Red-Black Tree

CS313E

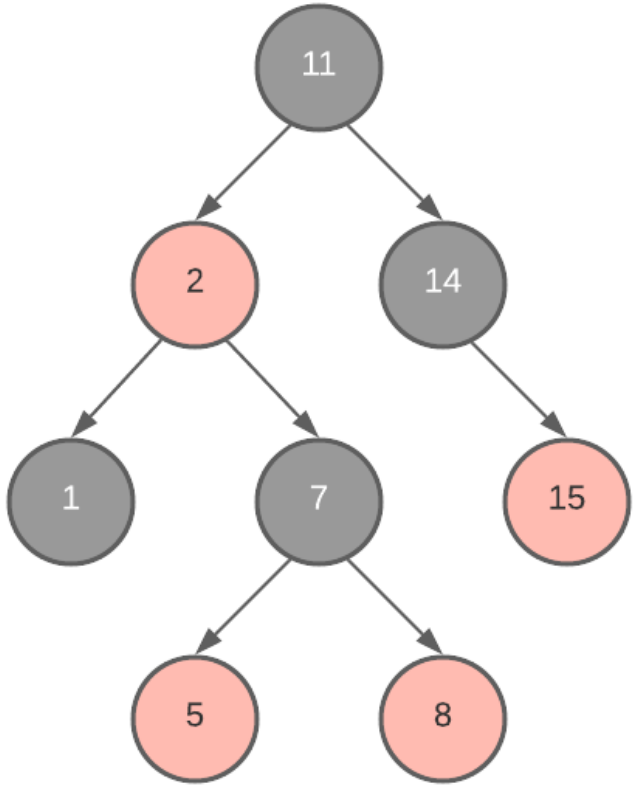
Table of Content

First Things First

- Binary Search Tree (BST)
- Balanced Search Tree
- Root
- Leaves
- Uncles/Aunts
- Grandparents
- Parents
- NIL

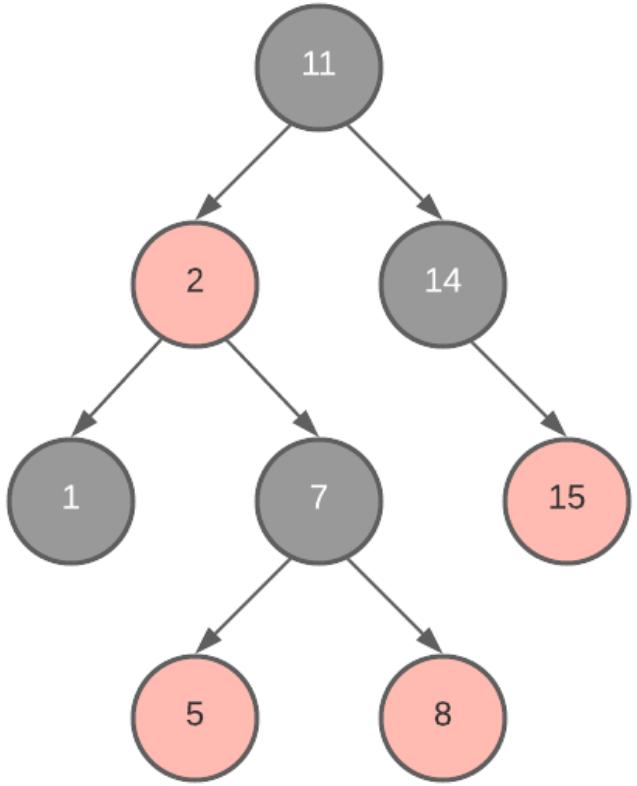
Overview

- Red-Black tree is a type of self balancing binary search tree.
- They have an extra bit that interpreted as the color (red or black).
- The colors are used to ensure the tree remains balance during insertion and deletion
  - Because of the self balancing insert, delete, and search are all O(log(n)) time where n is the number of nodes in a tree



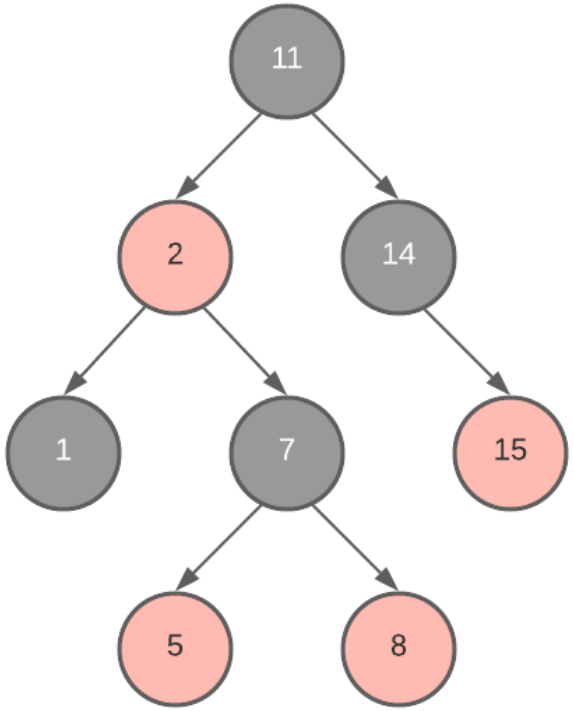
Rules of Red-Black Tree

1. Every node has a color of either red or black
2. Root and leaves (NIL) of the tree is ALWAYS black
3. If a node is a red, it's children must be black
4. Every node from a node (including root) to any of it's descendant NIL node has the same number of black nodes



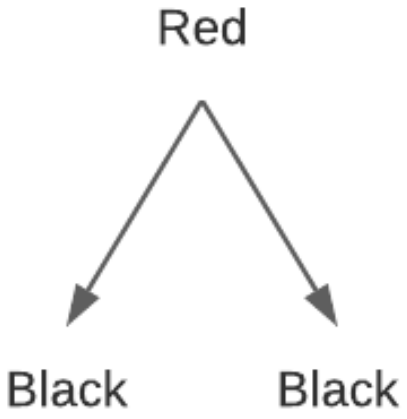
Example Red-Black Tree

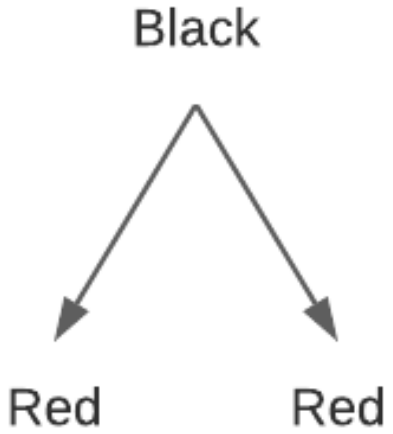
- Black-height – the number of black nodes on a path from the root to a leaf.
- Longest path (root to farthest NIL) is no more than twice the length of the shortest path (root to nearest NIL)
  - Shortest path: all black nodes
  - Longest Path: alternating red and black
- The height of the tree is log(n)



Rebalancing

- Rebalancing is used to fix broken properties of Red-Black Trees when inserting and deleting.



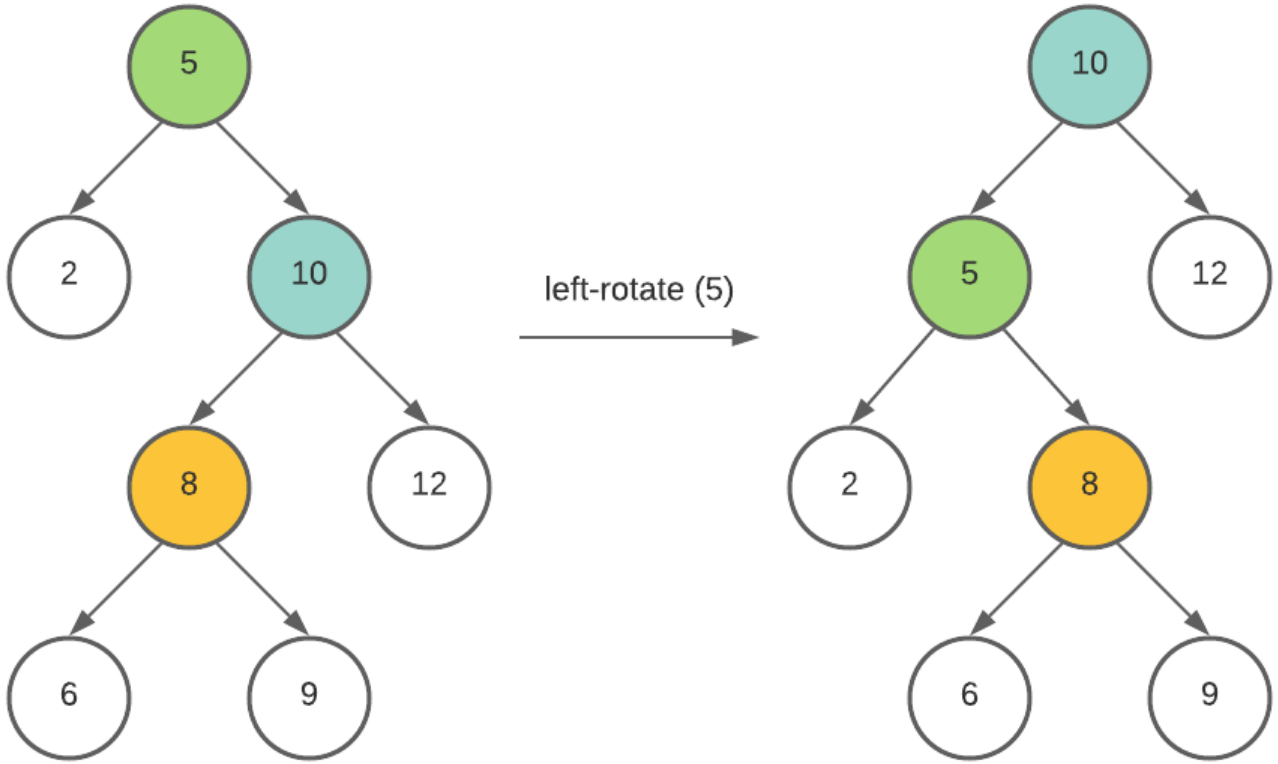


Rotation

- Alters the structure of a tree by rearranging subtrees
- Goal is to decrease the height of the tree
  - Larger subtrees are move up, smaller subtrees down
- Does not affect the order of elements
- Time complexity of O(1)

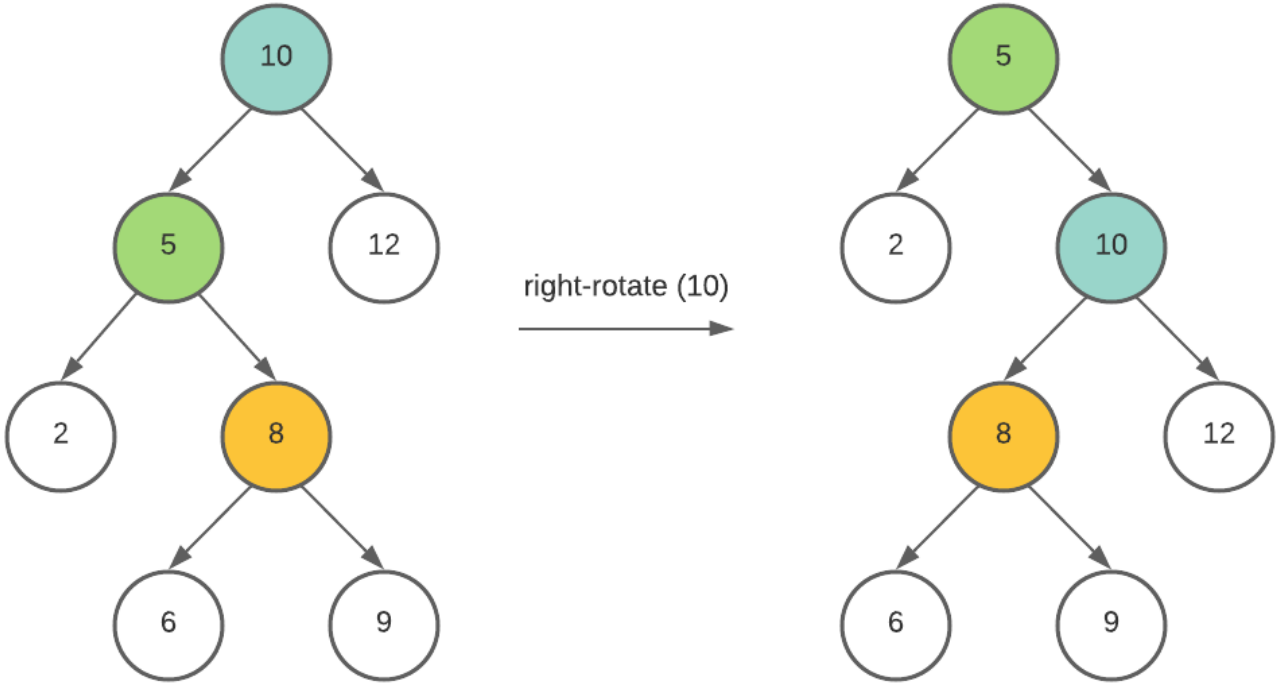
Left Rotation

- Example: left-rotate (5)
  - (5)’s right child (10) becomes it’s parent
  - (5) becomes the left child of (10)
  - (8) becomes the right child of (5)



Right Rotation

- Example: right-rotate (10)
  - (10)’s right child (5) becomes it’s parent
  - (10) becomes the right child of (5)
  - (8) becomes the left child of (10)

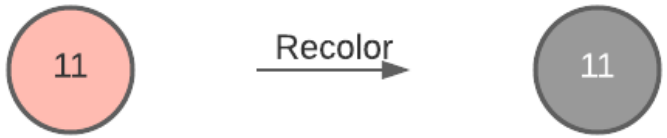


Insertion

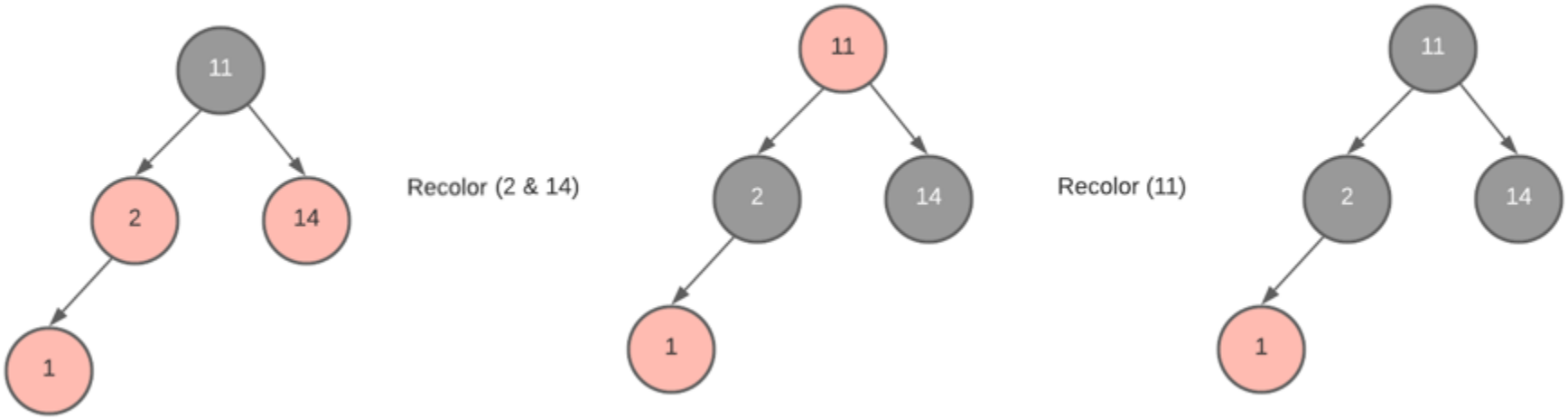
- Insert the node and color it red
- Recolor and rotate nodes to fix violation
- 4 Cases where Z is the node new node being inserted
  1. Z is root → Color the new node black
  2. Z.uncle is red → Recolor
  3. Z.uncle is black (triangle) → rotate on new node's parent
  4. Z.uncle is black (line) → rotate on new node's grandparent & recolor

Case 1: Z = root

1. Z (11) is inserted into the root of the tree as red
2. To fix the violated property of “the root must be Black” it’s recolored to black.

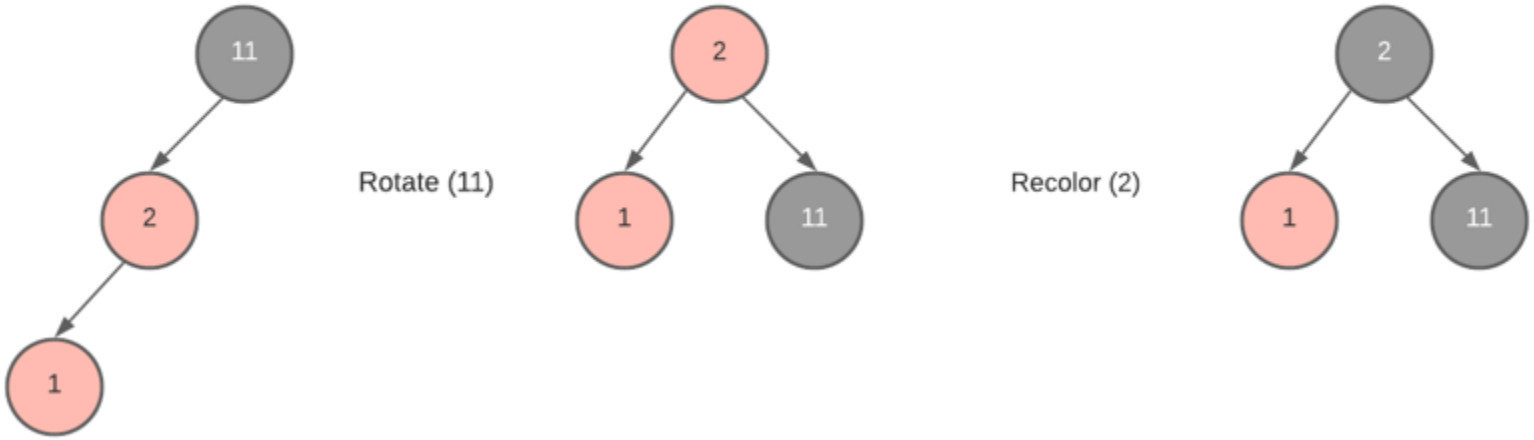


Case 2: Z.uncle = red



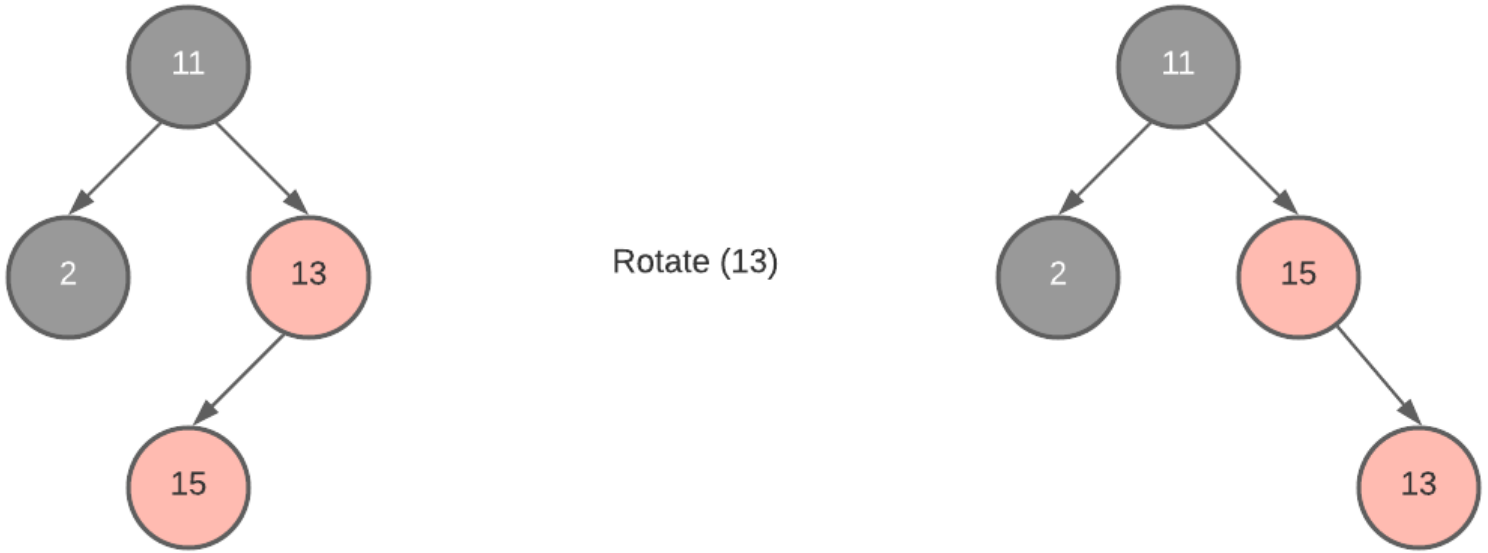
Z (1) is inserted

Case 3: Z.uncle = black (line)



Z (1) is inserted

Case 4: Z.uncle = black (triangle)



Z (15) is inserted

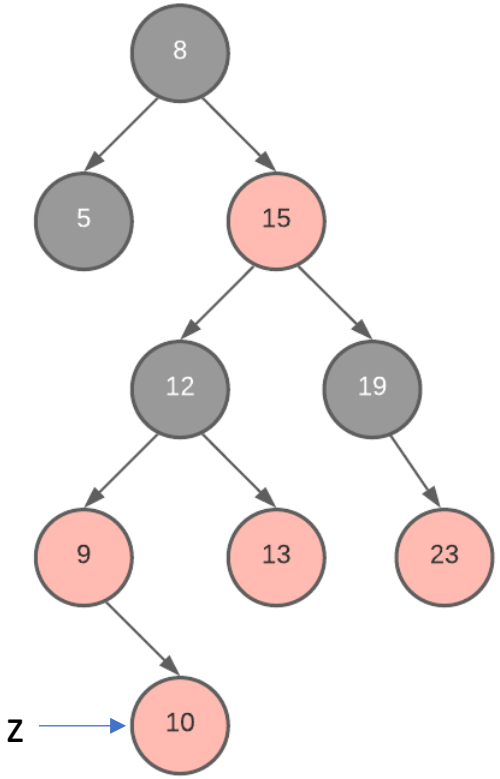
Insertion Algorithm

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

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

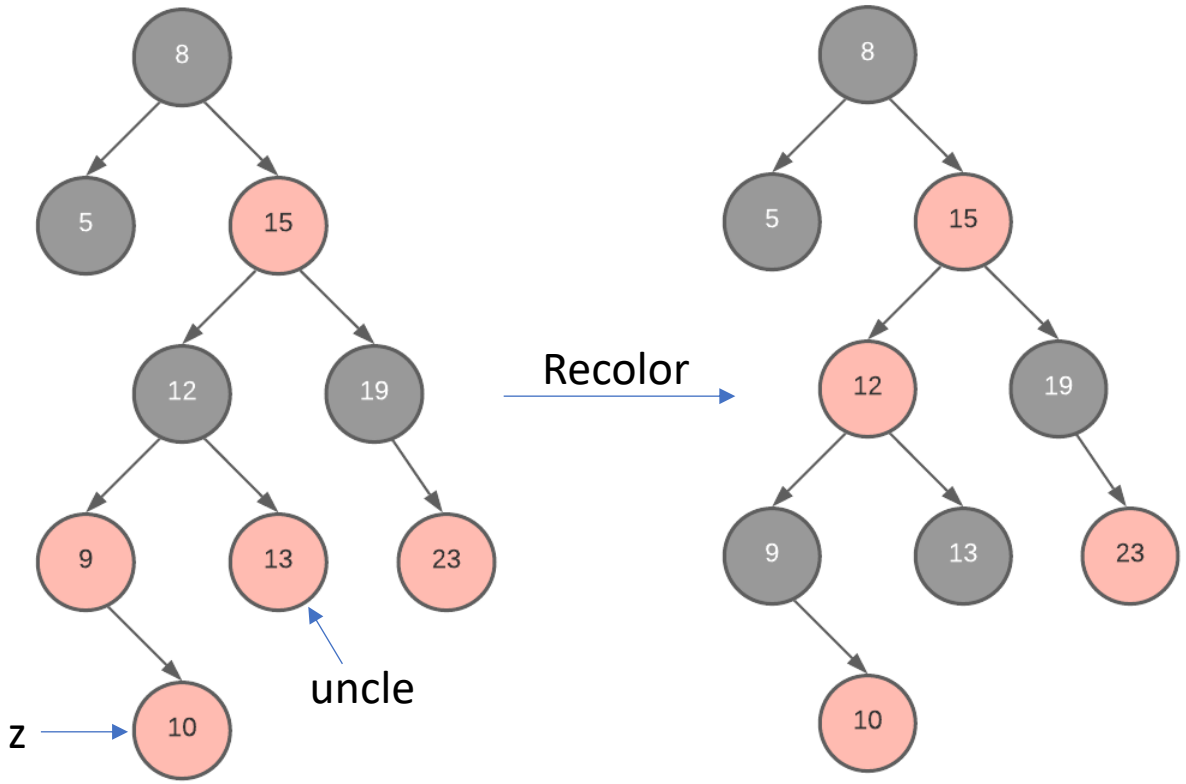
Example of Insertion

1. Insert 10 (z) and color it red



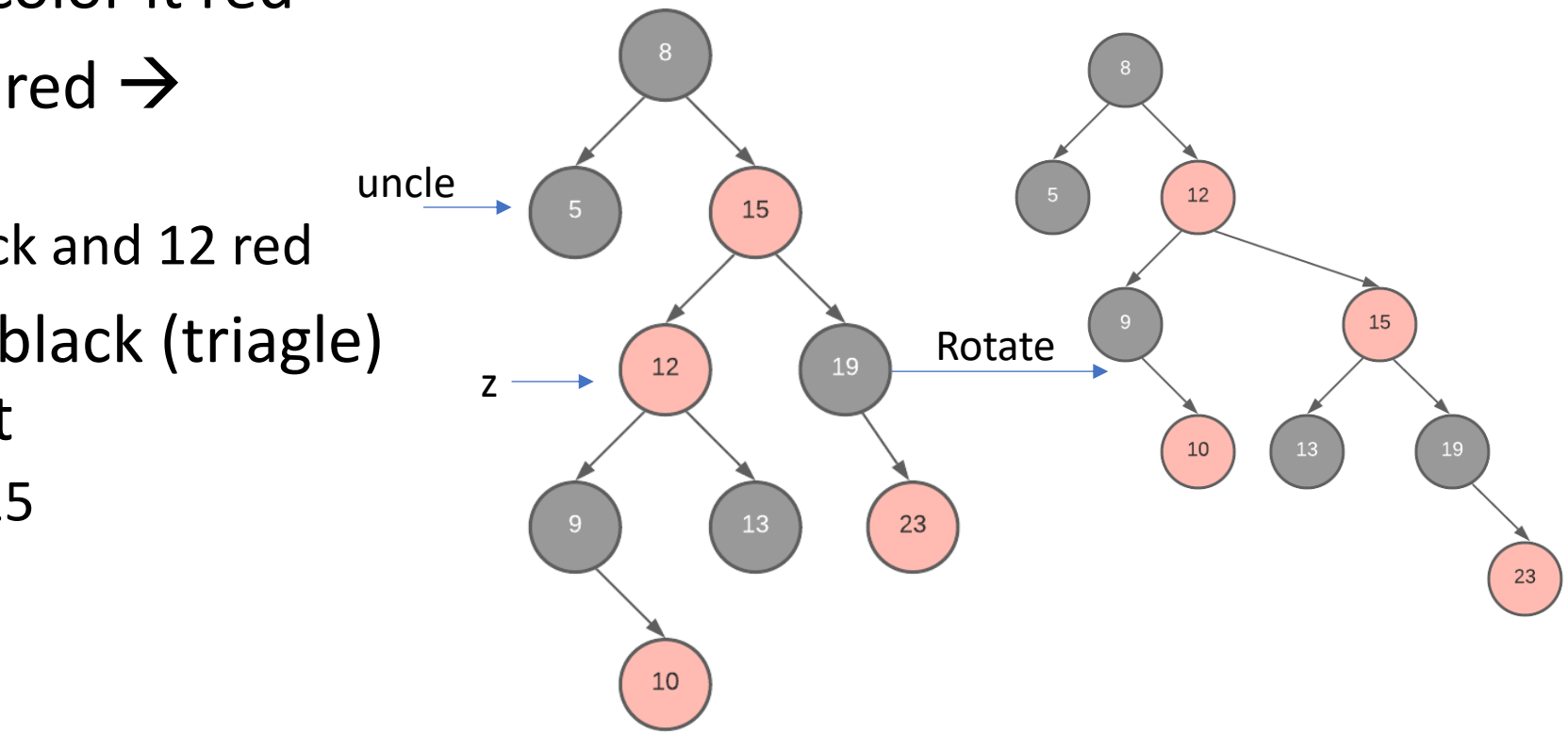
Example of Insertion

1. Insert 10 (z) and color it red
2. **Case 1:** z.uncle is red → recolor
   - Recolor 9 & 13 black and 12 red



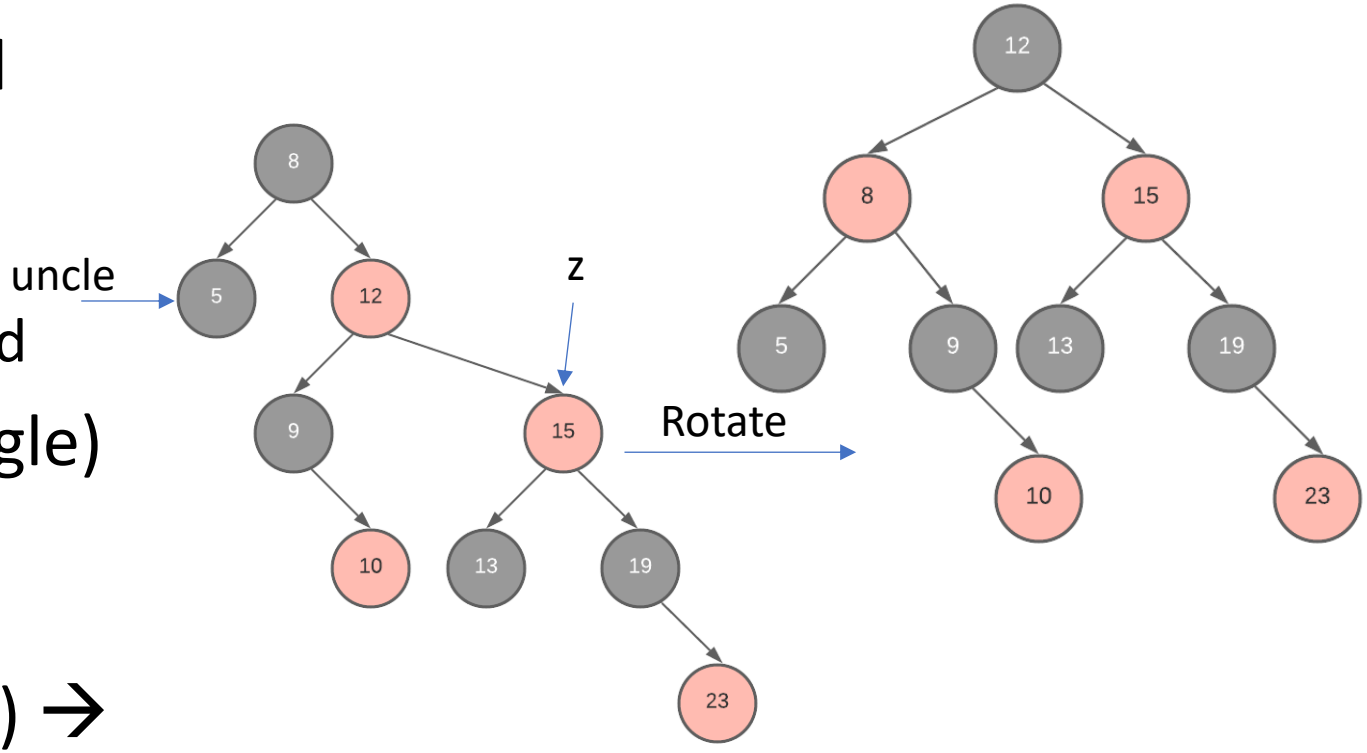
Example of Insertion

1. Insert 10 (z) and color it red
2. **Case 1:** z.uncle is red → recolor
   - Recolor 9 & 13 black and 12 red
3. **Case 2:** z.uncle = black (triagle) → rotate z.parent
   a) Rotate right on 15



Example of Insertion

1. Insert 10 (z) and color it red
2. **Case 1:** z.uncle is red → recolor
   - Recolor 9 & 13 black and 12 red
3. **Case 2:** z.uncle = black (triagle) → rotate z.parent
   a) Rotate right on 15
4. **Case 3:** z.uncle = black (line) → rotate z.grandparent & recolor
   a) Left rotate on 8 and recolor



Deletion Overview

- Deletion in Red-Black tree uses Binary Search Tree as the basis of removal
  - BST determines the successor of the node removed
- Complication comes from the Red-Black Tree rules (Next few slides will go more in depth)
  - Red Removed?
    - Won’t have any change in black heights
    - Won’t have red nodes in a row
    - Can’t have been the root
  - Black?
    - Could violate any of the root rule, red rule, or black-height rule

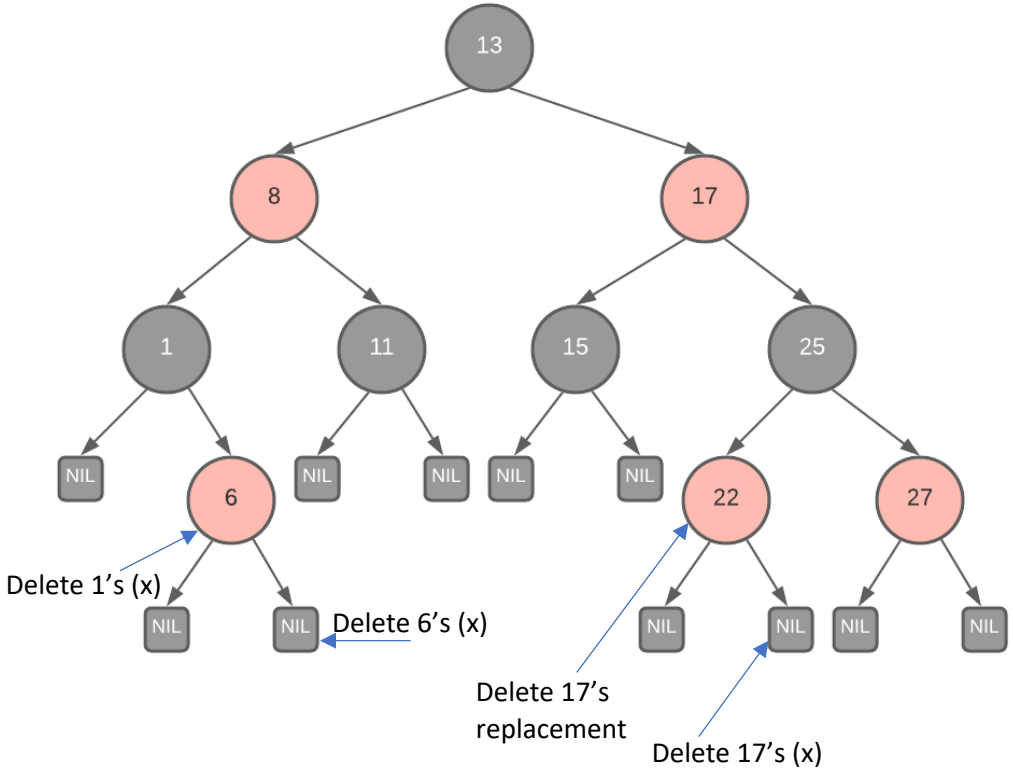
Delete Algorithm

```
RB-DELETE(T, z)
 1  y = z
 2  y-original-color = y.color
 3  if z.left == T.nil
 4       x = z.right
 5       RB-TRANSPLANT(T, z, z.right)
 6  elseif z.right == T.nil
 7       x = z.left
 8       RB-TRANSPLANT(T, z, z.left)
 9  else y = TREE-MINIMUM(z.right)
10       y-original-color = y.color
11       x = y.right
12       if y.p == z
13            x.p = y
14       else RB-TRANSPLANT(T, y, y.right)
15            y.right = z.right
16            y.right.p = y
17       RB-TRANSPLANT(T, z, y)
18       y.left = z.left
19       y.left.p = y
20       y.color = z.color
21  if y-original-color == BLACK
22       RB-DELETE-FIXUP(T, x)
```

```
RB-DELETE-FIXUP(T, x)
 1  while x ≠ T.root and x.color == BLACK
 2       if x == x.p.left
 3            w = x.p.right
 4            if w.color == RED
 5                 w.color = BLACK                          // case 1
 6                 x.p.color = RED                          // case 1
 7                 LEFT-ROTATE(T, x.p)                      // case 1
 8                 w = x.p.right                            // case 1
 9            if w.left.color == BLACK and w.right.color == BLACK
10                 w.color = RED                            // case 2
11                 x = x.p                                  // case 2
12            else if w.right.color == BLACK
13                      w.left.color = BLACK                // case 3
14                      w.color = RED                       // case 3
15                      RIGHT-ROTATE(T, w)                  // case 3
16                      w = x.p.right                       // case 3
17                 w.color = x.p.color                      // case 4
18                 x.p.color = BLACK                        // case 4
19                 w.right.color = BLACK                    // case 4
20                 LEFT-ROTATE(T, x.p)                      // case 4
21                 x = T.root                               // case 4
22       else (same as then clause with "right" and "left" exchanged)
23  x.color = BLACK
```

Deletion – Initial Steps

- Initial Steps
  - **If node deleted has 2 NIL children →** replacement x is NIL *(Ex: Delete 6)*
  - **If the node deleted has 1 NIL child and 1 non-NIL child →** replacement x is the non NIL child *(Ex: Delete 1)*
  - **If the node deleted has 2 non-NIL children →** set x to the replacement's right before the replacement is spliced out *(Ex: Delete 17)*

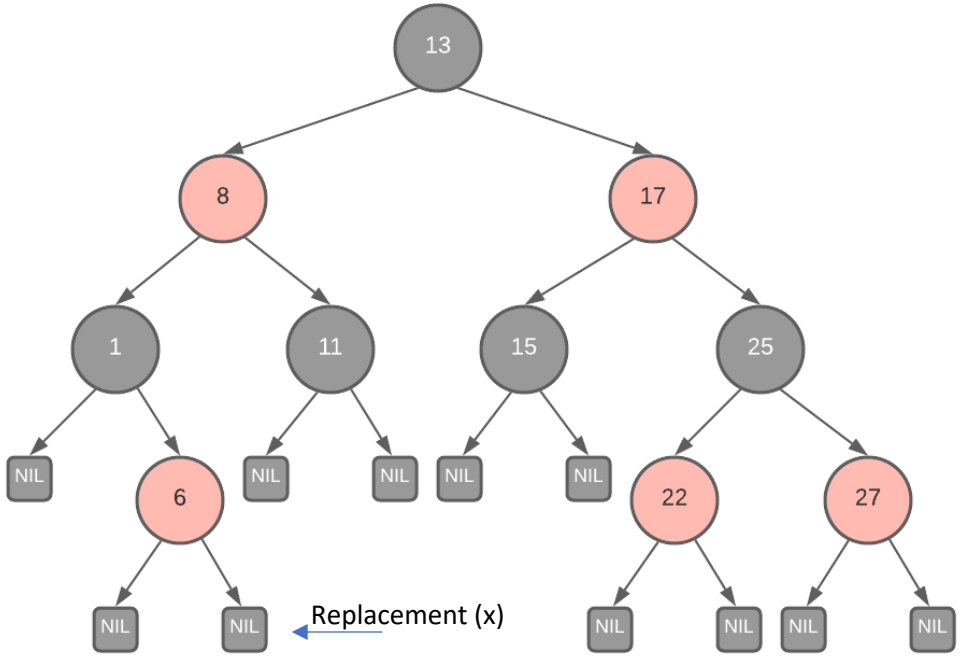


Deletion 2nd Initial Steps

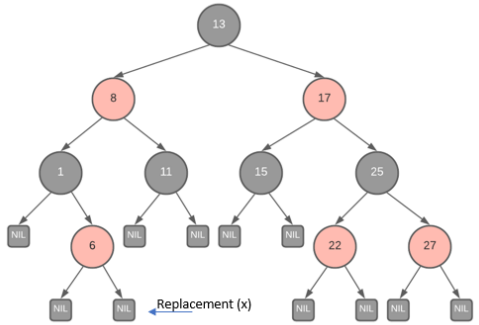
- 2nd Initial Steps
  - **If node deleted is red and it's replacement is red or NIL** → done
  - **If node deleted is red and replacement is black** → color the replacement red and proceed to the cases
  - **If the node deleted is black and its replacement is red** → color the replacement black, then done
  - **If the node deleted is black and it's replacement is NIL or black** → proceed to the appropriate case

Example 1 – 2 NIL Children

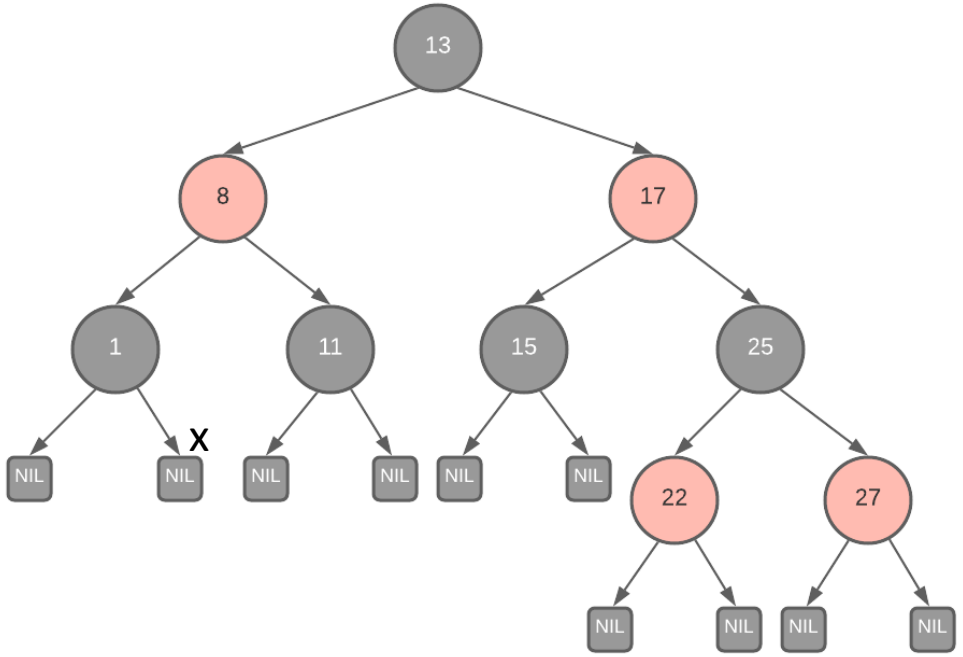
- We want to delete 6
  1. **Initial Step:** If node deleted has 2 NIL children → replacement x is NIL



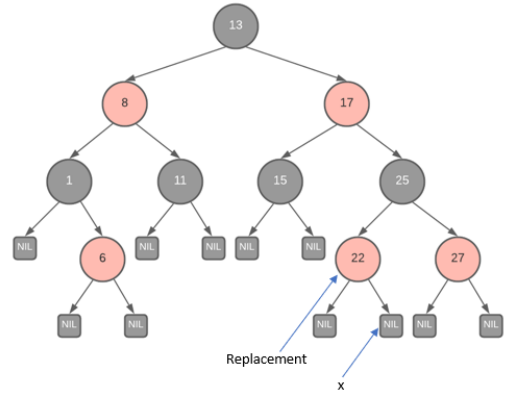
Example 1 – 2 NIL Children Cont.



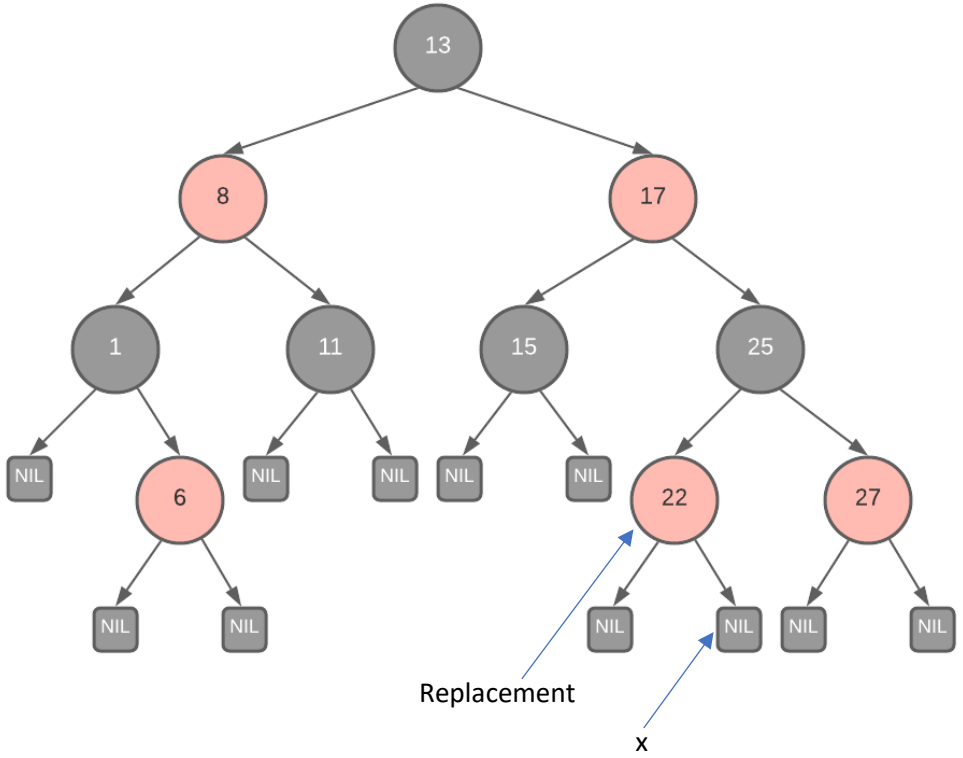
- We want to delete 6
  1. **Initial Step:** If node deleted has 2 NIL children → replacement x is NIL
  2. **2nd Initial Step:** If node deleted is red and it's replacement is red or NIL → done



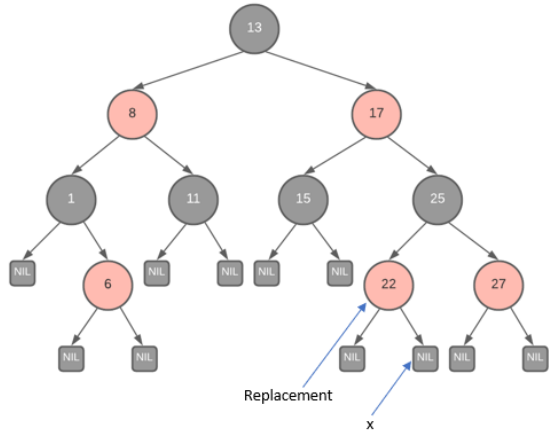
Example 2 – 2 non-NIL Children



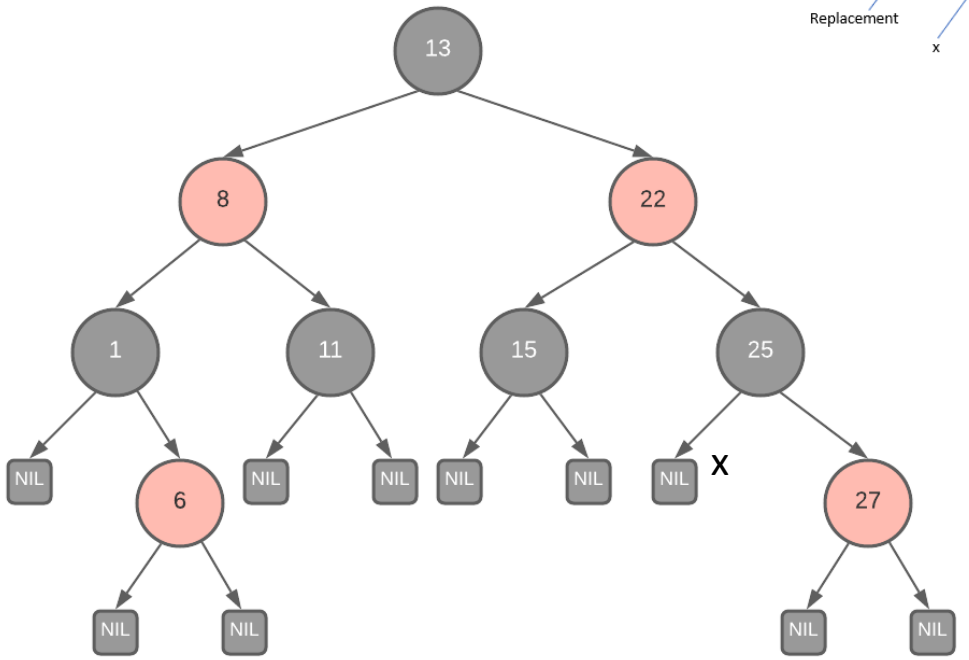
- We want to delete 17
  1. **Initial Step:** If the node deleted has 2 non-NIL children → set x to the replacement's right before the replacement is spliced out



Example 2 – 2 non-NIL Children Cont



- We want to delete 17
  1. **Initial Step:** If the node deleted has 2 non-NIL children → set x to the replacement's right before the replacement is spliced out
  2. **2nd Initial Step:** If node deleted is red and it's replacement is red or NIL → done



Deletion Cases for when replacement x is…

Case 0: Node x is red

Case 1: Node x is black and it's sibling w is red

Case 2: Node x is black and it's sibling w is black & both of w's children are black

Case 3: Node x is black & it's sibling w is black &

- If x is the left child, w's left child is red & w's right child is black
- If x is the right child, w's right child is red and w's left child is black

Case 4: Node x is black and its sibling w is black and

- If x is the left child, w's right child is red
- If x is the right child, w's left child is red

Deletion – Case 0

- Case 0: Node x is red
  1. Color x black
  2. Done

Deletion – Case 1 Node x is black and it's sibling is red

1. Color w black
2. Color x.parent red
3. Rotate x.parent
   a) If x is the left child do a left rotation
   b) If x is the right child do a right rotation
4. Change w
   a) If w is the left child set w = x.parent.right
   b) If x is the right child set w = x.parent.left
5. Based on x and the new w decide if further cases are needed

Deletion – Case 2 Node x is black and its sibling w is black and both w's children are black

1. Color w red
2. Set x = x.parent
   a) If new x is red, color x black. Then done
   b) If new x is black, see if further cases are needed.

Deletion Case 3 Node x is black and its sibling w is black and …do the following…

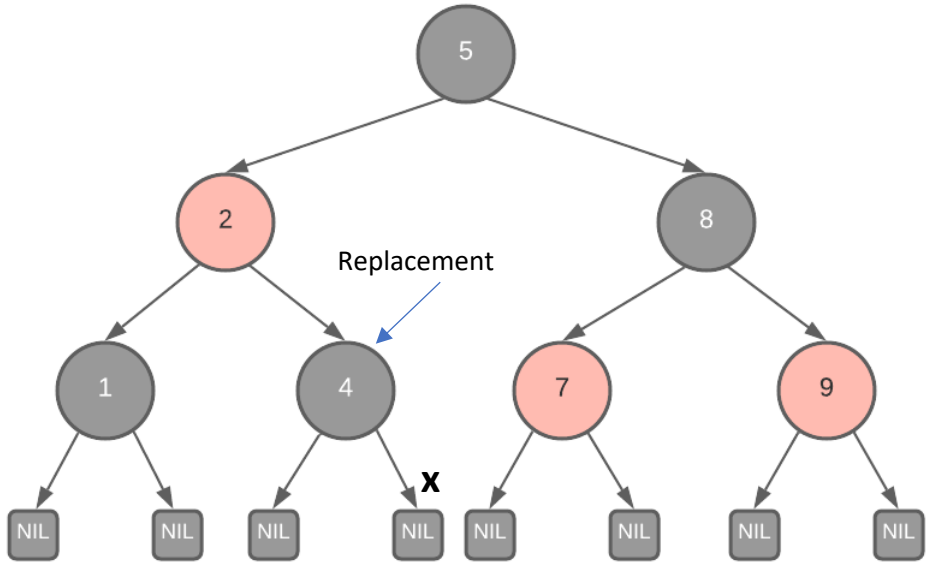
1. Color w's child black
   a) If x is the left child, color w.left black
   b) If x is the right child, color w.right black
2. Color w red
3. Rotate w
   a) If x is the left child do a right rotation
   b) If x is the right child do a left rotation
4. Now we have to change w
   a) If x is the left child set w = x.parent.right
   b) If x is the right child set w = x.parent.left
5. Proceed to case 4.

Deletion – Case 4 Node x is black and its sibling w is black and … do the following…

1. Color w the same color as x.parent
2. Color x.parent black
3. Color w's child black
   a) If x is the left child, color w:right black
   b) If x is the right child, color w:left black
4. Rotate x.parent
   a) If x is the left child do a left rotation
   b) If x is the right child do a right rotation
5. We are done.

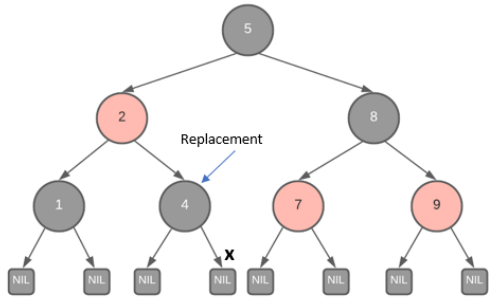
Example 1

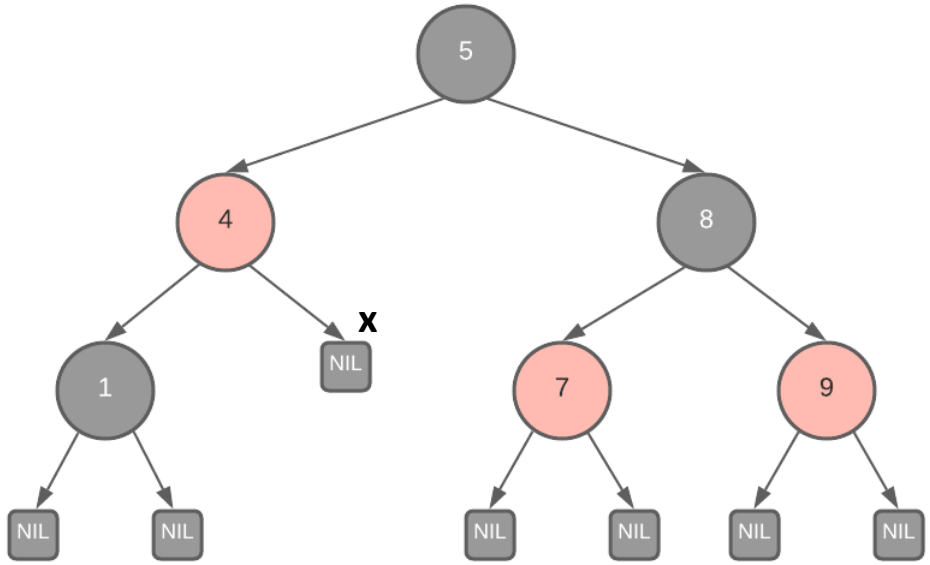
- We want to delete 2
  - **Initial Step:** If the node deleted has 2 non-NIL children → set x to the replacement's right before the replacement is spliced out



Example 1

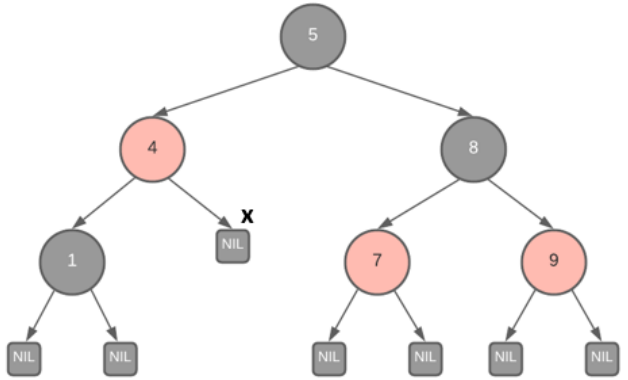
- We want to delete 2
  - **Initial Step:** If the node deleted has 2 non-NIL children → set x to the replacement's right before the replacement is spliced out
  - **2nd Initial Step:** If node deleted is red and replacement is black → color the replacement red and proceed to the cases

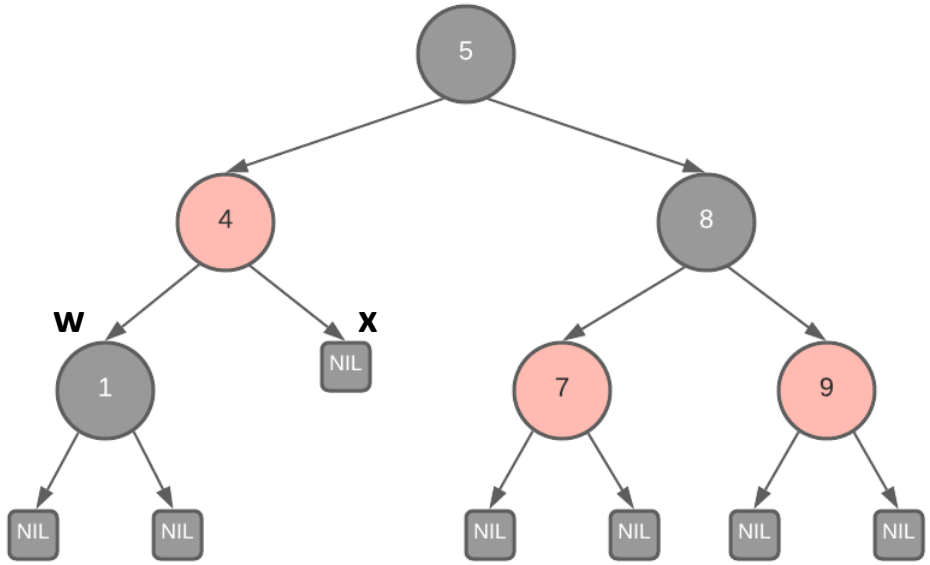




Example 1

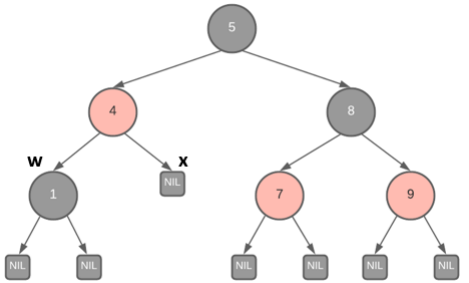
- We want to delete 2
  - **Initial Step:** If the node deleted has 2 non-NIL children → set x to the replacement's right before the replacement is spliced out
  - **2nd Initial Step:** If node deleted is red and replacement is black → color the replacement red and proceed to the cases
  - **Choose Case:** Case 2 – Node x is black and it's sibling w is black & both of w's children are black



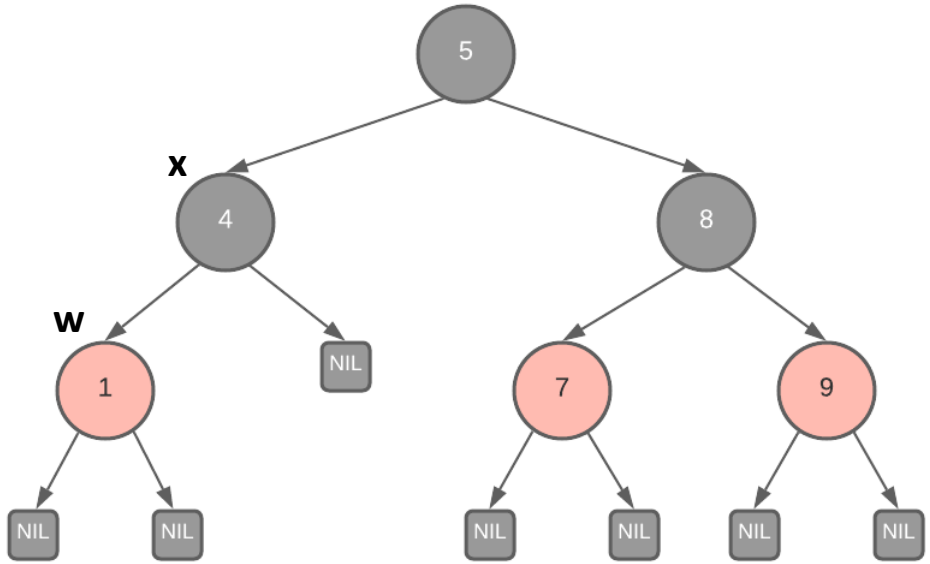


Example 1

- We want to delete 2
  - **Choose Case:** Case 2 – Node x is black and it's sibling w is black & both of w's children are black
    1. Color w red
    2. Set x = x.parent
       a) If new x is red, color x black. Then done

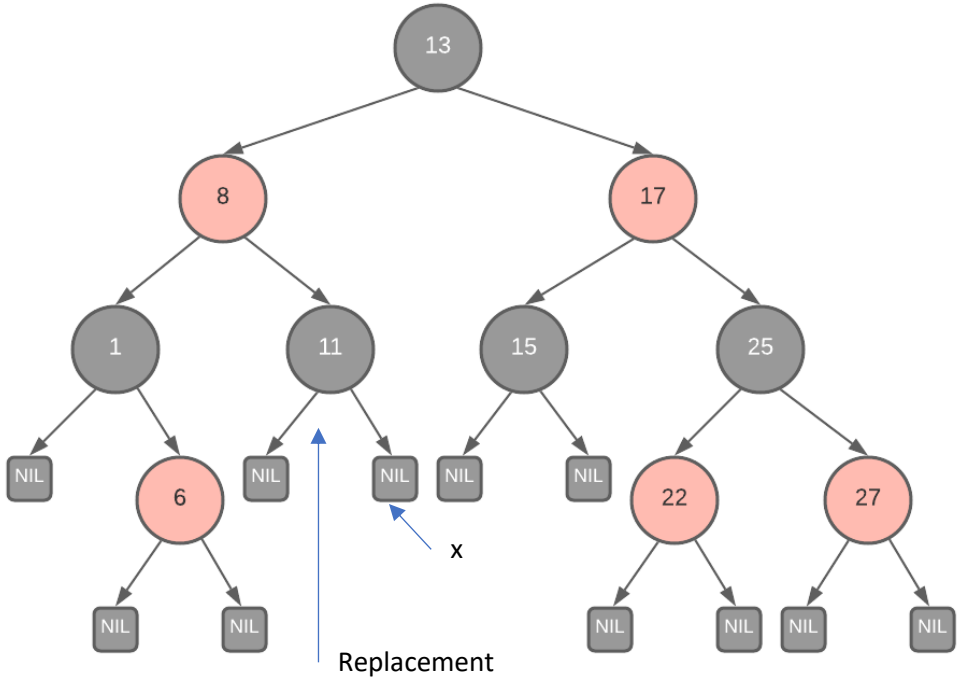


Previous Step

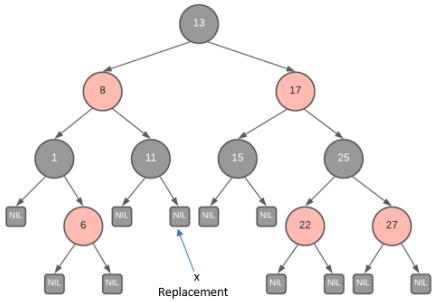


Example 2

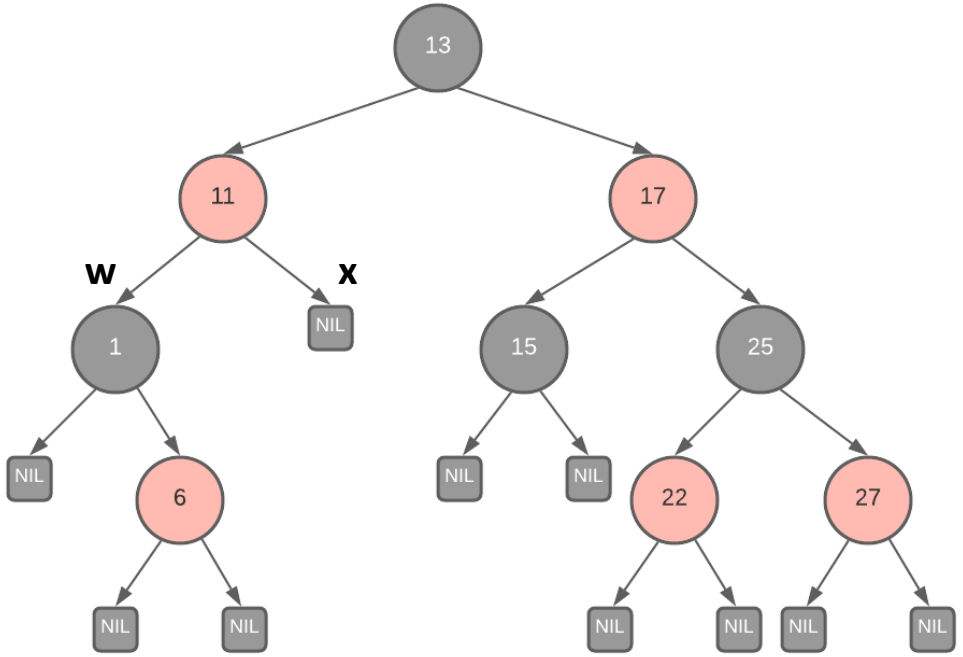
- We want to delete 8
  1. **Initial Step:** If the node deleted has 2 non-NIL children → set x to the replacement's right before the replacement is spliced out



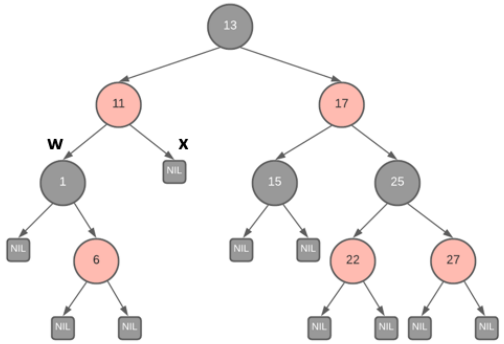
Example 2



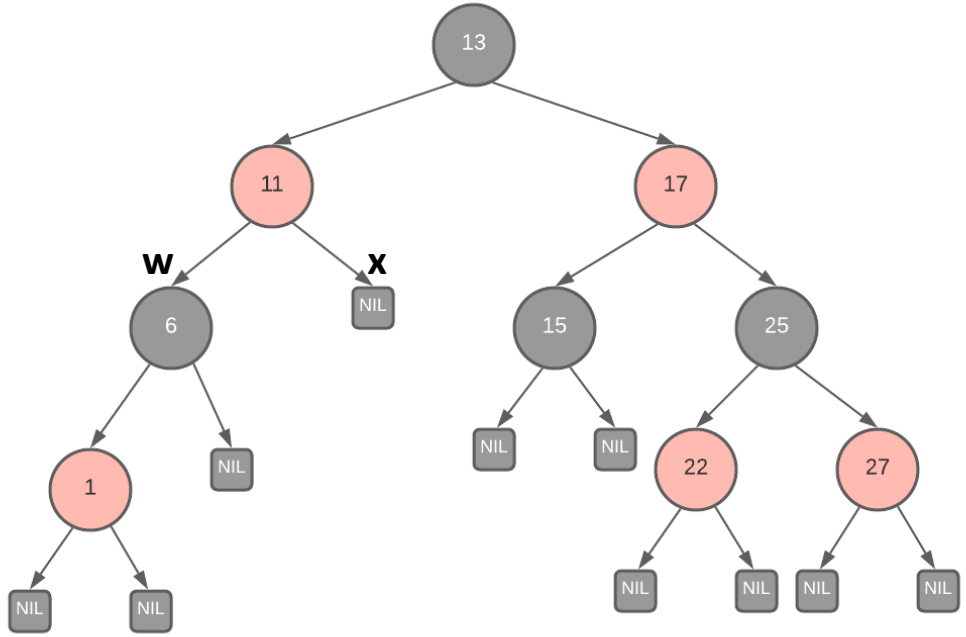
- We want to delete 8
  1. **Initial Step:** If the node deleted has 2 non-NIL children → set x to the replacement's right before the replacement is spliced out
  2. **2nd Initial Step:** If node deleted is red and replacement is black → color the replacement red and proceed to the cases
  3. **Choose Case:** Case 3 Node x is black & it's sibling w is black



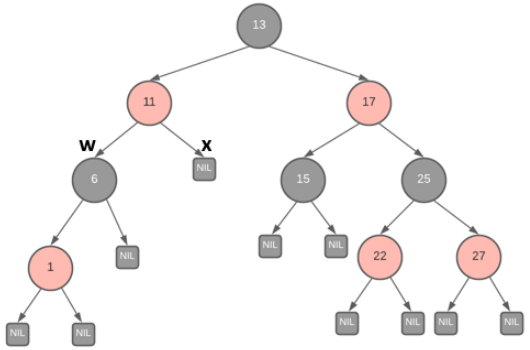
Example 2



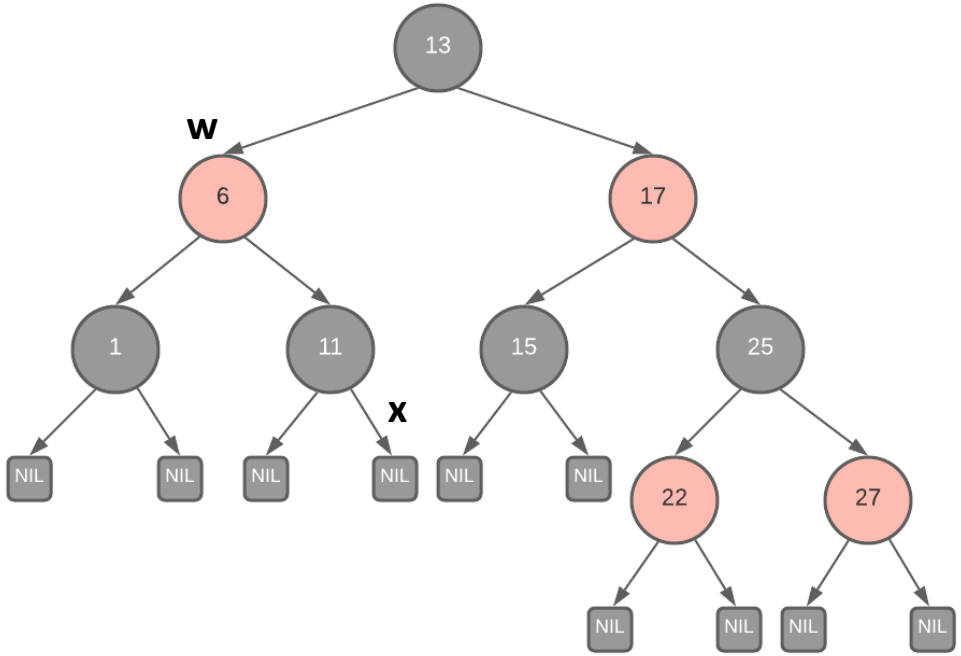
- We want to delete 8
  1. **Choose Case:** Case 3 Node x is black & it's sibling w is black
     b) If x is the right child, color w.right black
  2. Color w red
  3. Rotate w
     b) If x is the right child do a left rotation
  4. Now we have to change w
     b) If x is the right child set w = x.parent.left
  5. Proceed to case 4.



Example 2



- We want to delete 8
  1. **Choose Case:** Case 4
  2. Color w the same color as x.parent
  3. Color x.parent black
  4. Color w's child black
     b) If x is the right child, color w.left black
  5. Rotate x.parent
     b) If x is the right child do a right rotation
  6. We are done.



Readings

- Chapter 12 of CLRS Book.

Cormen, Thomas H., et al. *Introduction to Algorithms, 3e* MIT Press, 2014.

- Red Black Tree Visualization

https://www.cs.usfca.edu/~galles/visualization/RedBlack.html

https://yongdanielliang.github.io/animation/web/RBTree.html

https://www.cs.csubak.edu/~msarr/visualizations/RedBlack.html