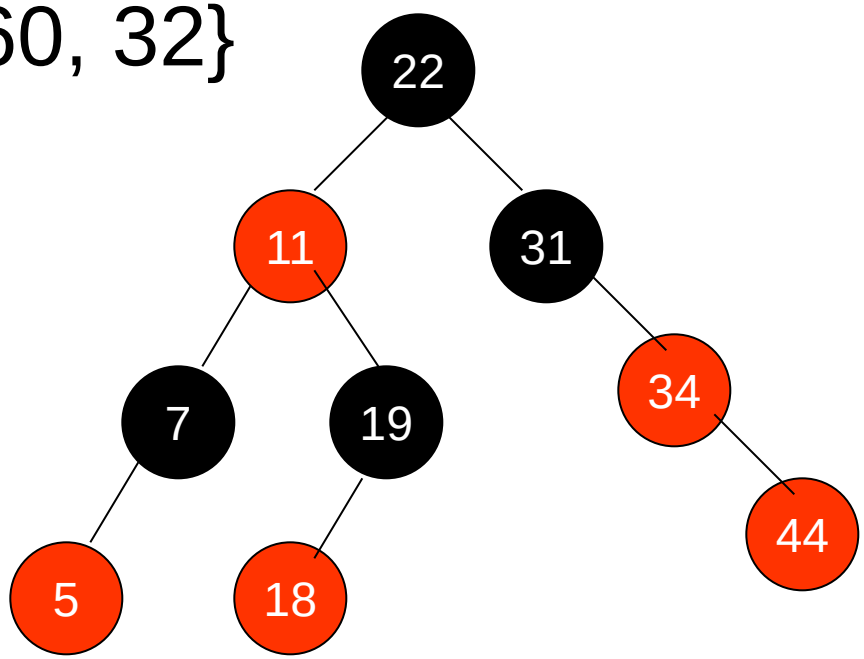
- Which case to apply, here ?
- Case 2 seems to be obvious choice,
  - *z's* uncle *y* is black and *z* is a right child
- but …

Example

- If p[z] is right child of p[p[z]] then we use case 2’ or case 3’

Example

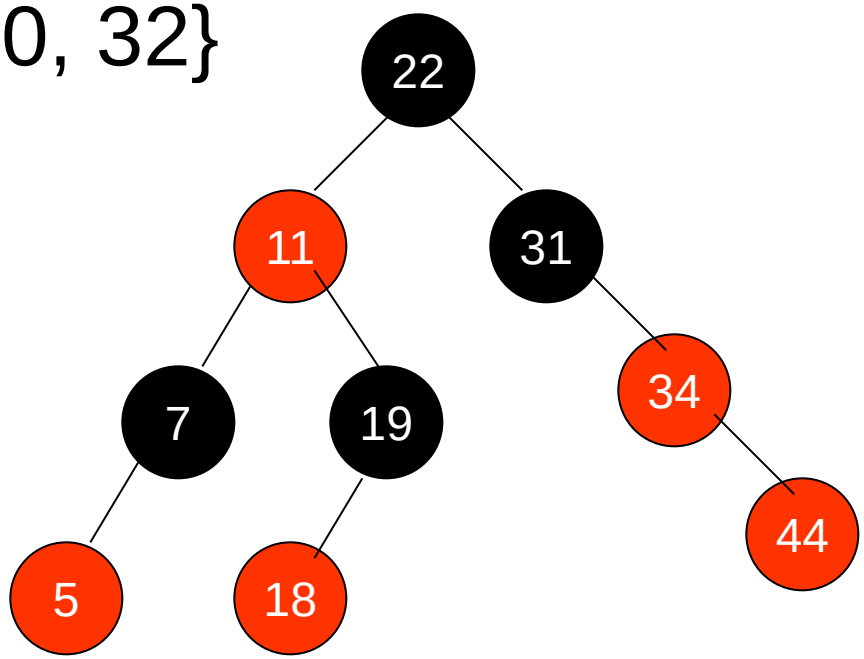
- Input : { 31, 22, 19, 11, 7, 5, 18, 34, 44, 59, 80, 85, 60, 32}



- Which case to apply, here ?
- p[z] is right child of p[p[z]]

Example

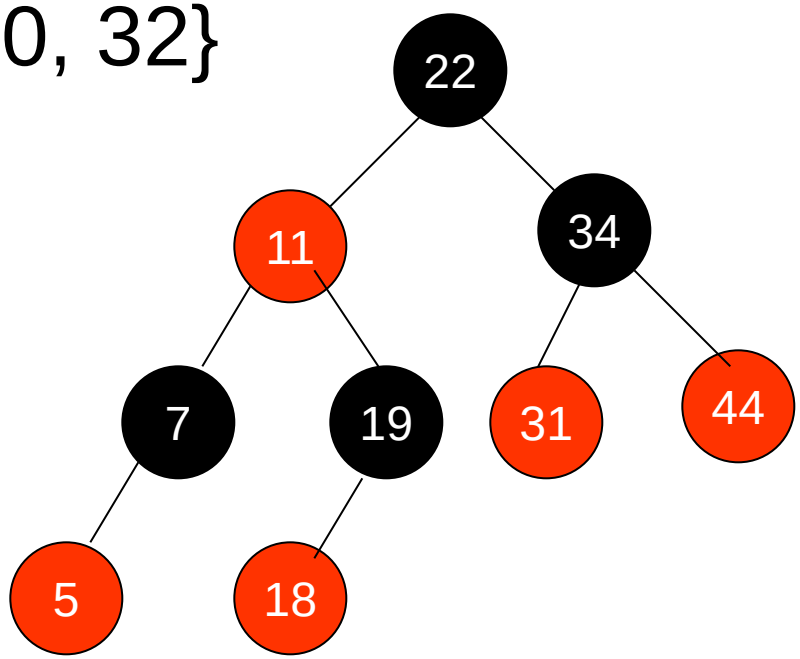
- Input : { 31, 22, 19, 11, 7, 5, 18, 34, 44, 59, 80, 85, 60, 32}



- z, p[z] – red , y is black, p[z] is right child of p[p[z]], z is **right** child (case-3')
- Change color of *p[p[z]]* to red, *p[z]* to black and perform a **left** rotation on *p[p[z]]*

Example

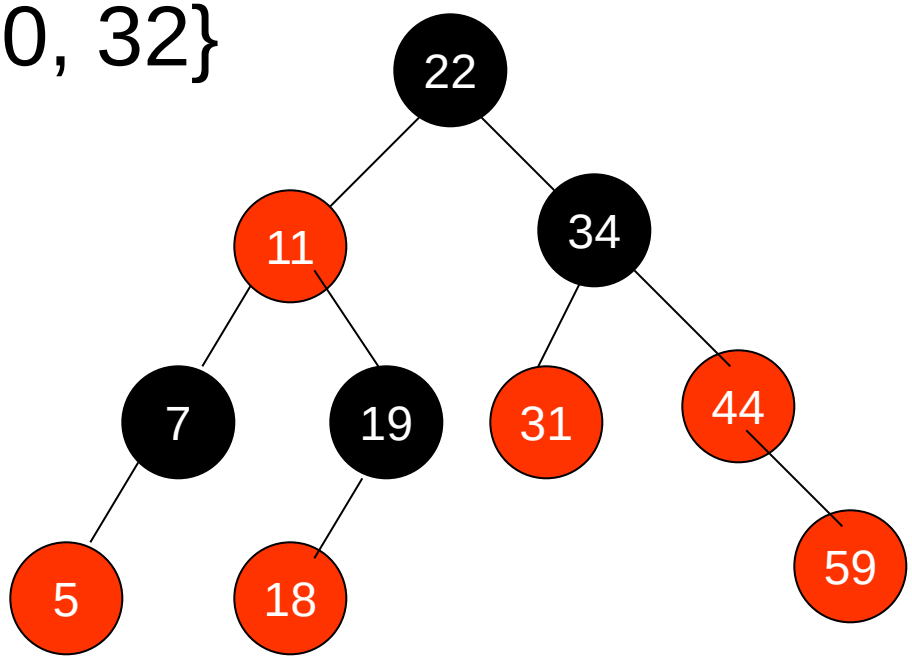
- Input : { 31, 22, 19, 11, 7, 5, 18, 34, 44, 59, 80, 85, 60, 32}



- z, p[z] – red , y is black, p[z] is right child of p[p[z]], z is **right** child (case-3')
- Change color of *p[p[z]]* to red, *p[z]* to black and perform a **left** rotation on *p[p[z]]*

Example

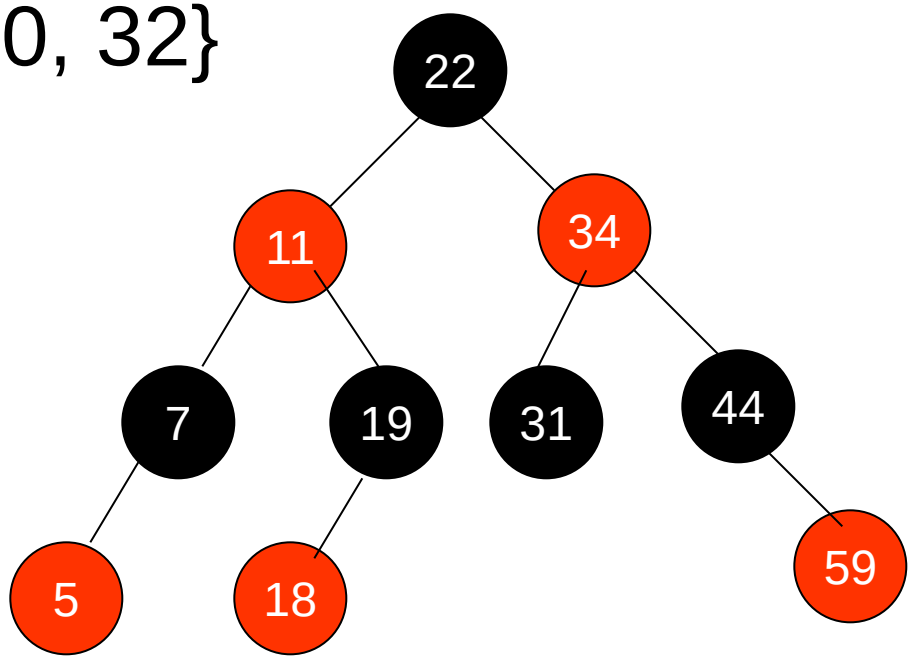
- Input : { 31, 22, 19, 11, 7, 5, 18, 34, 44, 59, 80, 85, 60, 32}



- z, p[z], y – red (case-1)
  - Since *p[p[z]]* is black, color it red, color p[z] and y black.
  - Check for the three cases from *p[p[z]]*, i.e., now *z* is the node *p[p[z]]*

Example

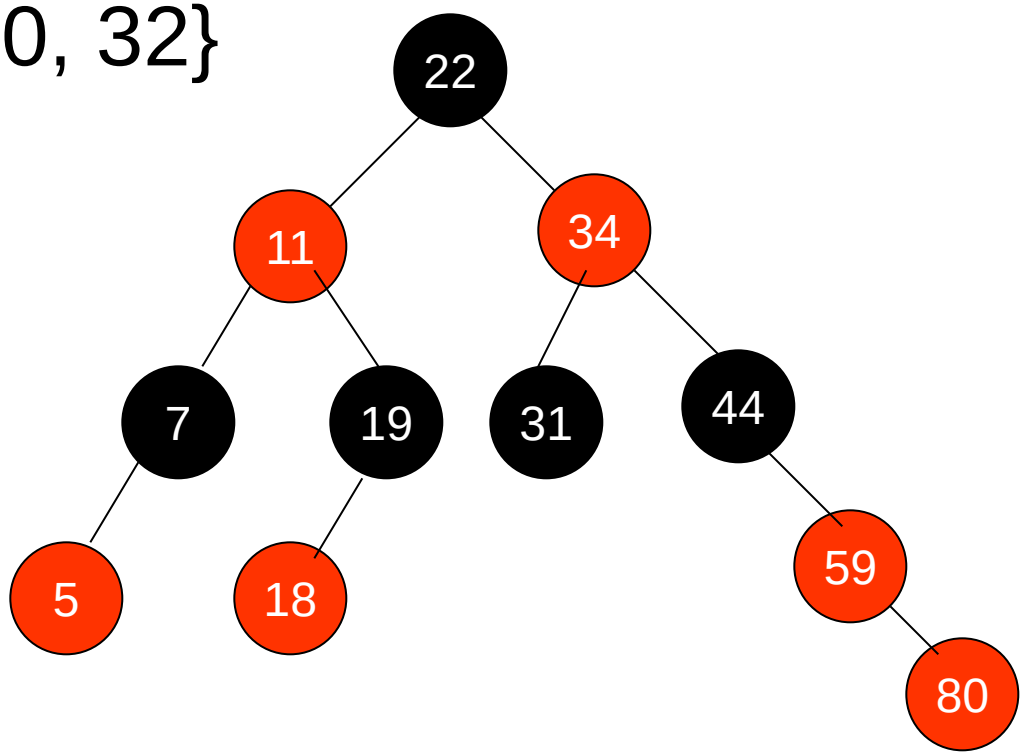
- Input : { 31, 22, 19, 11, 7, 5, 18, 34, 44, 59, 80, 85, 60, 32}



- z, p[z], y – red (case-1)
  - Since *p[p[z]]* is black, color it red, color p[z] and y black.
  - Check for the three cases from *p[p[z]]*, i.e., now *z* is the node *p[p[z]]*

Example

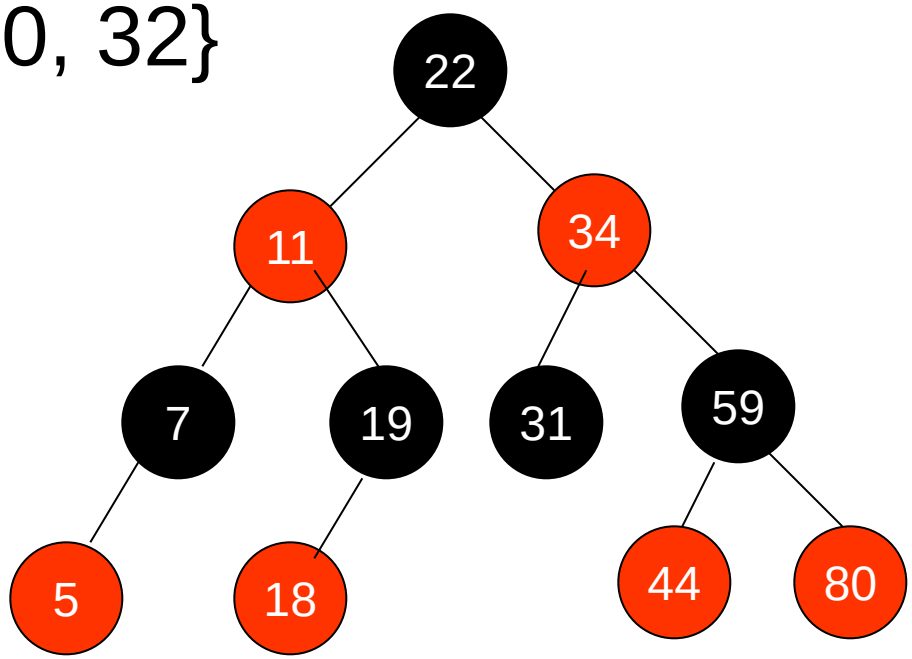
- Input : { 31, 22, 19, 11, 7, 5, 18, 34, 44, 59, 80, 85, 60, 32}



- z, p[z] – red , y is black, p[z] is right child of p[p[z]], z is **right** child (case-3')
- Change color of *p[p[z]]* to red, *p[z]* to black and perform a **left** rotation on *p[p[z]]*

Example

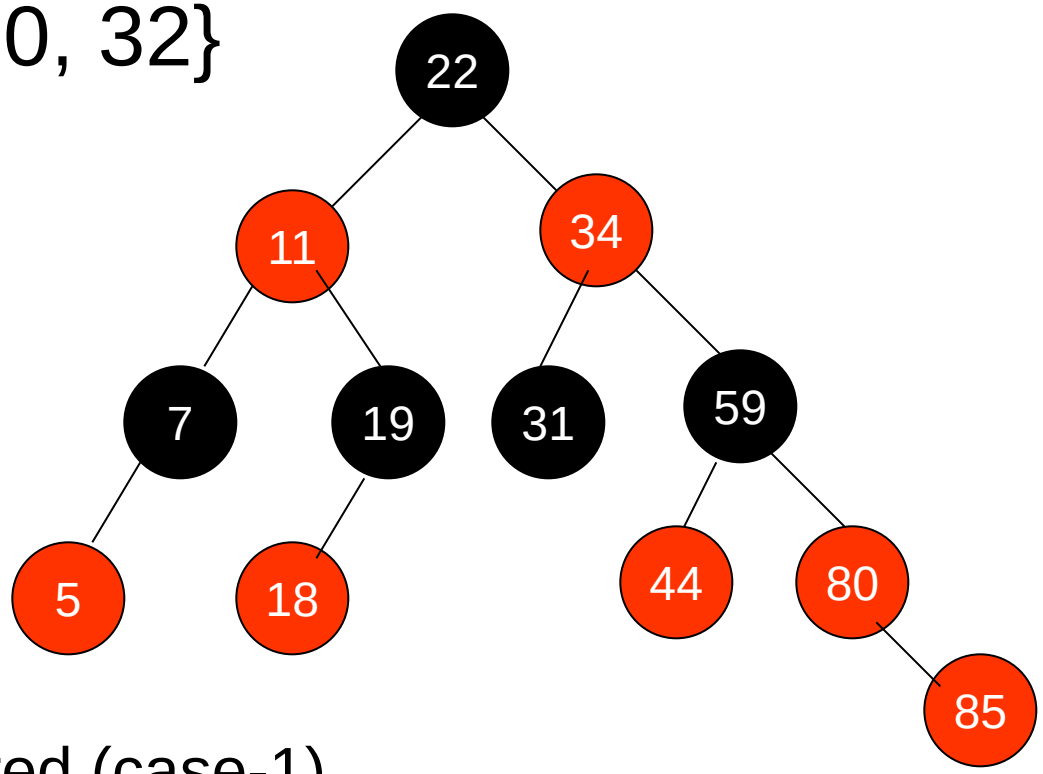
- Input : { 31, 22, 19, 11, 7, 5, 18, 34, 44, 59, 80, 85, 60, 32}



- z, p[z] – red , y is black, p[z] is right child of p[p[z]], z is **right** child (case-3)
- Change color of *p[p[z]]* to red, *p[z]* to black and perform a **left** rotation on *p[p[z]]*

Example

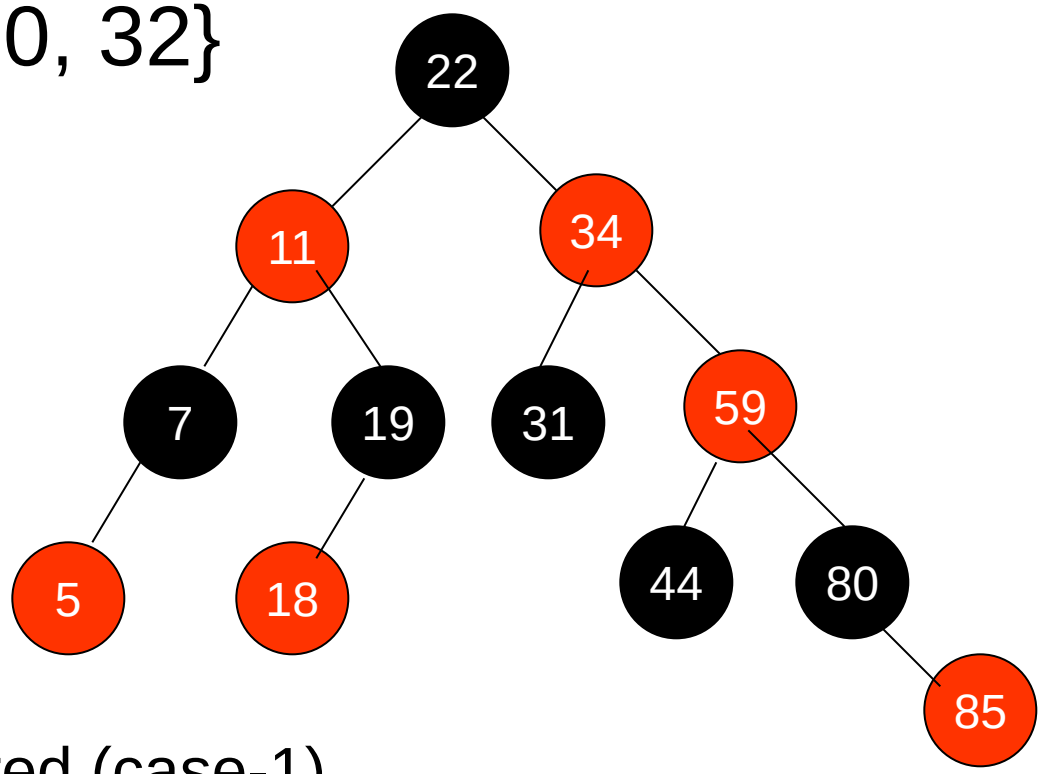
- Input : { 31, 22, 19, 11, 7, 5, 18, 34, 44, 59, 80, 85, 60, 32}



- z, p[z], y – red (case-1)
  - Since *p[p[z]]* is black, color it red, color p[z] and y black.
  - Check for the three cases from *p[p[z]]*, i.e., now *z* is the node *p[p[z]]*

Example

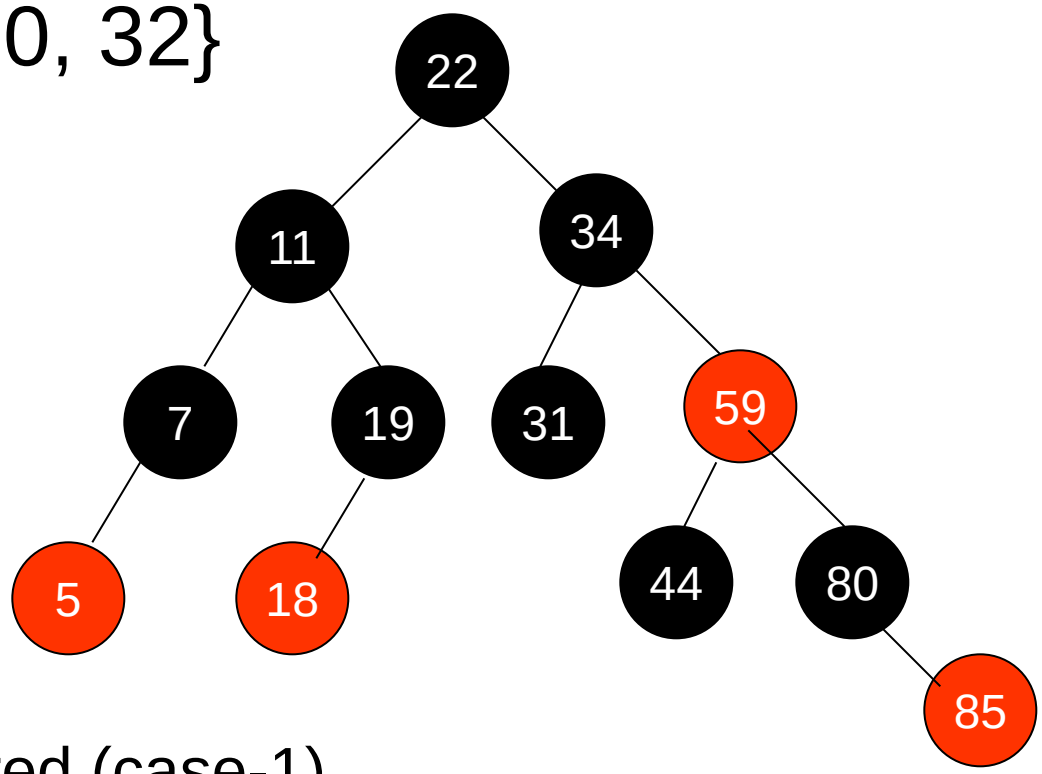
- Input : { 31, 22, 19, 11, 7, 5, 18, 34, 44, 59, 80, 85, 60, 32}



- z, p[z], y – red (case-1)
  - Since *p[p[z]]* is black, color it red, color p[z] and y black.
  - Check for the three cases from *p[p[z]]*, i.e., now *z* is the node *p[p[z]]*

Example

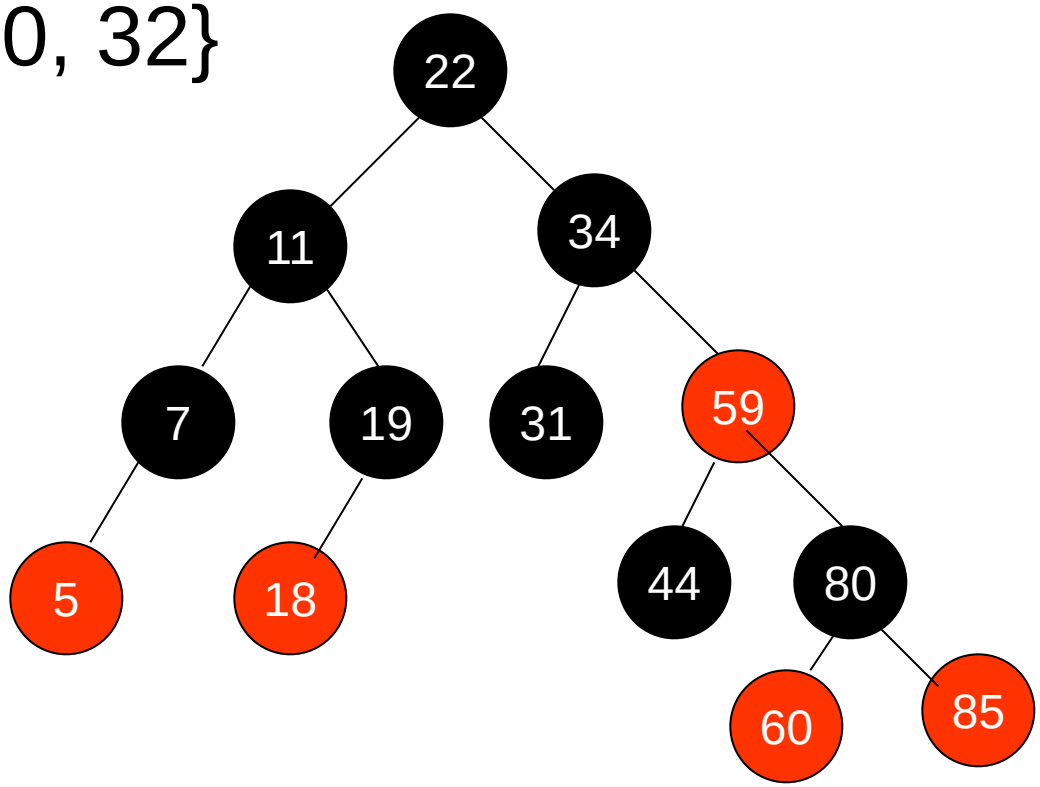
- Input : { 31, 22, 19, 11, 7, 5, 18, 34, 44, 59, 80, 85, 60, 32}



- z, p[z], y – red (case-1)
  - Since *p[p[z]]* is black, color it red, color p[z] and y black.
  - Check for the three cases from *p[p[z]]*, i.e., now *z* is the node *p[p[z]]*

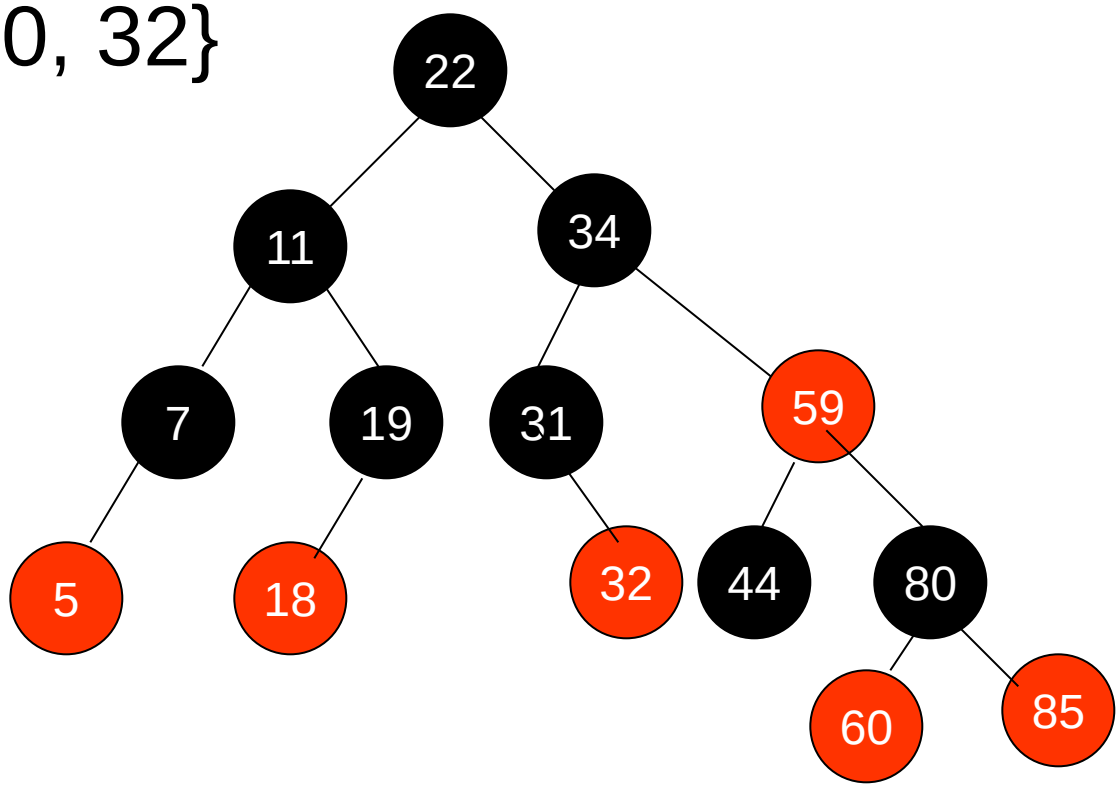
Example

- Input : { 31, 22, 19, 11, 7, 5, 18, 34, 44, 59, 80, 85, 60, 32}



Example

- Input : { 31, 22, 19, 11, 7, 5, 18, 34, 44, 59, 80, 85, 60, 32}



```
RB-INSERT(T, z)
 1  y ← nil[T]
 2  x ← root[T]
 3  while x ≠ nil[T]
 4        do y ← x
 5           if key[z] < key[x]
 6              then x ← left[x]
 7              else  x ← right[x]
 8  p[z] ← y
 9  if y = nil[T]
10     then root[T] ← z
11     else  if key[z] < key[y]
12              then left[y] ← z
13              else  right[y] ← z
14  left[z] ← nil[T]
15  right[z] ← nil[T]
16  color[z] ← RED
17  RB-INSERT-FIXUP(T, z)
```

```
RB-INSERT-FIXUP(T, z)
 1  while color[p[z]] = RED
 2        do if p[z] = left[p[p[z]]]
 3              then y ← right[p[p[z]]]
 4                   if color[y] = RED
 5                     then color[p[z]] ← BLACK                    ▷ Case 1
 6                          color[y] ← BLACK                      ▷ Case 1
 7                          color[p[p[z]]] ← RED                  ▷ Case 1
 8                          z ← p[p[z]]                           ▷ Case 1
 9                     else if z = right[p[z]]
10                             then z ← p[z]                      ▷ Case 2
11                                  LEFT-ROTATE(T, z)             ▷ Case 2
12                          color[p[z]] ← BLACK                   ▷ Case 3
13                          color[p[p[z]]] ← RED                  ▷ Case 3
14                          RIGHT-ROTATE(T, p[p[z]])              ▷ Case 3
15              else (same as then clause
                      with "right" and "left" exchanged)
16  color[root[T]] ← BLACK
```

- Chapter 13, “Red Black Trees”, Introduction to Algorithms By *Cormen et al*

Questions?

"He who asks a question is a fool for five minutes; he who does not ask a question remains a fool forever"
Chinese Proverb

"The wise man doesn't give the right answers, he poses the right questions."
Claude Levi-Strauss

"A wise man can learn more from a foolish question than a fool can learn from a wise answer."
Bruce Lee