Completely Fair Scheduler

Dr. Naser Al Madi

Learning objectives

- Some project feedback and software design tips
- Completely Fair Scheduler
- Red-black Trees

Project feedback

*I will continue to return graded work before the next project is due.

How to organize your notebook

Coupling, cohesion, and dependency injection

- SOLID Design: The Open-Close Principle

operating_system.kernel(scheduler.FCFS_scheduler)

vs.

operating_system.kernel(“FCFS”)

Pycodestyle

https://www.python.org/dev/peps/pep-0008/

Black

Pytest

Completely Fair Scheduling

Completely Fair Scheduling (CFS)

- Linux scheduler since 2.6.23 (2007)
- No heuristics.
- Elegant handling of I/O and CPU bound processes.
- By Ingo Molnár based on work from Con Kolivas

Ideal fairness

- Divide processor time equally among processes.

If there are N processes in the system, each process should get (100/N)% of the CPU time.

Ideal fairness

- Divide processor time equally among processes.

If there are N processes in the system, each process should get (100/N)% of the CPU time.

process	Burst (ms)
A	8
B	4
C	16
D	4

A	1	2	3	4	6	8			
B	1	2	3	4					
C	1	2	3	4	6	8	12	16	
D	1	2	3	4					

Ideal fairness

- Divide processor time equally among processes.

If there are N processes in the system, each process should get (100/N)% of the CPU time.

process	Burst (ms)
A	8
B	4
C	16
D	4

4 ms slice: each process got 4/4 = 1ms or CPU time

A	1	2	3	4	6	8			
B	1	2	3	4					
C	1	2	3	4	6	8	12	16	
D	1	2	3	4					

Ideal fairness

- Divide processor time equally among processes.

If there are N processes in the system, each process should get (100/N)% of the CPU time.

process	Burst (ms)
A	8
B	4
C	16
D	4

4 ms slice: each process got 4/2 = 2ms or CPU time

A	1	2	3	4	6	8			
B	1	2	3	4					
C	1	2	3	4	6	8	12	16	
D	1	2	3	4					

Ideal fairness

- Divide processor time equally among processes.

If there are N processes in the system, each process should get (100/N)% of the CPU time.

process	Burst (ms)
A	8
B	4
C	16
D	4

4 ms slice: each process got 4/1 = 4ms or CPU time

A	1	2	3	4	6	8			
B	1	2	3	4					
C	1	2	3	4	6	8	12	16	
D	1	2	3	4					

Virtual Runtime

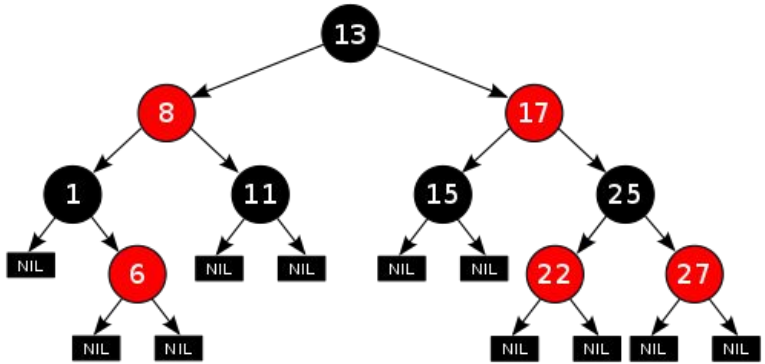
- In the PCB of each process a value called virtual runtime is added (vruntime).
- At each scheduling point, if process has run for t ms, then (vruntime += t).
- Vruntime for a process therefore monotonically increases.
- Implemented as unsigned 64 bit integer.

CFS main concept

- When timer interrupt occurs:
  - Choose the task with the lowest vruntime.
  - Compute its **dynamic timeslice**.
  - Program the high resolution time with this timeslice.
- The process begins to execute in the CPU.
- When interrupt occurs again, context switch if there is another task with a smaller runtime.

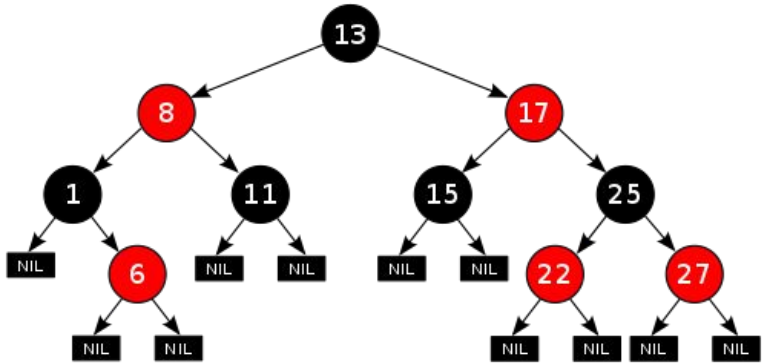
Picking the next process to run

- CFS uses a red-black tree.
  - Each node in the tree represents a runnable task.
  - Nodes ordered according to their vruntime
  - Nodes on the left have lower vruntime
  - The left most node is the task with the least vruntime



Picking the next process to run

- At a context switch:
  - Pick the left most node in tree in O(1)
  - If the finished process is runnable, it is inserted into the tree depending on the new vruntime in O(log(n))
  - Tasks move from left to right of the tree after execution. Starvation is avoided.



Why red black tree?

- Self balancing
- All operations are O(log n)

I/O and CPU bound processes

- What we need,
  - I/O bound should get higher priority and get a longer time to execute compared to CPU bound.
  - CFS achieves this efficiently:
    - I/O bound processes have small CPU bursts therefore will have a low vruntime.
    - They would appear towards the left of the tree, thus are given higher priorities.
    - I/O bound processes will typically have larger time slices because they have smaller vruntime.

Priority in CFS

Priority used to weigh the vruntime with nice value

- If process has run for t ms, then:

$$vruntime \mathrel{+}= t * (weight)$$

- A low priority means that time moves at a faster rate compared to high priority task.

New processes

- Gets added to the RB-tree
- Starts with an initial value of min_vruntime
- This ensures that it gets to execute quickly

Red-Black Trees

Let’s start by refreshing our memory!!

Motivating example:

You have a collection of data items, and you want to find a specific element in the data:

5	1	3	2	7	8	0	6

Motivating example:

You have a collection of data items, and you want to find a specific element in the data:

5	1	3	2	7	8	0	6

Let’s assume that the data structure is an array, what is the worst case time complexity to find the element?

Motivating example:

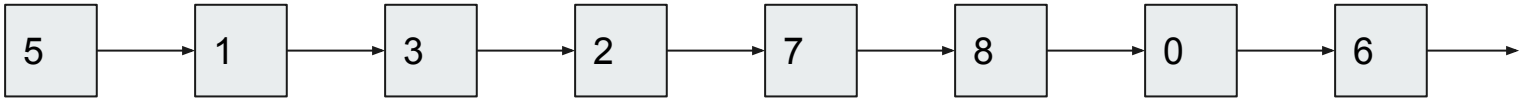
You have a collection of data items, and you want to find a specific element in the data:

5	1	3	2	7	8	0	6

Let's assume that the data structure is an array, what is the worst case time complexity to find the element? **O(n)** good job!

Motivating example:

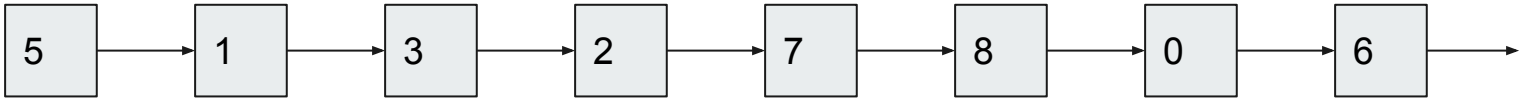
You have a collection of data items, and you want to find a specific element in the data:



What about a linked list? What is the worst case time complexity for search?

Motivating example:

You have a collection of data items, and you want to find a specific element in the data:



What about a linked list? What is the worst case time complexity for search? Also **O(n)** if the element we're looking for is the last element in the list.

Average time complexity

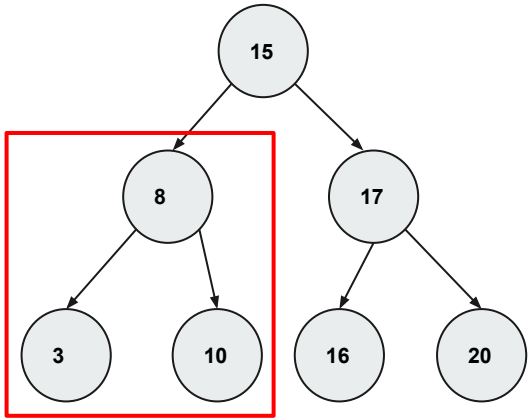
Data structure	search
Array	O(n)
Linkedlist	O(n)
Binary search tree	O(log(n))

Binary Search Trees

- Binary search tree is a type of binary trees, therefore each node has at most 2 children.

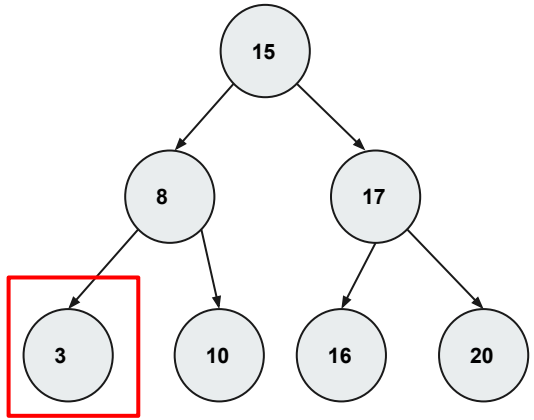
Binary Search Trees

- Binary search tree is a type of binary trees, therefore each node has at most 2 children.
- All values on the left subtree are less than the value of the node.



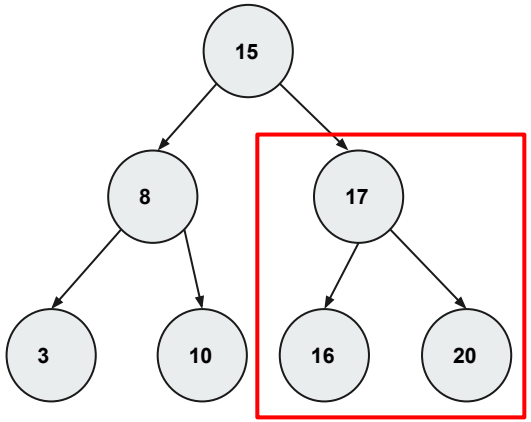
Binary Search Trees

- Binary search tree is a type of binary trees, therefore each node has at most 2 children.
- All values on the left subtree are less than the value of the node.



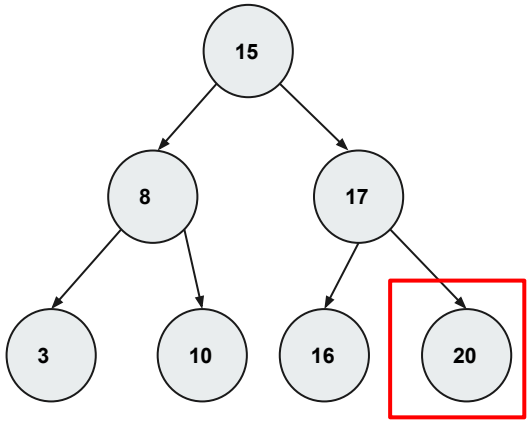
Binary Search Trees

- Binary search tree is a type of binary trees, therefore each node has at most 2 children.
- All values on the left subtree are less than the value of the node.
- All values on the right subtree are greater than the value of the node.



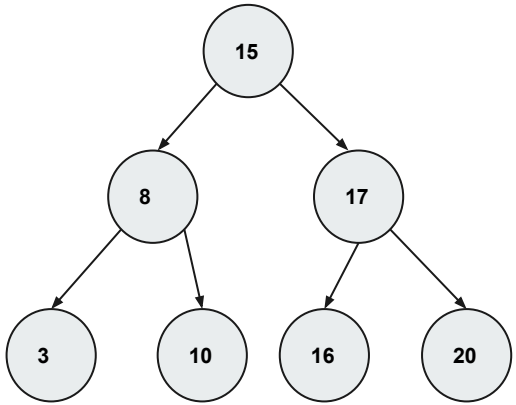
Binary Search Trees

- Binary search tree is a type of binary trees, therefore each node has at most 2 children.
- All values on the left subtree are less than the value of the node.
- All values on the right subtree are greater than the value of the node.



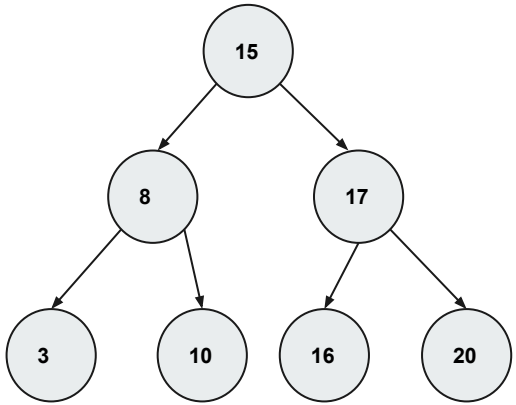
Binary Search Trees

- Binary search tree is a type of binary trees, therefore each node has at most 2 children.
- All values on the left subtree are less than the value of the node.
- All values on the right subtree are greater than the value of the node.
- The left and right subtree each must also be binary search trees.



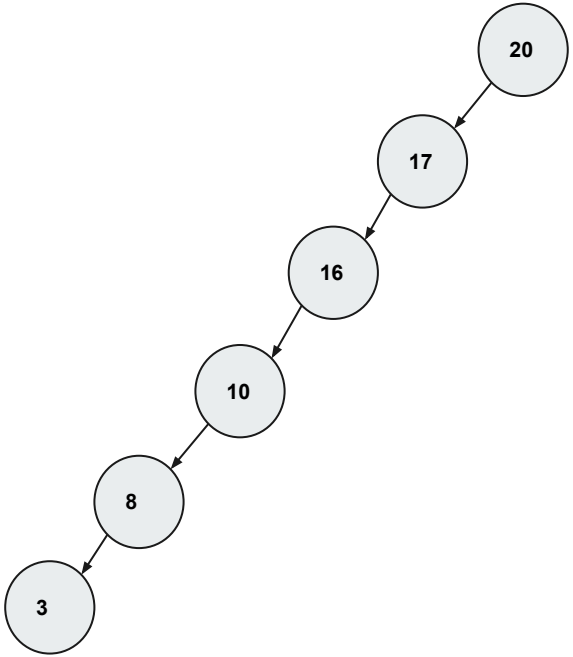
BST search performance:

- Based on the tree structure we can search without visiting every single node.
- To search for any value we use the following logic:
  - Start at root.
  - Compare node value to search item, if search item is greater go right.
  - If search item is smaller then go left.



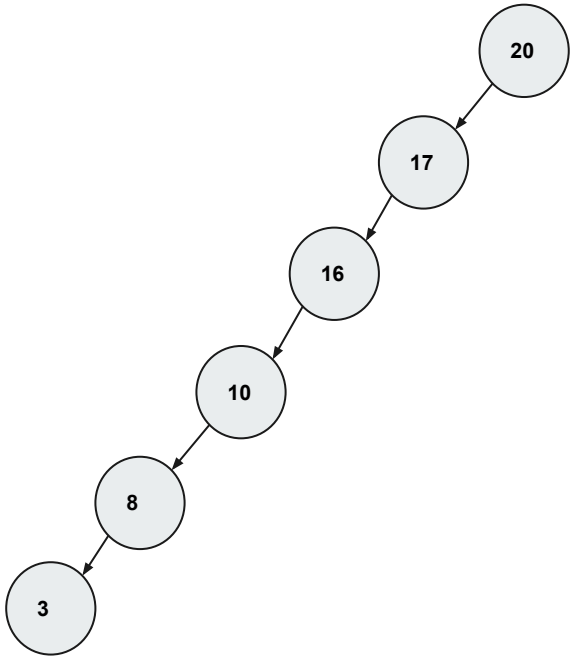
BST search performance:

- Is this a binary search tree?



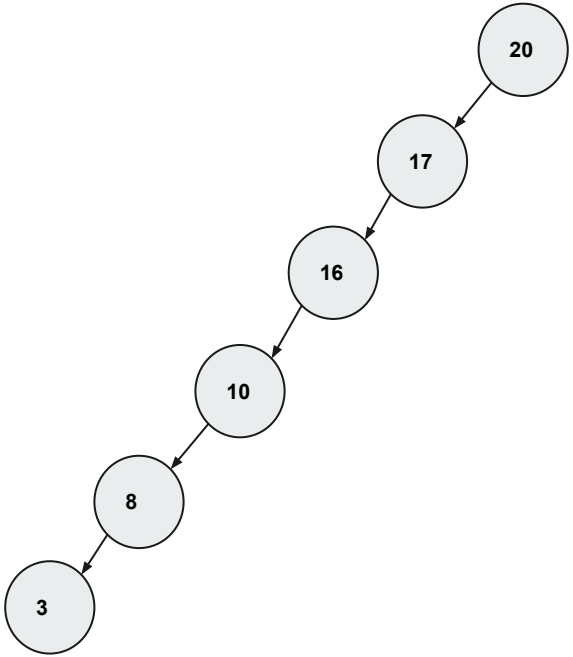
BST search performance:

- Is this a binary search tree?
- Yes! Since each left subtree is smaller than the value of the node.



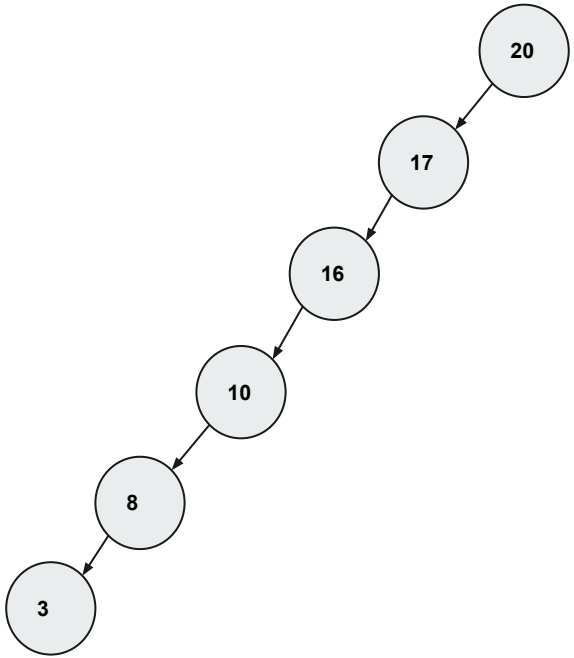
BST search performance:

- What is the time complexity for searching this BST?



BST search performance:

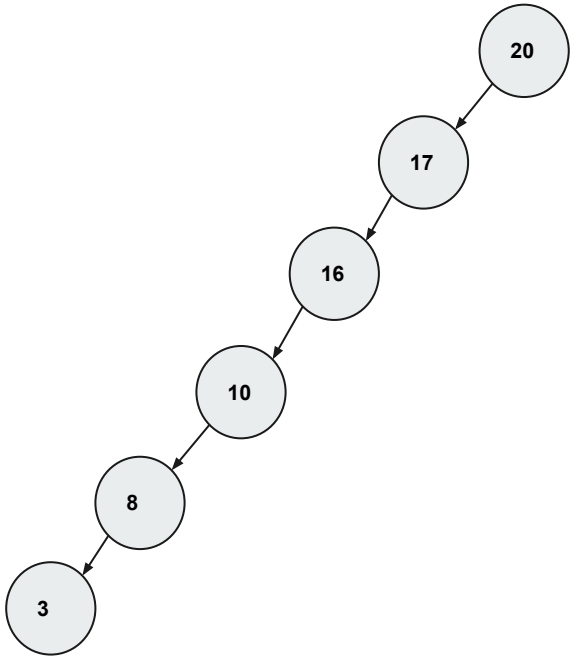
- What is the time complexity for searching this BST?
- O(n) 😭



BST search performance:

- What is the time complexity for searching this BST?
- O(n) 😭

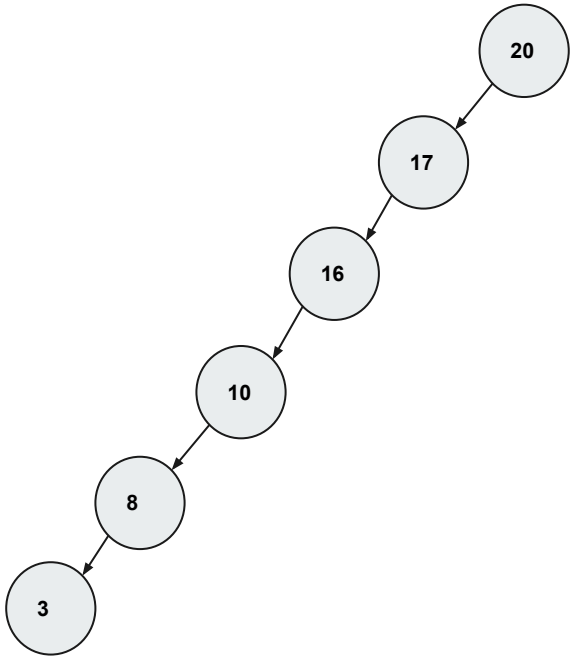
- In other words, the big O time complexity is equal to the height of the tree.



BST search performance:

- What is the time complexity for searching this BST?
- O(n) 😭

- That's why we prefer a balanced BST.



Balanced Binary Search Tree

In addition to all the properties of a BST.

A binary search tree is balanced if and only if the depth of the two subtrees of every node never differ by more than 1.

Balanced Binary Search Tree

In addition to all the properties of a BST.

A binary search tree is balanced if and only if the depth of the two subtrees of every node never differ by more than 1.

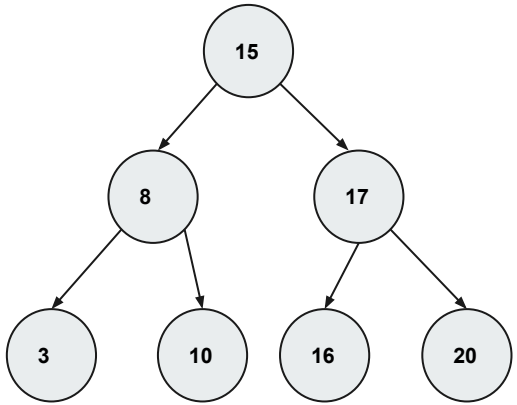
|depth of left subtree - depth of right subtree| <= 1

Balanced Binary Search Tree Example

In addition to all the properties of a BST.

A binary search tree is balanced if and only if the depth of the two subtrees of every node never differ by more than 1.

|depth of left subtree - depth of right subtree| <= 1

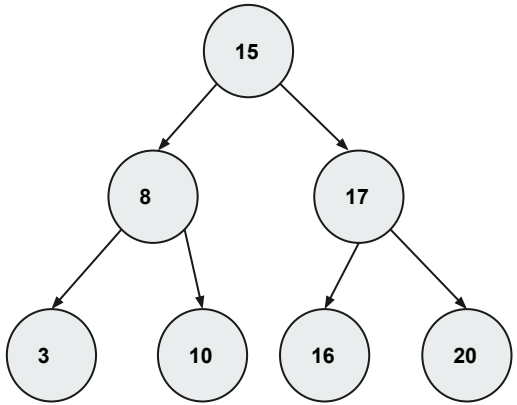


Balanced Binary Search Tree Example

In addition to all the properties of a BST.

A binary search tree is balanced if and only if the depth of the two subtrees of every node never differ by more than 1.

|depth of left subtree - depth of right subtree| <= 1



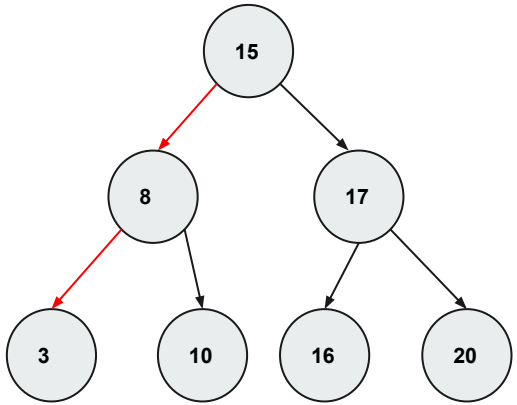
Balanced Binary Search Tree Example

In addition to all the properties of a BST.

A binary search tree is balanced if and only if the depth of the two subtrees of every node never differ by more than 1.

|depth of left subtree - depth of right subtree| <= 1

| 2 - |



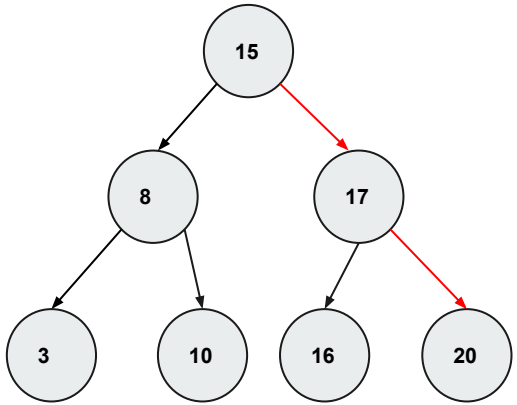
Balanced Binary Search Tree Example

In addition to all the properties of a BST.

A binary search tree is balanced if and only if the depth of the two subtrees of every node never differ by more than 1.

|depth of left subtree - depth of right subtree| <= 1

| 2 - 2 |



Balanced Binary Search Tree Example

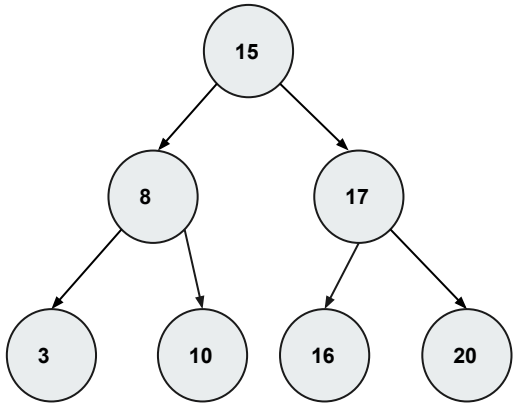
In addition to all the properties of a BST.

A binary search tree is balanced if and only if the depth of the two subtrees of every node never differ by more than 1.

|depth of left subtree - depth of right subtree| <= 1

| 2 - 2 | <= 1

balanced

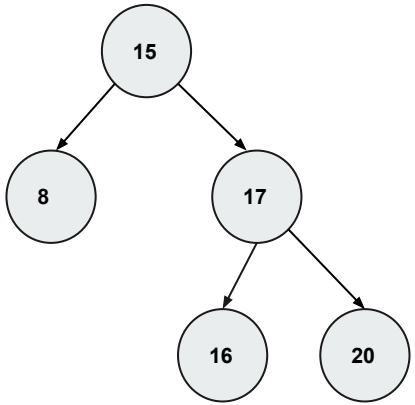


Balanced Binary Search Tree Example 2

In addition to all the properties of a BST.

A binary search tree is balanced if and only if the depth of the two subtrees of every node never differ by more than 1.

|depth of left subtree - depth of right subtree| <= 1



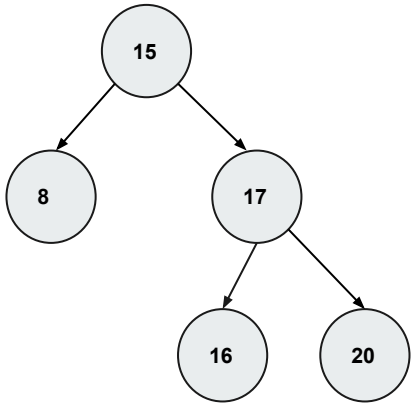
Balanced Binary Search Tree Example 2

In addition to all the properties of a BST.

A binary search tree is balanced if and only if the depth of the two subtrees of every node never differ by more than 1.

|depth of left subtree - depth of right subtree| <= 1

| 1 - 2| <= 1



Balanced Binary Search Tree Example 2

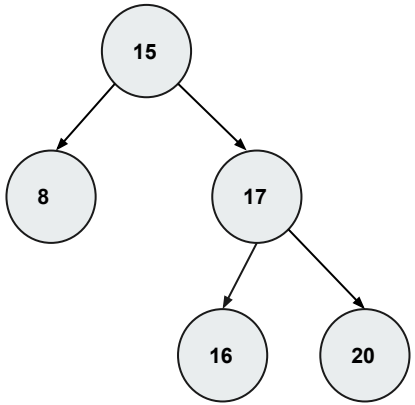
In addition to all the properties of a BST.

A binary search tree is balanced if and only if the depth of the two subtrees of every node never differ by more than 1.

|depth of left subtree - depth of right subtree| <= 1

| 1 - 2| <= 1

balanced

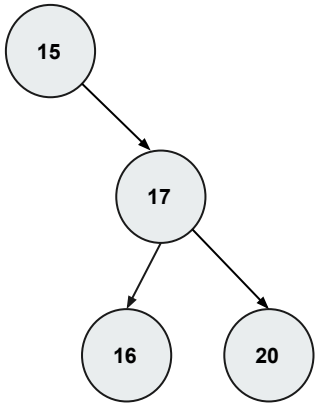


Balanced Binary Search Tree Example 3

In addition to all the properties of a BST.

A binary search tree is balanced if and only if the depth of the two subtrees of every node never differ by more than 1.

|depth of left subtree - depth of right subtree| <= 1



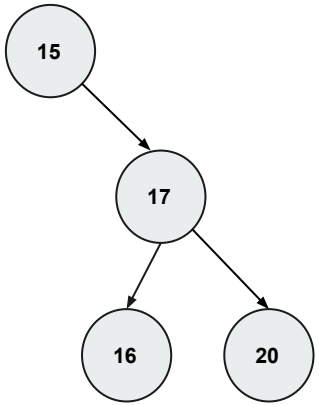
Balanced Binary Search Tree Example 3

In addition to all the properties of a BST.

A binary search tree is balanced if and only if the depth of the two subtrees of every node never differ by more than 1.

|depth of left subtree - depth of right subtree| <= 1

| 0 - 2 | <= 1



Balanced Binary Search Tree Example 3

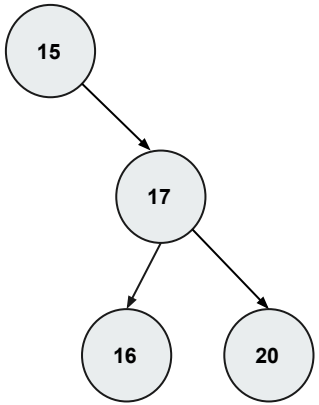
In addition to all the properties of a BST.

A binary search tree is balanced if and only if the depth of the two subtrees of every node never differ by more than 1.

|depth of left subtree - depth of right subtree| <= 1

| 0 - 2 | <= 1

Not balanced



Red-black tree

- Special type of binary search trees
- **Approximately** balanced
- All operations in O(log(n))

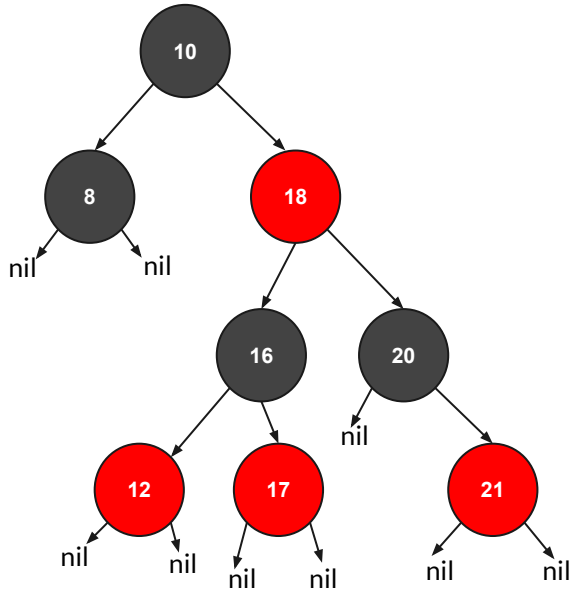
Better than heaps in:

- Delete any node in O(log(n))
- Not based on array implementation (no need for sequential memory)
- We can easily modify it to delete node with *min_vruntime* in O(1)

Red-black tree properties

*in addition to the properties of binary search trees

1. Every node is either black or red
2. Root and leaves (nil) are black
3. If a node is red, its children are black
4. All paths from a node to its nil descendants contain the same number of black nodes



Operations

- Search
- Insert
- Remove

Operations

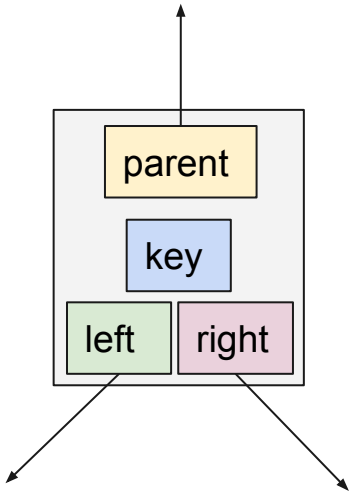
- Search
- Insert
- Remove

The two operations mess up the properties of a red-black tree, we solve this by rotation and recoloring.

RBTree node

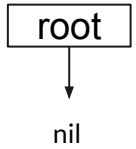
Each RBTree node keeps track of:

- Its value or key
- Left subtree
- Right subtree
- Its parent



Insert operation pseudocode

```
RB-Insert(T,z)
  y = nil[T]
  x = root[T]
  while x != nil[T]
     y = x
     if key[z] < key[x] then
        x = left[x]
     else
        x = right[x]
  p[z] = y
  if y = nil[T]
     root[T] = z
  else
    if key[z] < key[y] then
      left[y] = z
    else
      right[y] = z
  left[z] = nil[T]
  right[z] = nil[T]
  color[z] = RED
  RB-Insert-fixup(T,z)
```

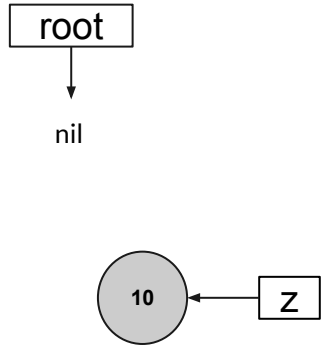


Assuming:

- Z is a node
- T is the tree

Insert operation pseudocode

```
RB-Insert(T,z)
  y = nil[T]
  x = root[T]
  while x != nil[T]
     y = x
     if key[z] < key[x] then
        x = left[x]
     else
        x = right[x]
  p[z] = y
  if y = nil[T]
     root[T] = z
  else
    if key[z] < key[y] then
      left[y] = z
    else
      right[y] = z
  left[z] = nil[T]
  right[z] = nil[T]
  color[z] = RED
  RB-Insert-fixup(T,z)
```

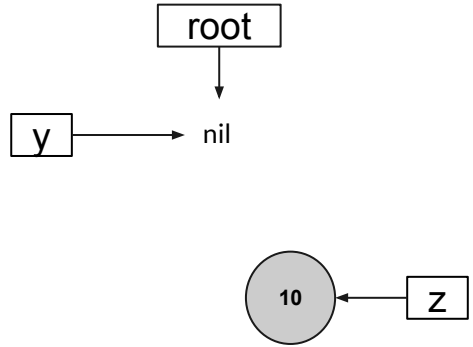


Assuming:

- Z is a node
- T is the tree

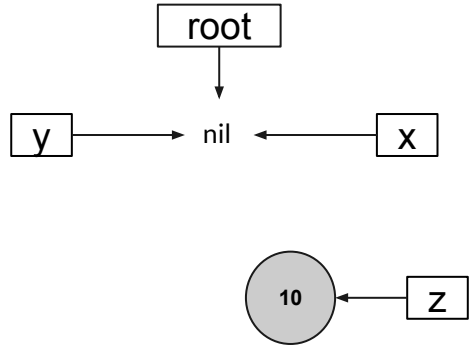
Insert operation pseudocode

```
RB-Insert(T,z)
  y = nil[T]
  x = root[T]
  while x != nil[T]
     y = x
     if key[z] < key[x] then
        x = left[x]
     else
        x = right[x]
  p[z] = y
  if y = nil[T]
     root[T] = z
  else
    if key[z] < key[y] then
      left[y] = z
    else
      right[y] = z
  left[z] = nil[T]
  right[z] = nil[T]
  color[z] = RED
  RB-Insert-fixup(T,z)
```



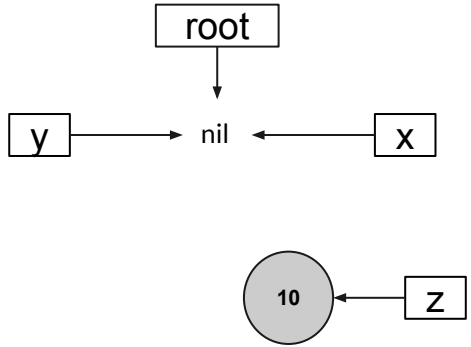
Insert operation pseudocode

```
RB-Insert(T,z)
  y = nil[T]
  x = root[T]
  while x != nil[T]
     y = x
     if key[z] < key[x] then
        x = left[x]
     else
        x = right[x]
  p[z] = y
  if y = nil[T]
     root[T] = z
  else
    if key[z] < key[y] then
      left[y] = z
    else
      right[y] = z
  left[z] = nil[T]
  right[z] = nil[T]
  color[z] = RED
  RB-Insert-fixup(T,z)
```



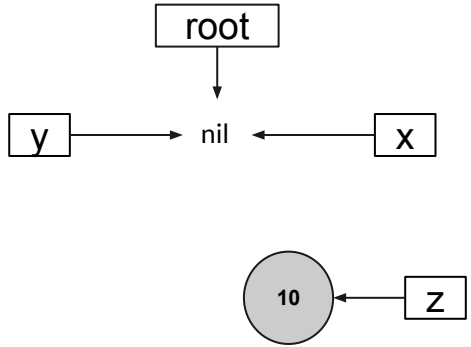
Insert operation pseudocode

```
RB-Insert(T,z)
  y = nil[T]
  x = root[T]
  while x != nil[T]
      y = x
      if key[z] < key[x] then
        x = left[x]
      else
        x = right[x]
  p[z] = y
  if y = nil[T]
     root[T] = z
  else
    if key[z] < key[y] then
      left[y] = z
    else
      right[y] = z
  left[z] = nil[T]
  right[z] = nil[T]
  color[z] = RED
  RB-Insert-fixup(T,z)
```



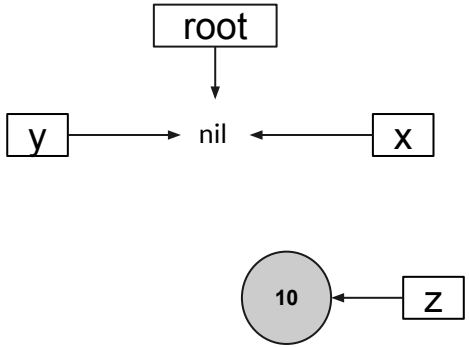
Insert operation pseudocode

```
RB-Insert(T,z)
  y = nil[T]
  x = root[T]
  while x != nil[T]
      y = x
      if key[z] < key[x] then
        x = left[x]
      else
        x = right[x]
  p[z] = y
  if y = nil[T]
     root[T] = z
  else
    if key[z] < key[y] then
      left[y] = z
    else
      right[y] = z
  left[z] = nil[T]
  right[z] = nil[T]
  color[z] = RED
  RB-Insert-fixup(T,z)
```



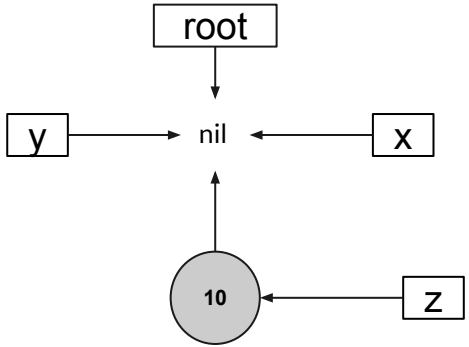
Insert operation pseudocode

```
RB-Insert(T,z)
  y = nil[T]
  x = root[T]
  while x != nil[T]
      y = x
      if key[z] < key[x] then
        x = left[x]
      else
        x = right[x]
  p[z] = y
  if y = nil[T]
     root[T] = z
  else
    if key[z] < key[y] then
      left[y] = z
    else
      right[y] = z
  left[z] = nil[T]
  right[z] = nil[T]
  color[z] = RED
  RB-Insert-fixup(T,z)
```



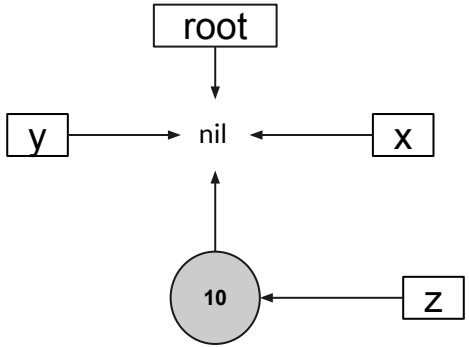
Insert operation pseudocode

```
RB-Insert(T,z)
   y = nil[T]
   x = root[T]
   while x != nil[T]
      y = x
      if key[z] < key[x] then
         x = left[x]
      else
         x = right[x]
   p[z] = y
   if y = nil[T]
      root[T] = z
   else
     if key[z] < key[y] then
       left[y] = z
     else
       right[y] = z
   left[z] = nil[T]
   right[z] = nil[T]
   color[z] = RED
   RB-Insert-fixup(T,z)
```



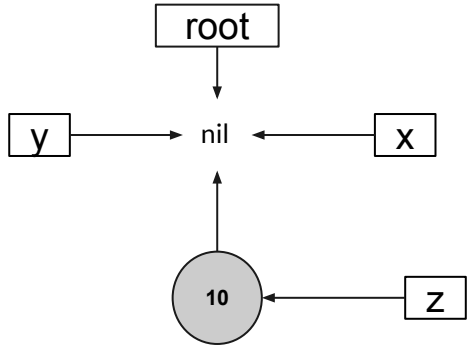
Insert operation pseudocode

```
RB-Insert(T,z)
  y = nil[T]
  x = root[T]
  while x != nil[T]
     y = x
     if key[z] < key[x] then
        x = left[x]
     else
        x = right[x]
  p[z] = y
  if y = nil[T]
     root[T] = z
  else
    if key[z] < key[y] then
      left[y] = z
    else
      right[y] = z
  left[z] = nil[T]
  right[z] = nil[T]
  color[z] = RED
  RB-Insert-fixup(T,z)
```



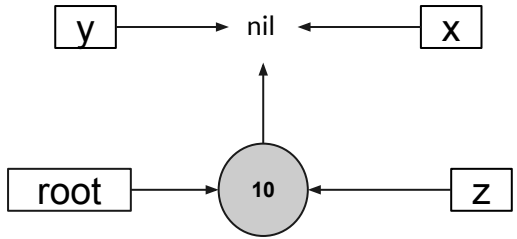
Insert operation pseudocode

```
RB-Insert(T,z)
  y = nil[T]
  x = root[T]
  while x != nil[T]
     y = x
     if key[z] < key[x] then
        x = left[x]
     else
        x = right[x]
  p[z] = y
  if y = nil[T]
     root[T] = z
  else
    if key[z] < key[y] then
      left[y] = z
    else
      right[y] = z
  left[z] = nil[T]
  right[z] = nil[T]
  color[z] = RED
  RB-Insert-fixup(T,z)
```



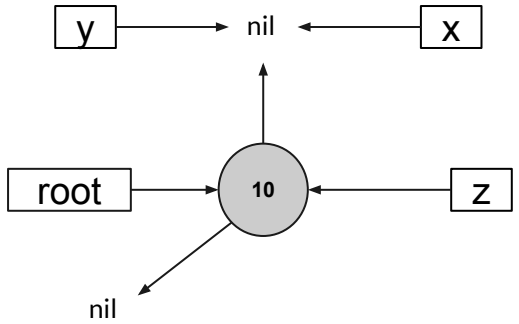
Insert operation pseudocode

```
RB-Insert(T,z)
  y = nil[T]
  x = root[T]
  while x != nil[T]
     y = x
     if key[z] < key[x] then
        x = left[x]
     else
        x = right[x]
  p[z] = y
  if y = nil[T]
     root[T] = z
  else
    if key[z] < key[y] then
      left[y] = z
    else
      right[y] = z
  left[z] = nil[T]
  right[z] = nil[T]
  color[z] = RED
  RB-Insert-fixup(T,z)
```



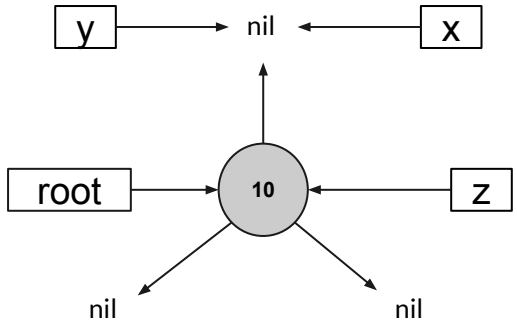
Insert operation pseudocode

```
RB-Insert(T,z)
   y = nil[T]
   x = root[T]
   while x != nil[T]
      y = x
      if key[z] < key[x] then
         x = left[x]
      else
         x = right[x]
   p[z] = y
   if y = nil[T]
      root[T] = z
   else
     if key[z] < key[y] then
       left[y] = z
     else
       right[y] = z
   left[z] = nil[T]
   right[z] = nil[T]
   color[z] = RED
   RB-Insert-fixup(T,z)
```



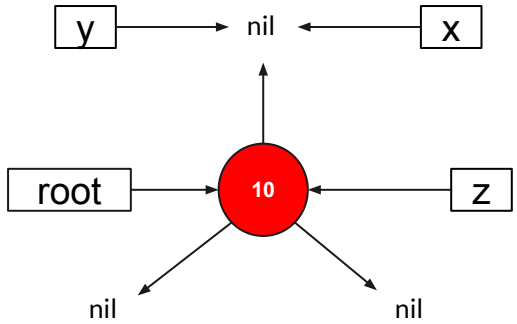
Insert operation pseudocode

```
RB-Insert(T,z)
   y = nil[T]
   x = root[T]
   while x != nil[T]
      y = x
      if key[z] < key[x] then
         x = left[x]
      else
         x = right[x]
   p[z] = y
   if y = nil[T]
      root[T] = z
   else
     if key[z] < key[y] then
       left[y] = z
     else
       right[y] = z
   left[z] = nil[T]
   right[z] = nil[T]
   color[z] = RED
   RB-Insert-fixup(T,z)
```



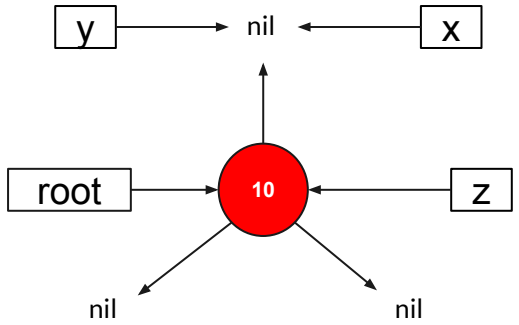
Insert operation pseudocode

```
RB-Insert(T,z)
  y = nil[T]
  x = root[T]
  while x != nil[T]
     y = x
     if key[z] < key[x] then
        x = left[x]
     else
        x = right[x]
  p[z] = y
  if y = nil[T]
     root[T] = z
  else
    if key[z] < key[y] then
      left[y] = z
    else
      right[y] = z
  left[z] = nil[T]
  right[z] = nil[T]
  color[z] = RED
  RB-Insert-fixup(T,z)
```



Insert operation pseudocode

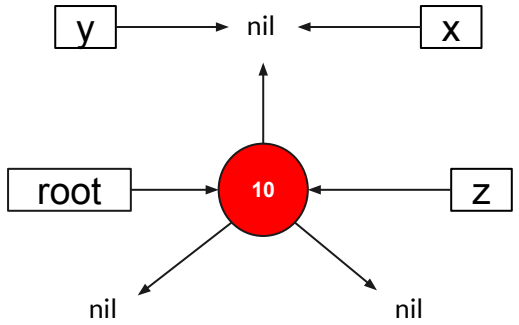
```
RB-Insert(T,z)
  y = nil[T]
  x = root[T]
  while x != nil[T]
     y = x
     if key[z] < key[x] then
        x = left[x]
     else
        x = right[x]
  p[z] = y
  if y = nil[T]
     root[T] = z
  else
    if key[z] < key[y] then
      left[y] = z
    else
      right[y] = z
  left[z] = nil[T]
  right[z] = nil[T]
  color[z] = RED
  RB-Insert-fixup(T,z)
```



RB-Insert-fixup() is the function that restores the RBTree properties after insert. We will talk about it in a bit.

Insert operation pseudocode

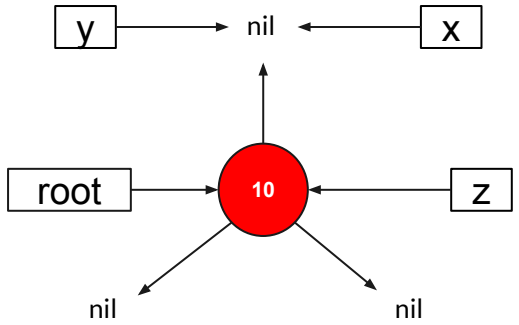
```
RB-Insert(T,z)
  y = nil[T]
  x = root[T]
  while x != nil[T]
     y = x
     if key[z] < key[x] then
        x = left[x]
     else
        x = right[x]
  p[z] = y
  if y = nil[T]
     root[T] = z
  else
    if key[z] < key[y] then
      left[y] = z
    else
      right[y] = z
  left[z] = nil[T]
  right[z] = nil[T]
  color[z] = RED
  RB-Insert-fixup(T,z)
```



Is this a valid RB-tree?

Insert operation pseudocode

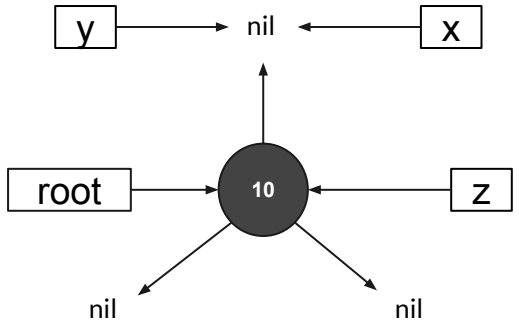
```
RB-Insert(T,z)
  y = nil[T]
  x = root[T]
  while x != nil[T]
     y = x
     if key[z] < key[x] then
        x = left[x]
     else
        x = right[x]
  p[z] = y
  if y = nil[T]
     root[T] = z
  else
    if key[z] < key[y] then
      left[y] = z
    else
      right[y] = z
  left[z] = nil[T]
  right[z] = nil[T]
  color[z] = RED
  RB-Insert-fixup(T,z)
```



Is this a valid RB-tree? Nope (root must be black)

Insert operation pseudocode

```
RB-Insert(T,z)
  y = nil[T]
  x = root[T]
  while x != nil[T]
     y = x
     if key[z] < key[x] then
        x = left[x]
     else
        x = right[x]
  p[z] = y
  if y = nil[T]
     root[T] = z
  else
    if key[z] < key[y] then
      left[y] = z
    else
      right[y] = z
  left[z] = nil[T]
  right[z] = nil[T]
  color[z] = RED
  RB-Insert-fixup(T,z)
```

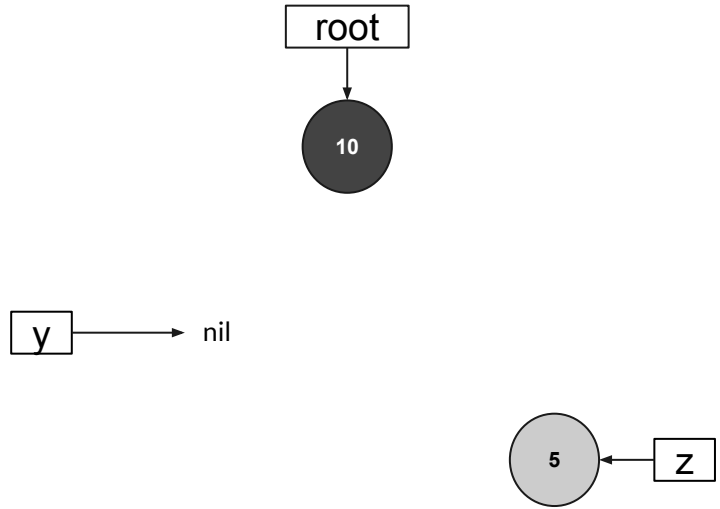


Is this a valid RB-tree? Nope (root must be black)

More interesting example

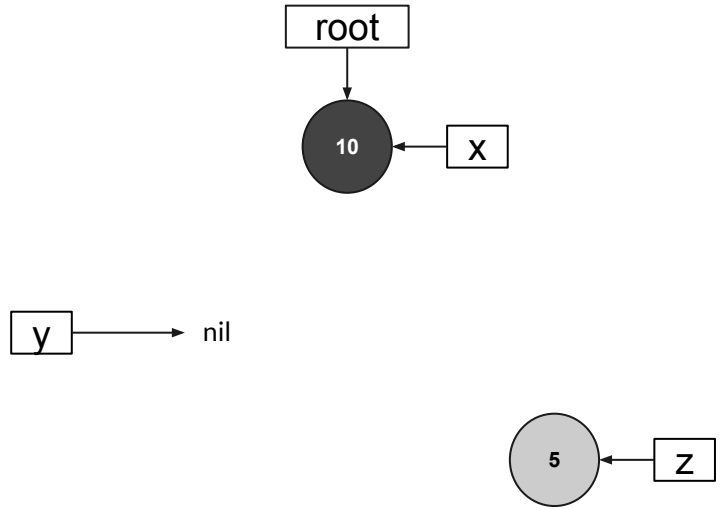
Insert operation pseudocode

```
RB-Insert(T,z)
  y = nil[T]
  x = root[T]
  while x != nil[T]
     y = x
     if key[z] < key[x] then
        x = left[x]
     else
        x = right[x]
  p[z] = y
  if y = nil[T]
     root[T] = z
  else
    if key[z] < key[y] then
      left[y] = z
    else
      right[y] = z
  left[z] = nil[T]
  right[z] = nil[T]
  color[z] = RED
  RB-Insert-fixup(T,z)
```



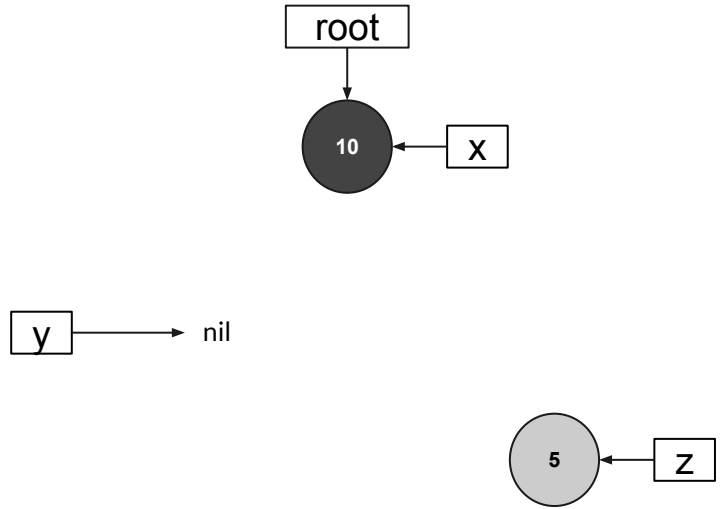
Insert operation pseudocode

```
RB-Insert(T,z)
   y = nil[T]
   x = root[T]
   while x != nil[T]
      y = x
      if key[z] < key[x] then
         x = left[x]
      else
         x = right[x]
   p[z] = y
   if y = nil[T]
      root[T] = z
   else
     if key[z] < key[y] then
       left[y] = z
     else
       right[y] = z
   left[z] = nil[T]
   right[z] = nil[T]
   color[z] = RED
   RB-Insert-fixup(T,z)
```



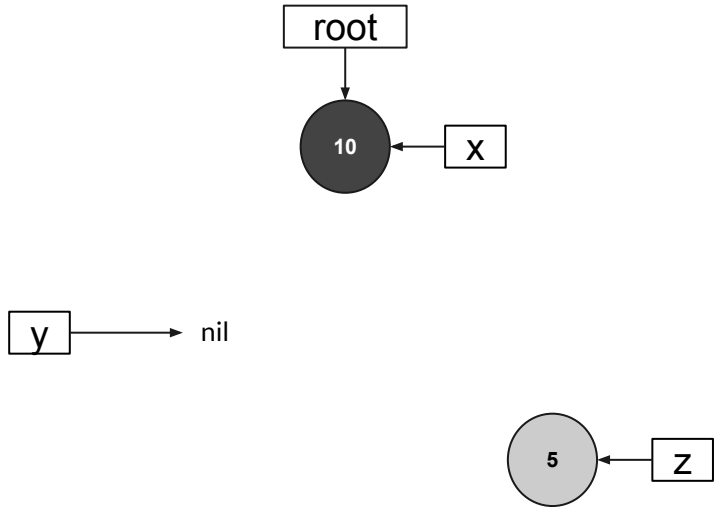
Insert operation pseudocode

```
RB-Insert(T,z)
  y = nil[T]
  x = root[T]
  while x != nil[T]
     y = x
     if key[z] < key[x] then
        x = left[x]
     else
        x = right[x]
  p[z] = y
  if y = nil[T]
     root[T] = z
  else
    if key[z] < key[y] then
      left[y] = z
    else
      right[y] = z
  left[z] = nil[T]
  right[z] = nil[T]
  color[z] = RED
  RB-Insert-fixup(T,z)
```



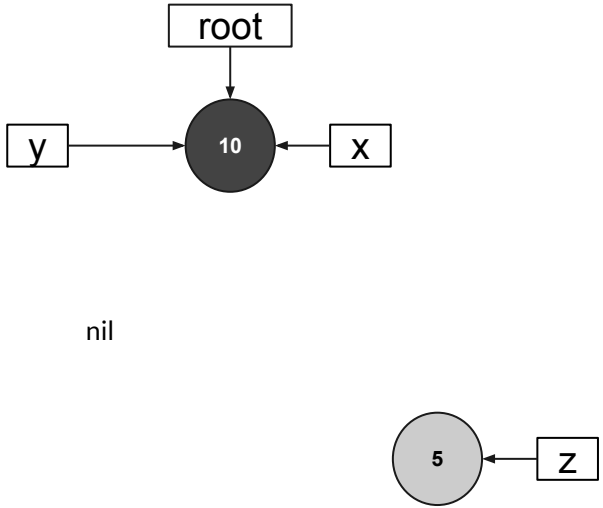
Insert operation pseudocode

```
RB-Insert(T,z)
  y = nil[T]
  x = root[T]
  while x != nil[T]
     y = x
     if key[z] < key[x] then
        x = left[x]
     else
        x = right[x]
  p[z] = y
  if y = nil[T]
     root[T] = z
  else
    if key[z] < key[y] then
      left[y] = z
    else
      right[y] = z
  left[z] = nil[T]
  right[z] = nil[T]
  color[z] = RED
  RB-Insert-fixup(T,z)
```



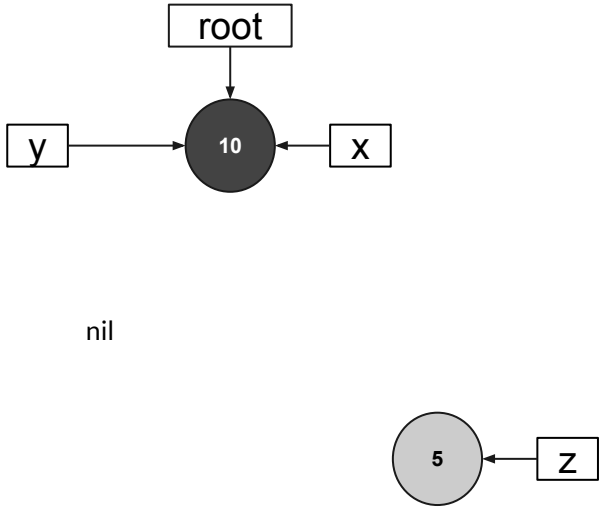
Insert operation pseudocode

```
RB-Insert(T,z)
  y = nil[T]
  x = root[T]
  while x != nil[T]
     y = x
     if key[z] < key[x] then
        x = left[x]
     else
        x = right[x]
  p[z] = y
  if y = nil[T]
     root[T] = z
  else
    if key[z] < key[y] then
      left[y] = z
    else
      right[y] = z
  left[z] = nil[T]
  right[z] = nil[T]
  color[z] = RED
  RB-Insert-fixup(T,z)
```



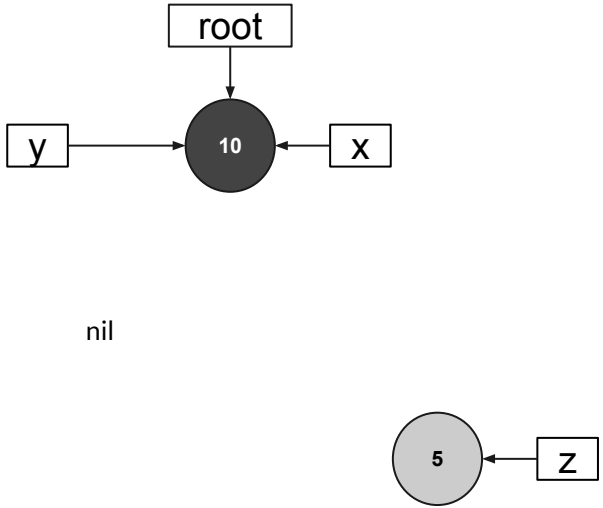
Insert operation pseudocode

```
RB-Insert(T,z)
  y = nil[T]
  x = root[T]
  while x != nil[T]
      y = x
      if key[z] < key[x] then
         x = left[x]
      else
         x = right[x]
  p[z] = y
  if y = nil[T]
     root[T] = z
  else
    if key[z] < key[y] then
       left[y] = z
    else
       right[y] = z
  left[z] = nil[T]
  right[z] = nil[T]
  color[z] = RED
  RB-Insert-fixup(T,z)
```



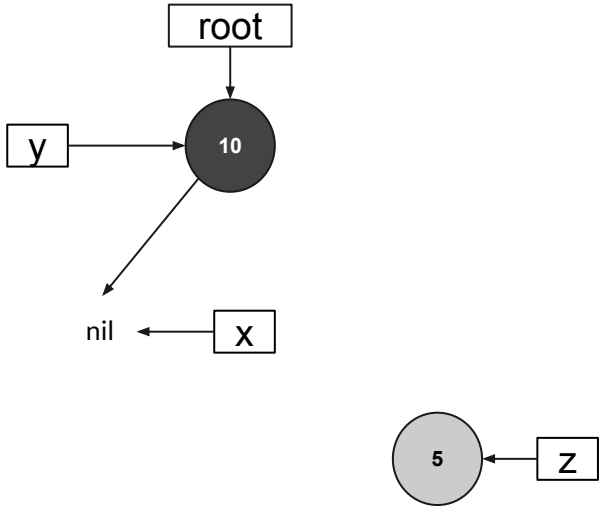
Insert operation pseudocode

```
RB-Insert(T,z)
  y = nil[T]
  x = root[T]
  while x != nil[T]
     y = x
     if key[z] < key[x] then
        x = left[x]
     else
        x = right[x]
  p[z] = y
  if y = nil[T]
     root[T] = z
  else
    if key[z] < key[y] then
      left[y] = z
    else
      right[y] = z
  left[z] = nil[T]
  right[z] = nil[T]
  color[z] = RED
  RB-Insert-fixup(T,z)
```



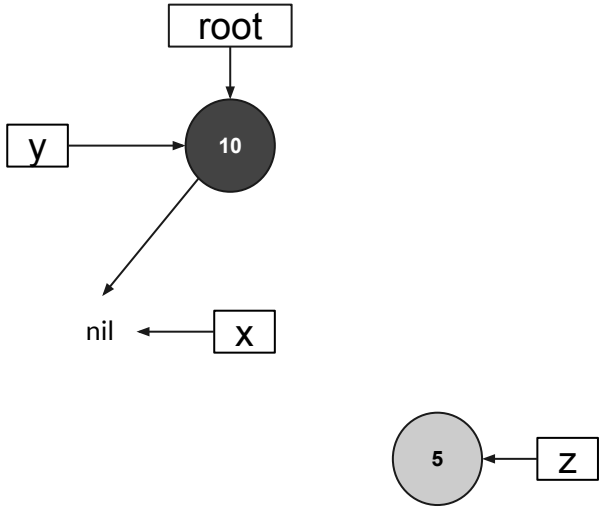
Insert operation pseudocode

```
RB-Insert(T,z)
  y = nil[T]
  x = root[T]
  while x != nil[T]
      y = x
      if key[z] < key[x] then
         x = left[x]
      else
         x = right[x]
  p[z] = y
  if y = nil[T]
     root[T] = z
  else
    if key[z] < key[y] then
       left[y] = z
    else
       right[y] = z
  left[z] = nil[T]
  right[z] = nil[T]
  color[z] = RED
  RB-Insert-fixup(T,z)
```



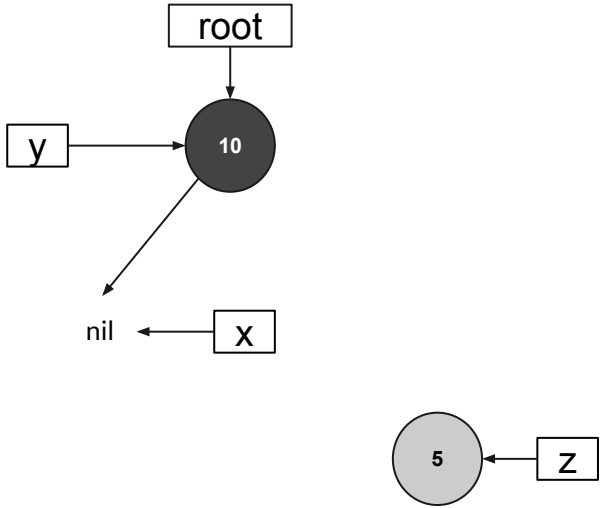
Insert operation pseudocode

```
RB-Insert(T,z)
  y = nil[T]
  x = root[T]
  while x != nil[T]
     y = x
     if key[z] < key[x] then
        x = left[x]
     else
        x = right[x]
  p[z] = y
  if y = nil[T]
     root[T] = z
  else
    if key[z] < key[y] then
      left[y] = z
    else
      right[y] = z
  left[z] = nil[T]
  right[z] = nil[T]
  color[z] = RED
  RB-Insert-fixup(T,z)
```



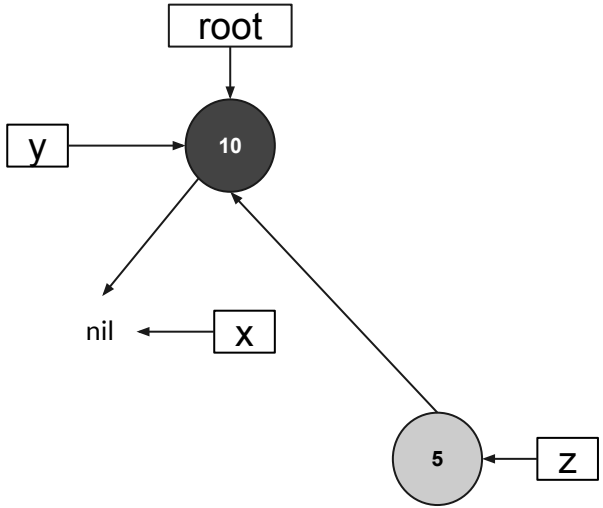
Insert operation pseudocode

```
RB-Insert(T,z)
  y = nil[T]
  x = root[T]
  while x != nil[T]
     y = x
     if key[z] < key[x] then
        x = left[x]
     else
        x = right[x]
  p[z] = y
  if y = nil[T]
     root[T] = z
  else
    if key[z] < key[y] then
      left[y] = z
    else
      right[y] = z
  left[z] = nil[T]
  right[z] = nil[T]
  color[z] = RED
  RB-Insert-fixup(T,z)
```



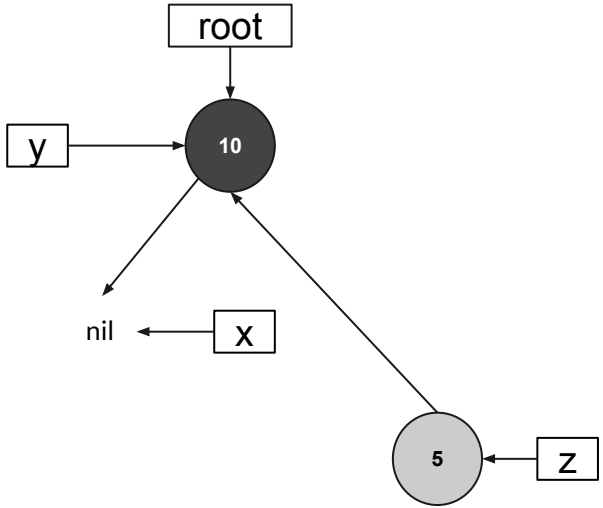
Insert operation pseudocode

```
RB-Insert(T,z)
   y = nil[T]
   x = root[T]
   while x != nil[T]
      y = x
      if key[z] < key[x] then
         x = left[x]
      else
         x = right[x]
   p[z] = y
   if y = nil[T]
      root[T] = z
   else
     if key[z] < key[y] then
       left[y] = z
     else
       right[y] = z
   left[z] = nil[T]
   right[z] = nil[T]
   color[z] = RED
   RB-Insert-fixup(T,z)
```



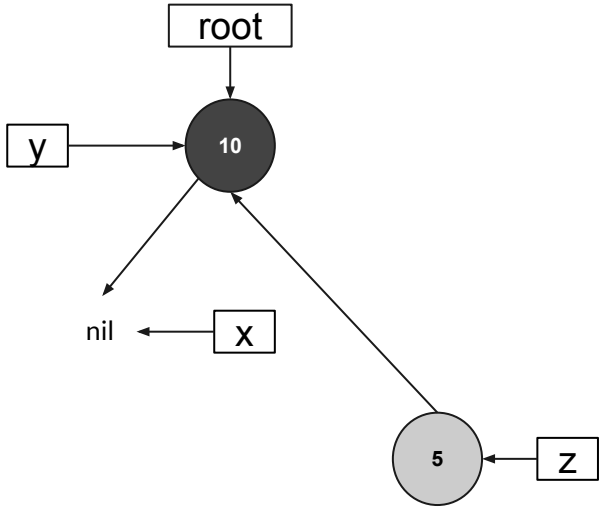
Insert operation pseudocode

```
RB-Insert(T,z)
  y = nil[T]
  x = root[T]
  while x != nil[T]
     y = x
     if key[z] < key[x] then
        x = left[x]
     else
        x = right[x]
  p[z] = y
  if y = nil[T]
     root[T] = z
  else
    if key[z] < key[y] then
      left[y] = z
    else
      right[y] = z
  left[z] = nil[T]
  right[z] = nil[T]
  color[z] = RED
  RB-Insert-fixup(T,z)
```



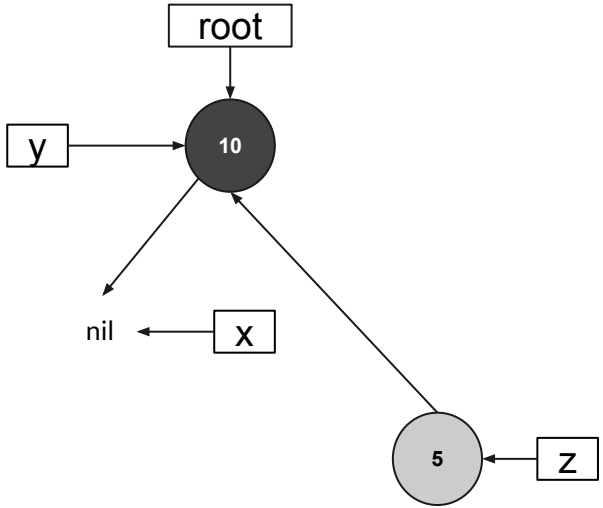
Insert operation pseudocode

```
RB-Insert(T,z)
  y = nil[T]
  x = root[T]
  while x != nil[T]
     y = x
     if key[z] < key[x] then
        x = left[x]
     else
        x = right[x]
  p[z] = y
  if y = nil[T]
     root[T] = z
  else
    if key[z] < key[y] then
      left[y] = z
    else
      right[y] = z
  left[z] = nil[T]
  right[z] = nil[T]
  color[z] = RED
  RB-Insert-fixup(T,z)
```



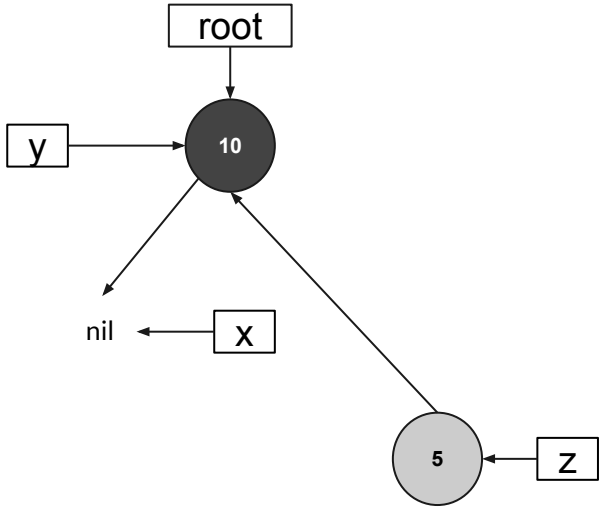
Insert operation pseudocode

```
RB-Insert(T,z)
  y = nil[T]
  x = root[T]
  while x != nil[T]
     y = x
     if key[z] < key[x] then
        x = left[x]
     else
        x = right[x]
  p[z] = y
  if y = nil[T]
     root[T] = z
  else
    if key[z] < key[y] then
      left[y] = z
    else
      right[y] = z
  left[z] = nil[T]
  right[z] = nil[T]
  color[z] = RED
  RB-Insert-fixup(T,z)
```



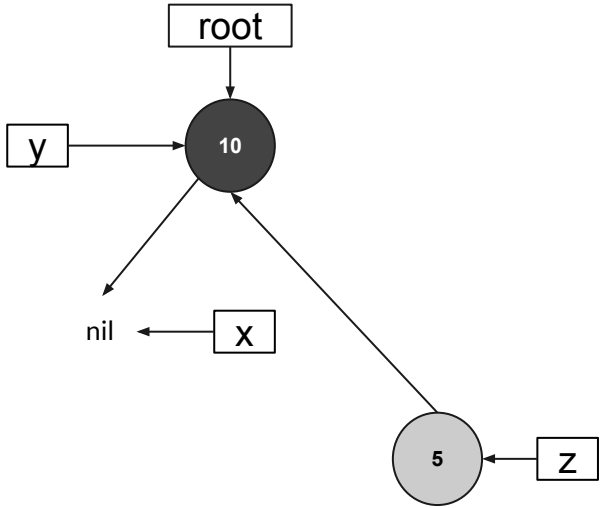
Insert operation pseudocode

```
RB-Insert(T,z)
  y = nil[T]
  x = root[T]
  while x != nil[T]
     y = x
     if key[z] < key[x] then
        x = left[x]
     else
        x = right[x]
  p[z] = y
  if y = nil[T]
     root[T] = z
  else
    if key[z] < key[y] then
      left[y] = z
    else
      right[y] = z
  left[z] = nil[T]
  right[z] = nil[T]
  color[z] = RED
  RB-Insert-fixup(T,z)
```



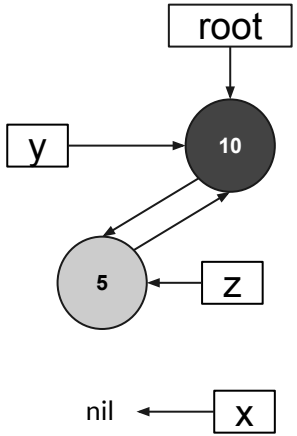
Insert operation pseudocode

```
RB-Insert(T,z)
  y = nil[T]
  x = root[T]
  while x != nil[T]
     y = x
     if key[z] < key[x] then
        x = left[x]
     else
        x = right[x]
  p[z] = y
  if y = nil[T]
     root[T] = z
  else
    if key[z] < key[y] then
      left[y] = z
    else
      right[y] = z
  left[z] = nil[T]
  right[z] = nil[T]
  color[z] = RED
  RB-Insert-fixup(T,z)
```



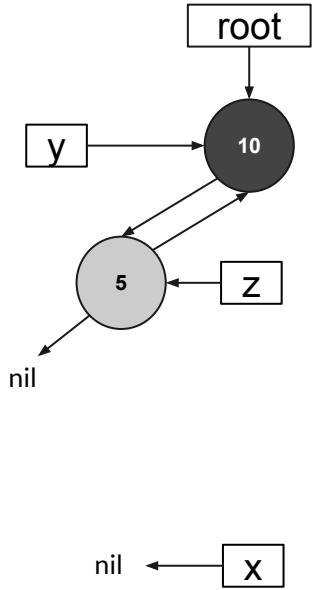
Insert operation pseudocode

```
RB-Insert(T,z)
  y = nil[T]
  x = root[T]
  while x != nil[T]
     y = x
     if key[z] < key[x] then
        x = left[x]
     else
        x = right[x]
  p[z] = y
  if y = nil[T]
     root[T] = z
  else
    if key[z] < key[y] then
      left[y] = z
    else
      right[y] = z
  left[z] = nil[T]
  right[z] = nil[T]
  color[z] = RED
  RB-Insert-fixup(T,z)
```



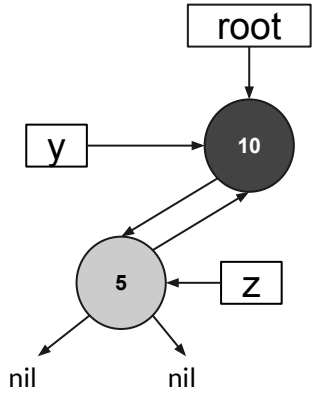
Insert operation pseudocode

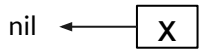
```
RB-Insert(T,z)
   y = nil[T]
   x = root[T]
   while x != nil[T]
      y = x
      if key[z] < key[x] then
         x = left[x]
      else
         x = right[x]
   p[z] = y
   if y = nil[T]
      root[T] = z
   else
     if key[z] < key[y] then
       left[y] = z
     else
       right[y] = z
   left[z] = nil[T]
   right[z] = nil[T]
   color[z] = RED
   RB-Insert-fixup(T,z)
```



Insert operation pseudocode

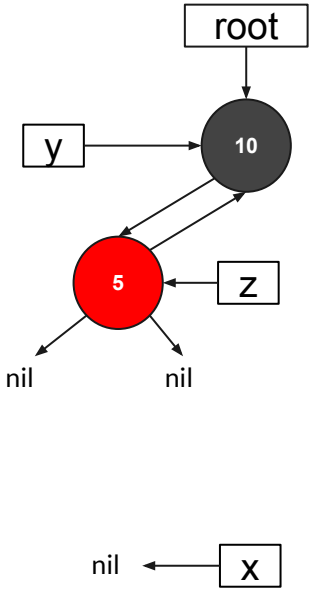
```
RB-Insert(T,z)
   y = nil[T]
   x = root[T]
   while x != nil[T]
      y = x
      if key[z] < key[x] then
         x = left[x]
      else
         x = right[x]
   p[z] = y
   if y = nil[T]
      root[T] = z
   else
     if key[z] < key[y] then
       left[y] = z
     else
       right[y] = z
   left[z] = nil[T]
   right[z] = nil[T]
   color[z] = RED
   RB-Insert-fixup(T,z)
```





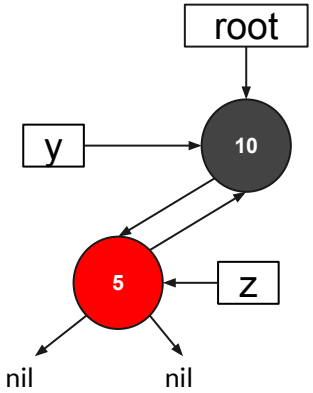
Insert operation pseudocode

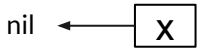
```
RB-Insert(T,z)
  y = nil[T]
  x = root[T]
  while x != nil[T]
     y = x
     if key[z] < key[x] then
        x = left[x]
     else
        x = right[x]
  p[z] = y
  if y = nil[T]
     root[T] = z
  else
    if key[z] < key[y] then
      left[y] = z
    else
      right[y] = z
  left[z] = nil[T]
  right[z] = nil[T]
  color[z] = RED
  RB-Insert-fixup(T,z)
```



Insert operation pseudocode

```
RB-Insert(T,z)
  y = nil[T]
  x = root[T]
  while x != nil[T]
     y = x
     if key[z] < key[x] then
        x = left[x]
     else
        x = right[x]
  p[z] = y
  if y = nil[T]
     root[T] = z
  else
    if key[z] < key[y] then
      left[y] = z
    else
      right[y] = z
  left[z] = nil[T]
  right[z] = nil[T]
  color[z] = RED
  RB-Insert-fixup(T,z)
```



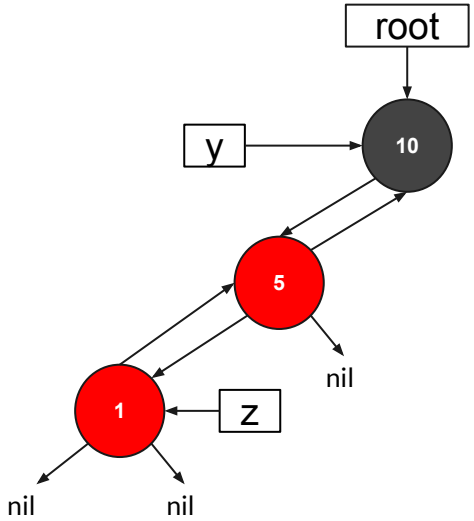


Nothing to fixup, this is a valid RB-tree

Let’s complicate this a bit

Insert operation pseudocode

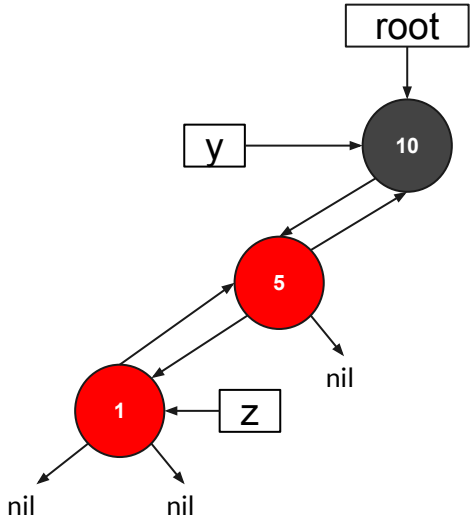
```
RB-Insert(T,z)
  y = nil[T]
  x = root[T]
  while x != nil[T]
     y = x
     if key[z] < key[x] then
        x = left[x]
     else
        x = right[x]
  p[z] = y
  if y = nil[T]
     root[T] = z
  else
    if key[z] < key[y] then
      left[y] = z
    else
      right[y] = z
  left[z] = nil[T]
  right[z] = nil[T]
  color[z] = RED
  RB-Insert-fixup(T,z)
```



Let's say we added the 1, is this a valid RB-tree?

Insert operation pseudocode

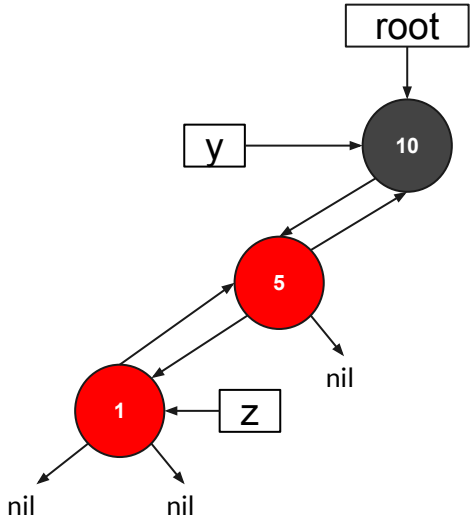
```
RB-Insert(T,z)
  y = nil[T]
  x = root[T]
  while x != nil[T]
     y = x
     if key[z] < key[x] then
        x = left[x]
     else
        x = right[x]
  p[z] = y
  if y = nil[T]
     root[T] = z
  else
    if key[z] < key[y] then
      left[y] = z
    else
      right[y] = z
  left[z] = nil[T]
  right[z] = nil[T]
  color[z] = RED
  RB-Insert-fixup(T,z)
```



Let's say we added the 1, is this a valid RB-tree? Nope.

Insert operation pseudocode

```
RB-Insert(T,z)
  y = nil[T]
  x = root[T]
  while x != nil[T]
     y = x
     if key[z] < key[x] then
        x = left[x]
     else
        x = right[x]
  p[z] = y
  if y = nil[T]
     root[T] = z
  else
    if key[z] < key[y] then
      left[y] = z
    else
      right[y] = z
  left[z] = nil[T]
  right[z] = nil[T]
  color[z] = RED
  RB-Insert-fixup(T,z)
```



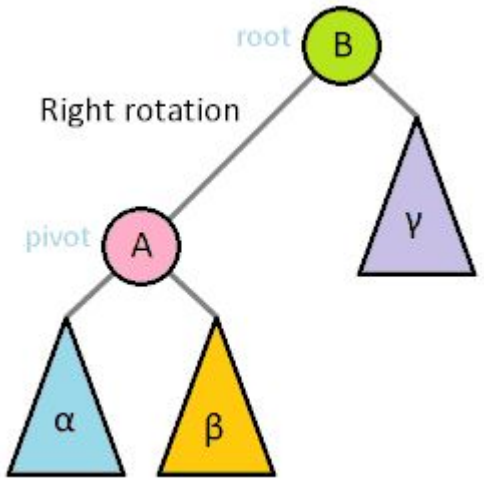
Let's say we added the 1, is this a valid RB-tree? Nope, we need to fix this tree.

Insert-fixup

Insert-fixup

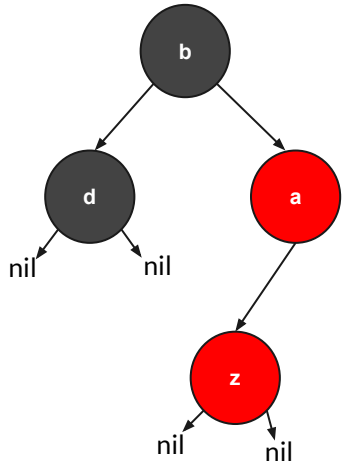
We can restore the properties of a red-black tree by:

- Recoloring
- Rotation



Fixup

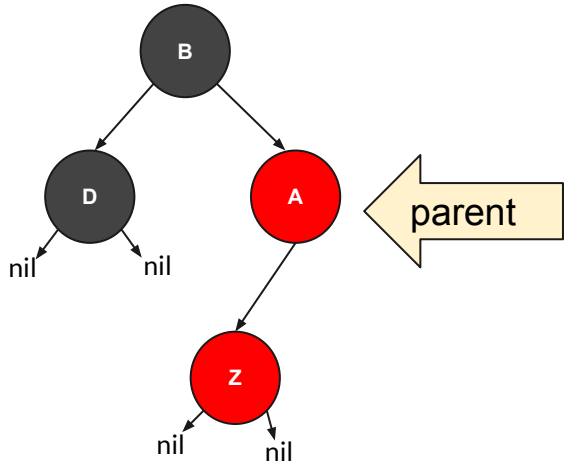
We have 4 scenarios, to learn them we must agree on some terminology:



Fixup

We have 4 scenarios, to learn them we must agree on some terminology. Z's relations are:

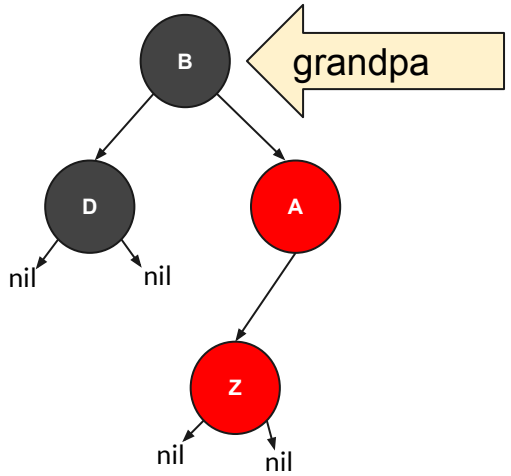
A: Parent
B: Grandparent
D: uncle



Fixup

We have 4 scenarios, to learn them we must agree on some terminology. Z's relations are:

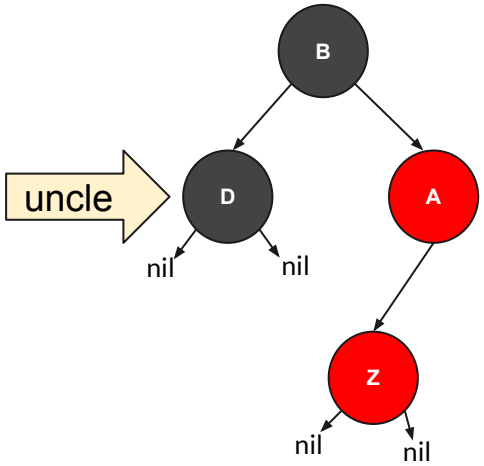
A: Parent
B: Grandparent
D: uncle



Fixup

We have 4 scenarios, to learn them we must agree on some terminology. Z's relations are:

A: Parent
B: Grandparent
D: uncle



Fixup scenarios

Case 0: Z is the root

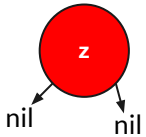
Case 1: Z's uncle is red

Case 2: Z's uncle is black (triangle)

Case 3: Z's uncle is black (line)

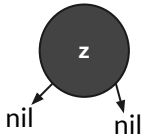
Case 0: Z is the root

1. Color Z black
2. Done!



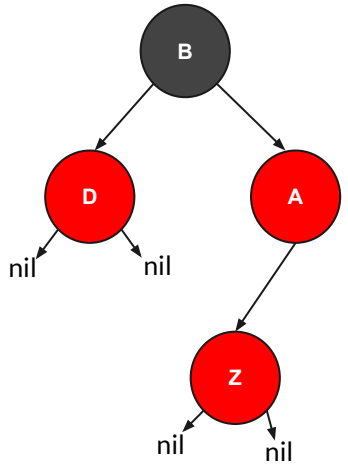
Case 0: Z is the root

1. Color Z black
2. Done!



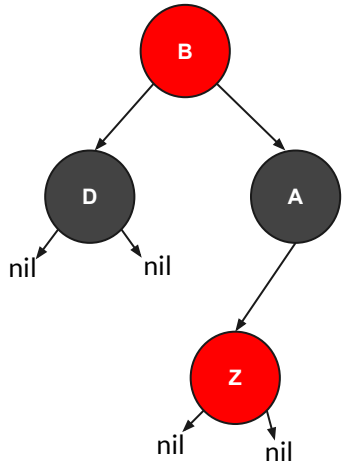
Case 1: Z's uncle is red

1. Recolor parent, uncle, and grandparent.
2. Done!



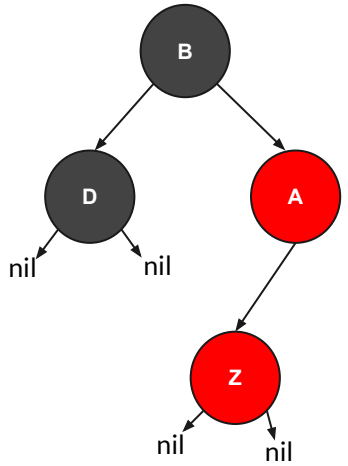
Case 1: Z's uncle is red

1. Recolor parent, uncle, and grandparent.
2. Done!



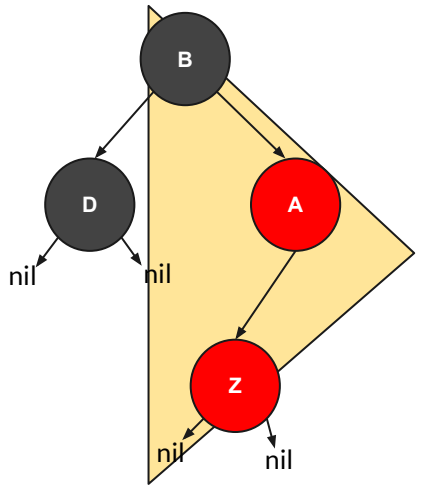
Case 2: Z's uncle is black (triangle)

Triangle: Z, its parent, and grandpa form a triangle



Case 2: Z's uncle is black (triangle)

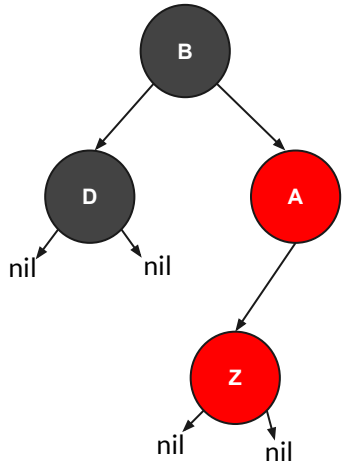
Triangle: Z, its parent, and grandpa form a triangle



Case 2: Z's uncle is black (triangle)

Triangle: Z, its parent, and grandpa form a triangle.

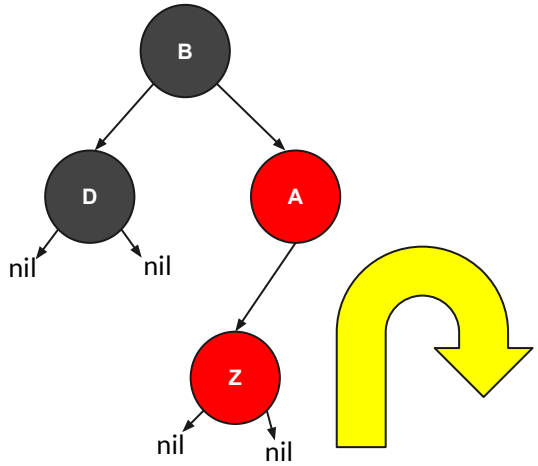
1. Rotate Z's parent in the opposite direction to Z.



Case 2: Z's uncle is black (triangle)

Triangle: Z, its parent, and grandpa form a triangle.

1. Rotate Z's parent in the opposite direction to Z.

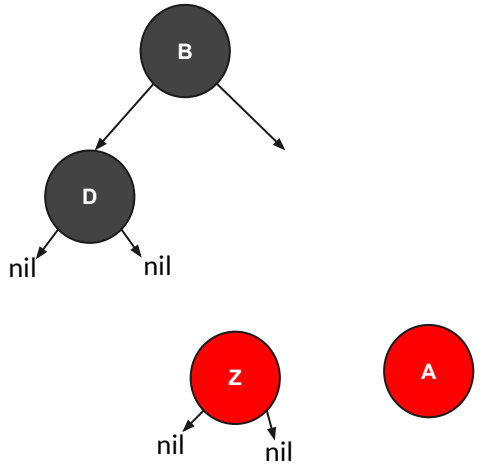


In this example, Z is on the left, so we rotate A to the right.

Case 2: Z's uncle is black (triangle)

Triangle: Z, its parent, and grandpa form a triangle.

1. Rotate Z's parent in the opposite direction to Z.

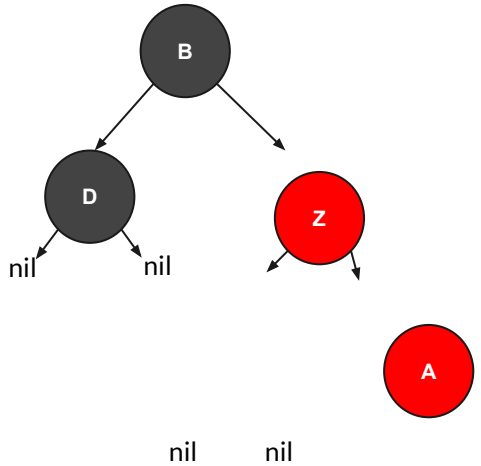


In this example, Z is on the left, so we rotate A to the right.

Case 2: Z's uncle is black (triangle)

Triangle: Z, its parent, and grandpa form a triangle.

1. Rotate Z's parent in the opposite direction to Z.

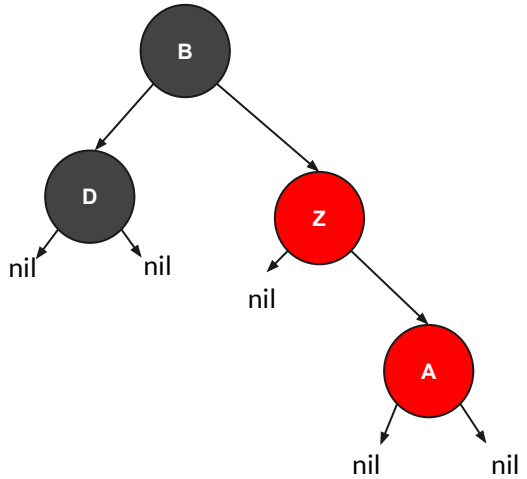


In this example, Z is on the left, so we rotate A to the right.

Case 2: Z's uncle is black (triangle)

Triangle: Z, its parent, and grandpa form a triangle.

1. Rotate Z's parent in the opposite direction to Z.

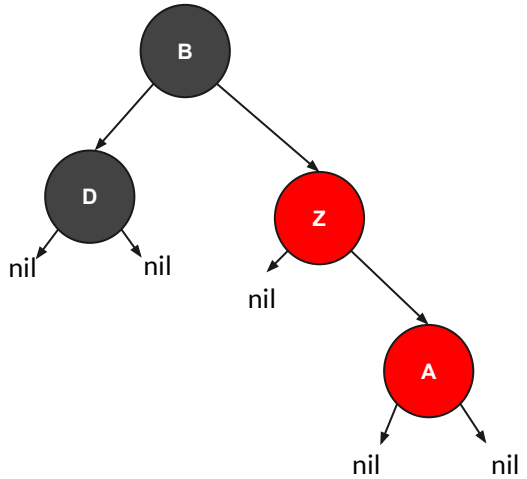


In this example, Z is on the left, so we rotate A to the right.

Case 2: Z's uncle is black (triangle)

Triangle: Z, its parent, and grandpa form a triangle.

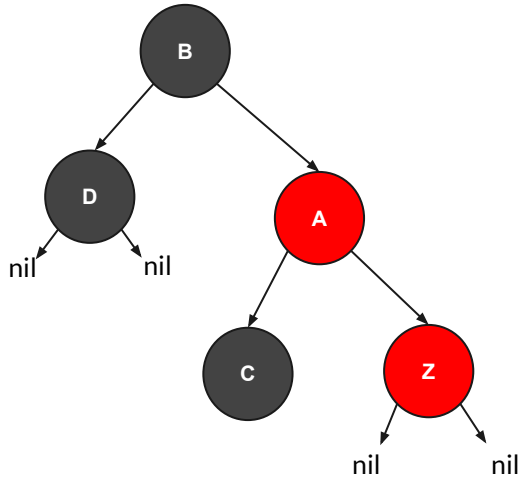
1. Rotate Z's parent in the opposite direction to Z.



I realize that this oversimplified example does not seem to solve the problem yet. We will see this in a realistic example in a bit.

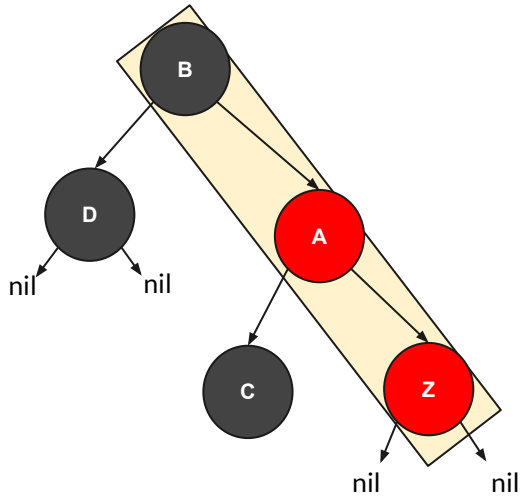
Case 3: Z's uncle is black (line)

Line: Z, its parent, and grandpa form a line.



Case 3: Z's uncle is black (line)

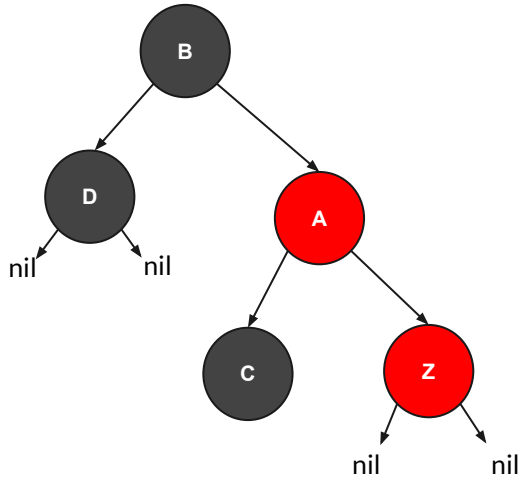
Line: Z, its parent, and grandpa form a line.



Case 3: Z's uncle is black (line)

Line: Z, its parent, and grandpa form a line.

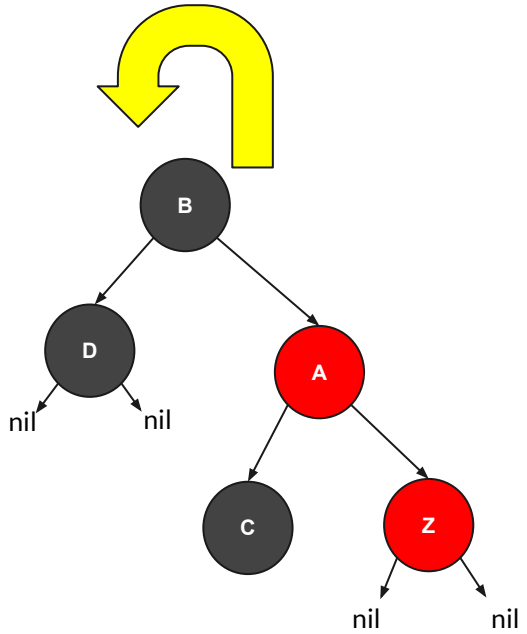
1. Rotate Z's **grandparent** in the opposite direction to Z.



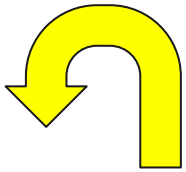
Case 3: Z's uncle is black (line)

Line: Z, its parent, and grandpa form a line.

1. Rotate Z's **grandparent** in the opposite direction to Z.

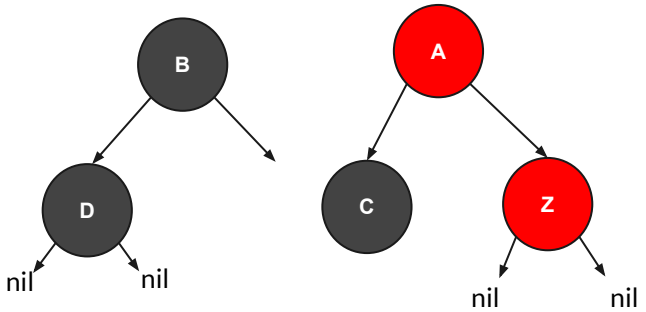


Case 3: Z's uncle is black (line)



Line: Z, its parent, and grandpa form a line.

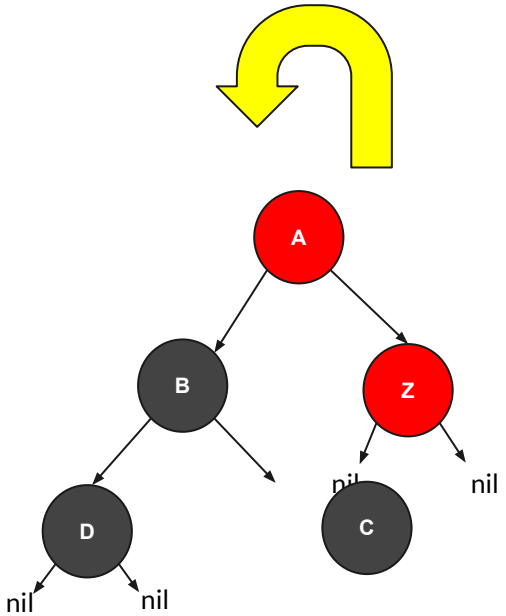
1. Rotate Z's **grandparent** in the opposite direction to Z.



Case 3: Z's uncle is black (line)

Line: Z, its parent, and grandpa form a line.

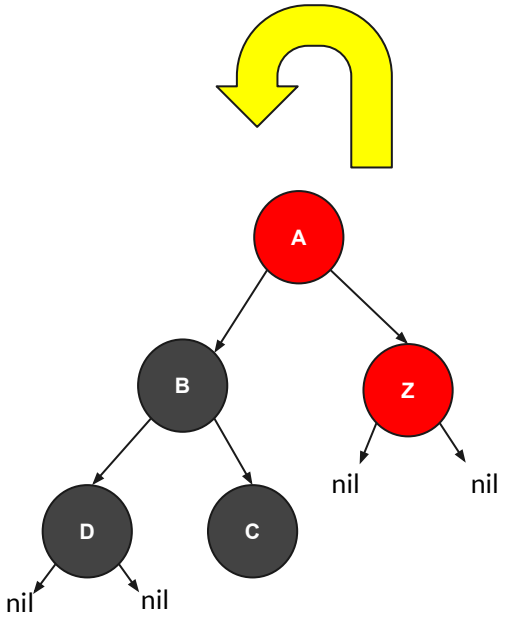
1. Rotate Z's **grandparent** in the opposite direction to Z.



Case 3: Z's uncle is black (line)

Line: Z, its parent, and grandpa form a line.

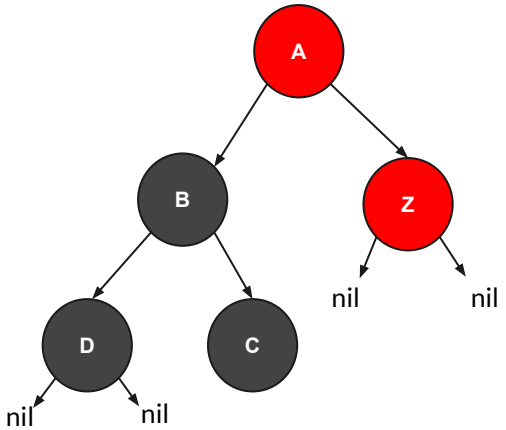
1. Rotate Z's **grandparent** in the opposite direction to Z.



Case 3: Z's uncle is black (line)

Line: Z, its parent, and grandpa form a line.

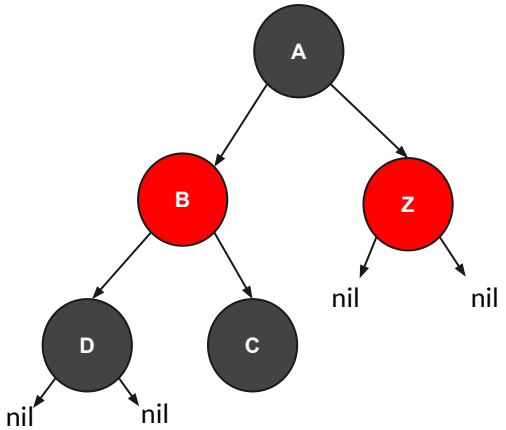
1. Rotate Z's **grandparent** in the opposite direction to Z.
2. Recolor parent and grandparent.



Case 3: Z's uncle is black (line)

Line: Z, its parent, and grandpa form a line.

1. Rotate Z's **grandparent** in the opposite direction to Z.
2. Recolor parent and grandparent.



Fixup scenarios summary

Case 0: Z is the root => color Z black

Case 1: Z's uncle is red => recolor parent, grandparent, and uncle

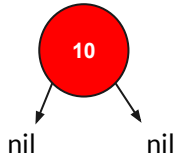
Case 2: Z's uncle is black (triangle) => rotate parent

Case 3: Z's uncle is black (line) => rotate grandparent + recolor parent and grandparent

Fixup pseudocode: case 0

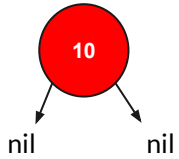
Fixup pseudocode: Case 0

```
RB-Insert-fixup(T,z) {
 while(z's parent is Red) {
    set y to be z's uncle
    if uncle y is Red {
          color parent and uncle black
          color grandparent red
          set z to grandparent
    }
    else {  // the uncle is black
         if (triangle) {
                   set z to parent
                   rotate to parent
                  }
         // rotate the grandparent and finish
         color parent of z black
         color grandparent of z red
         rotate grand parent of z
       }
  } // end while
 color root black
}
```



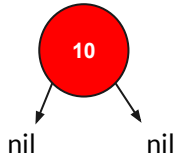
Fixup pseudocode: Case 0

```
RB-Insert-fixup(T,z) {
 while(z's parent is Red) {
    set y to be z's uncle
    if uncle y is Red {
          color parent and uncle black
          color grandparent red
          set z to grandparent
    }
    else {  // the uncle is black
         if (triangle) {
                   set z to parent
                   rotate to parent
                  }
         // rotate the grandparent and finish
         color parent of z black
         color grandparent of z red
         rotate grand parent of z
       }
  } // end while
 color root black
}
```



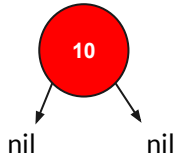
Fixup pseudocode: Case 0

```
RB-Insert-fixup(T,z) {
 while(z's parent is Red) {
    set y to be z's uncle
    if uncle y is Red {
          color parent and uncle black
          color grandparent red
          set z to grandparent
    }
    else {  // the uncle is black
         if (triangle) {
                   set z to parent
                   rotate to parent
                  }
         // rotate the grandparent and finish
         color parent of z black
         color grandparent of z red
         rotate grand parent of z
       }
  } // end while
 color root black
}
```



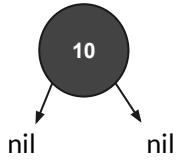
Fixup pseudocode: Case 0

```
RB-Insert-fixup(T,z) {
 while(z's parent is Red) {
    set y to be z's uncle
    if uncle y is Red {
          color parent and uncle black
          color grandparent red
          set z to grandparent
    }
    else {  // the uncle is black
         if (triangle) {
                  set z to parent
                  rotate to parent
                 }
         // rotate the grandparent and finish
         color parent of z black
         color grandparent of z red
         rotate grand parent of z
        }
  } // end while
 color root black
}
```



Fixup pseudocode: Case 0

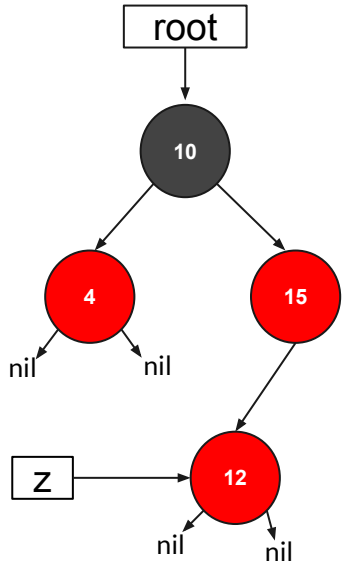
```
RB-Insert-fixup(T,z) {
 while(z's parent is Red) {
    set y to be z's uncle
    if uncle y is Red {
          color parent and uncle black
          color grandparent red
          set z to grandparent
    }
    else {  // the uncle is black
         if (triangle) {
                  set z to parent
                  rotate to parent
                 }
         // rotate the grandparent and finish
         color parent of z black
         color grandparent of z red
         rotate grand parent of z
        }
  } // end while
 color root black
}
```



Fixup pseudocode: case 1

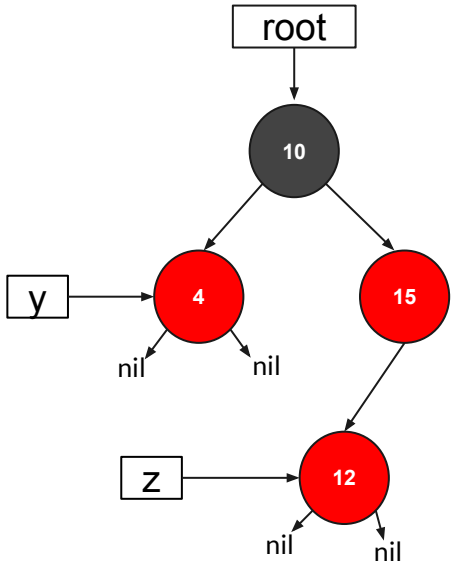
Fixup pseudocode: Case 1

```
RB-Insert-fixup(T,z) {
 while(z's parent is Red) {
    set y to be z's uncle
    if uncle y is Red {
          color parent and uncle black
          color grandparent red
          set z to grandparent
    }
    else {  // the uncle is black
         if (triangle) {
                   set z to parent
                   rotate to parent
                  }
         // rotate the grandparent and finish
         color parent of z black
         color grandparent of z red
         rotate grand parent of z
        }
  } // end while
 color root black
}
```



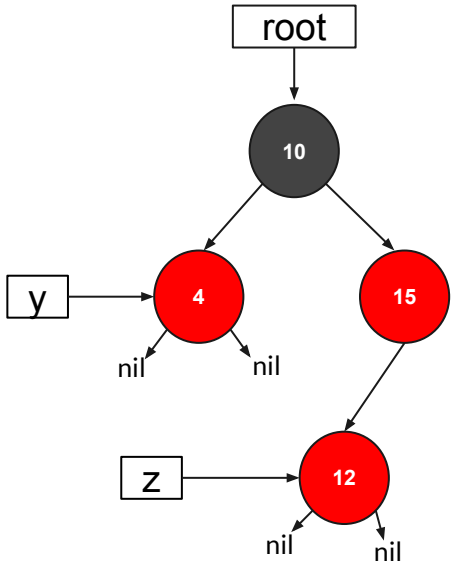
Fixup pseudocode: Case 1

```
RB-Insert-fixup(T,z) {
 while(z's parent is Red) {
    set y to be z's uncle
    if uncle y is Red {
          color parent and uncle black
          color grandparent red
          set z to grandparent
    }
    else {  // the uncle is black
         if (triangle) {
                   set z to parent
                   rotate to parent
                  }
         // rotate the grandparent and finish
         color parent of z black
         color grandparent of z red
         rotate grand parent of z
        }
  } // end while
 color root black
}
```



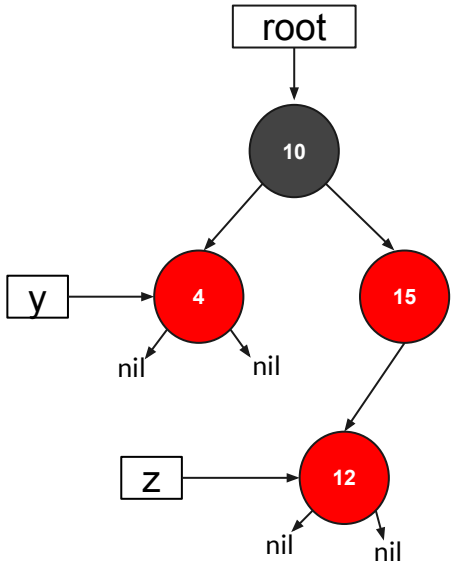
Fixup pseudocode: Case 1

```
RB-Insert-fixup(T,z) {
 while(z's parent is Red) {
    set y to be z's uncle
    if uncle y is Red {
          color parent and uncle black
          color grandparent red
          set z to grandparent
    }
    else {  // the uncle is black
         if (triangle) {
                  set z to parent
                  rotate to parent
                }
         // rotate the grandparent and finish
         color parent of z black
         color grandparent of z red
         rotate grand parent of z
       }
  } // end while
 color root black
}
```



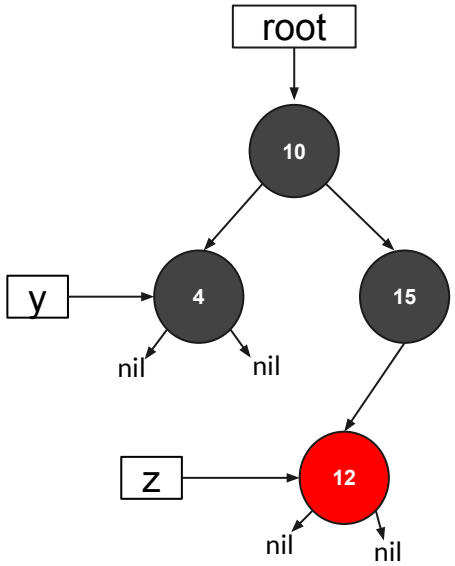
Fixup pseudocode: Case 1

```
RB-Insert-fixup(T,z) {
 while(z's parent is Red) {
    set y to be z's uncle
    if uncle y is Red {
          color parent and uncle black
          color grandparent red
          set z to grandparent
    }
    else {  // the uncle is black
         if (triangle) {
                  set z to parent
                  rotate to parent
                }
         // rotate the grandparent and finish
         color parent of z black
         color grandparent of z red
         rotate grand parent of z
        }
  } // end while
 color root black
}
```



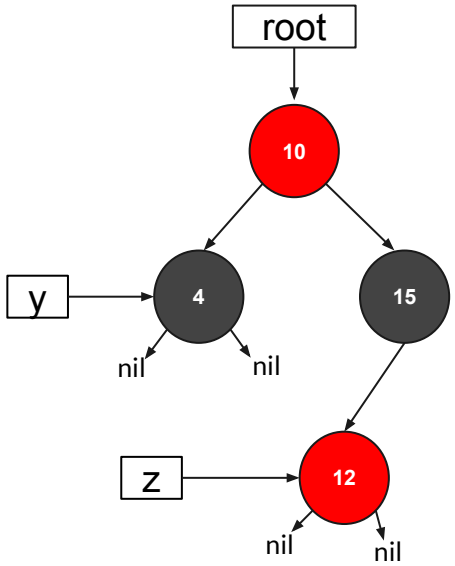
Fixup pseudocode: Case 1

```
RB-Insert-fixup(T,z) {
 while(z's parent is Red) {
    set y to be z's uncle
    if uncle y is Red {
          color parent and uncle black
          color grandparent red
          set z to grandparent
    }
    else {  // the uncle is black
          if (triangle) {
                    set z to parent
                    rotate to parent
                  }
          // rotate the grandparent and finish
          color parent of z black
          color grandparent of z red
          rotate grand parent of z
        }
  } // end while
 color root black
}
```



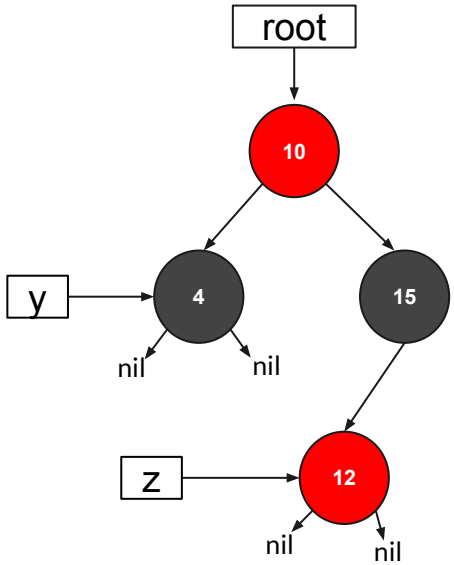
Fixup pseudocode: Case 1

```
RB-Insert-fixup(T,z) {
 while(z's parent is Red) {
    set y to be z's uncle
    if uncle y is Red {
          color parent and uncle black
          color grandparent red
          set z to grandparent
    }
    else {  // the uncle is black
         if (triangle) {
                  set z to parent
                  rotate to parent
                }
         // rotate the grandparent and finish
         color parent of z black
         color grandparent of z red
         rotate grand parent of z
        }
  } // end while
 color root black
}
```



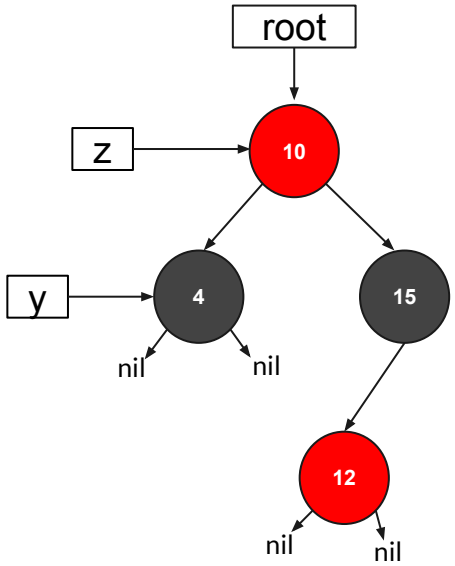
Fixup pseudocode: Case 1

```
RB-Insert-fixup(T,z) {
 while(z's parent is Red) {
    set y to be z's uncle
    if uncle y is Red {
          color parent and uncle black
          color grandparent red
          set z to grandparent
    }
    else {  // the uncle is black
         if (triangle) {
                  set z to parent
                  rotate to parent
                }
         // rotate the grandparent and finish
         color parent of z black
         color grandparent of z red
         rotate grand parent of z
        }
  } // end while
 color root black
}
```



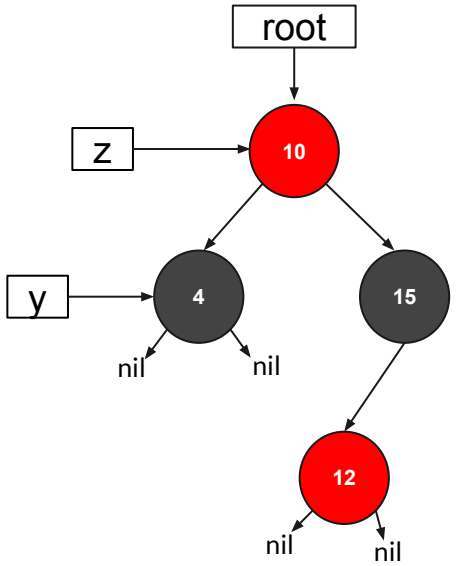
Fixup pseudocode: Case 1

```
RB-Insert-fixup(T,z) {
 while(z's parent is Red) {
    set y to be z's uncle
    if uncle y is Red {
          color parent and uncle black
          color grandparent red
          set z to grandparent
    }
    else {  // the uncle is black
         if (triangle) {
                  set z to parent
                  rotate to parent
                }
         // rotate the grandparent and finish
         color parent of z black
         color grandparent of z red
         rotate grand parent of z
        }
  } // end while
 color root black
}
```



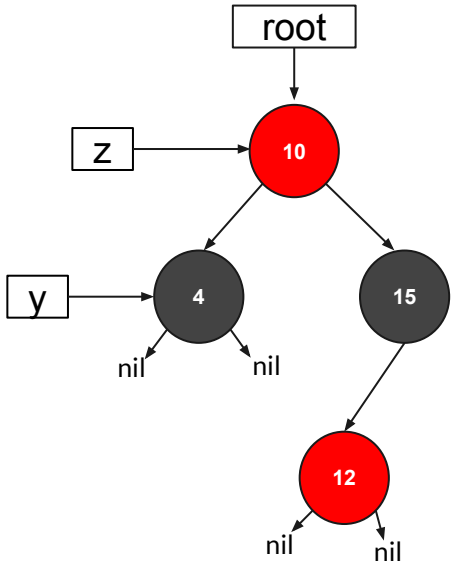
Fixup pseudocode: Case 1

```
RB-Insert-fixup(T,z) {
 while(z's parent is Red) {
    set y to be z's uncle
    if uncle y is Red {
          color parent and uncle black
          color grandparent red
          set z to grandparent
    }
    else {  // the uncle is black
         if (triangle) {
                  set z to parent
                  rotate to parent
                }
         // rotate the grandparent and finish
         color parent of z black
         color grandparent of z red
         rotate grand parent of z
        }
  } // end while
 color root black
}
```



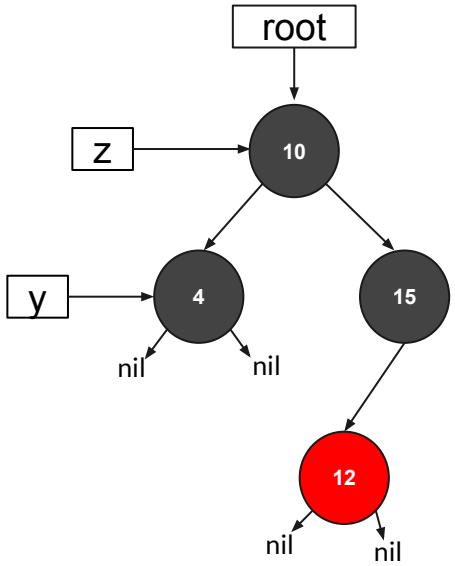
Fixup pseudocode: Case 1

```
RB-Insert-fixup(T,z) {
 while(z's parent is Red) {
    set y to be z's uncle
    if uncle y is Red {
          color parent and uncle black
          color grandparent red
          set z to grandparent
    }
    else {  // the uncle is black
         if (triangle) {
                   set z to parent
                   rotate to parent
                  }
         // rotate the grandparent and finish
         color parent of z black
         color grandparent of z red
         rotate grand parent of z
        }
 } // end while
 color root black
}
```



Fixup pseudocode: Case 1

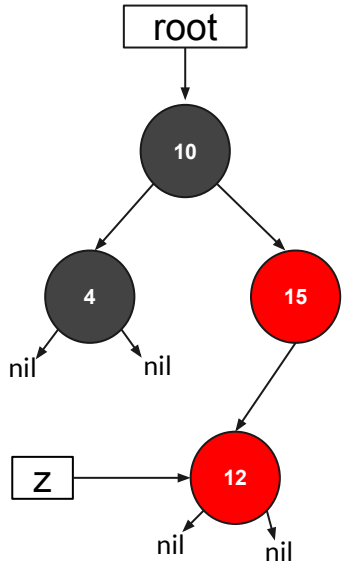
```
RB-Insert-fixup(T,z) {
 while(z's parent is Red) {
    set y to be z's uncle
    if uncle y is Red {
          color parent and uncle black
          color grandparent red
          set z to grandparent
    }
    else {  // the uncle is black
         if (triangle) {
                   set z to parent
                   rotate to parent
                  }
         // rotate the grandparent and finish
         color parent of z black
         color grandparent of z red
         rotate grand parent of z
        }
 } // end while
 color root black
}
```



Fixup pseudocode: case 2

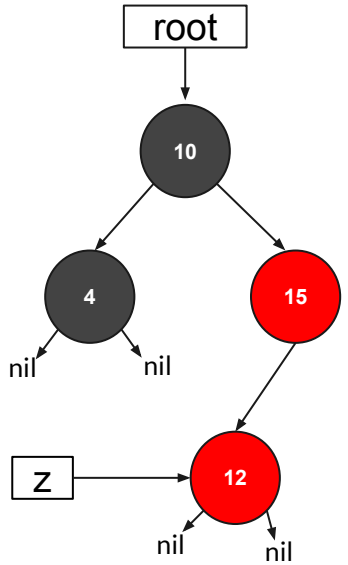
Fixup pseudocode: Case 2

```
RB-Insert-fixup(T,z) {
 while(z's parent is Red) {
    set y to be z's uncle
    if uncle y is Red {
          color parent and uncle black
          color grandparent red
          set z to grandparent
    }
    else {  // the uncle is black
          if (triangle) {
                    set z to parent
                    rotate to parent
                  }
          // rotate the grandparent and finish
          color parent of z black
          color grandparent of z red
          rotate grand parent of z
        }
  } // end while
 color root black
}
```



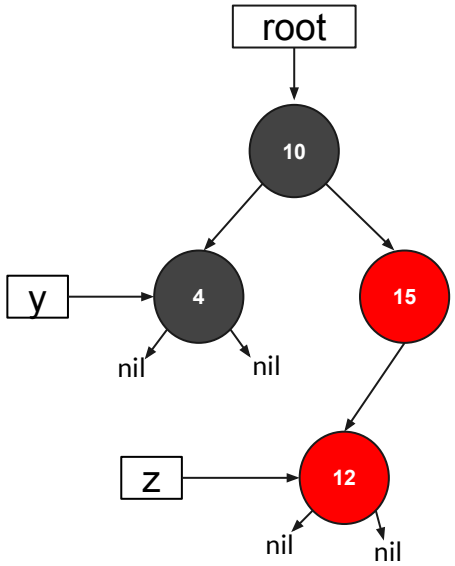
Fixup pseudocode: Case 2

```
RB-Insert-fixup(T,z) {
 while(z's parent is Red) {
    set y to be z's uncle
    if uncle y is Red {
          color parent and uncle black
          color grandparent red
          set z to grandparent
    }
    else {  // the uncle is black
         if (triangle) {
                   set z to parent
                   rotate to parent
                  }
         // rotate the grandparent and finish
         color parent of z black
         color grandparent of z red
         rotate grand parent of z
        }
  } // end while
 color root black
}
```



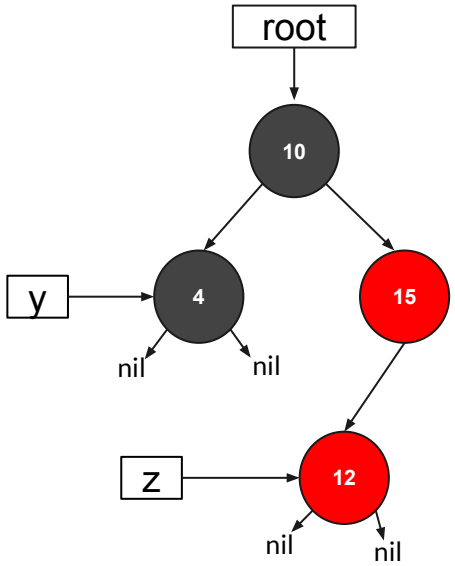
Fixup pseudocode: Case 2

```
RB-Insert-fixup(T,z) {
 while(z's parent is Red) {
    set y to be z's uncle
    if uncle y is Red {
          color parent and uncle black
          color grandparent red
          set z to grandparent
    }
    else {  // the uncle is black
         if (triangle) {
                   set z to parent
                   rotate to parent
                  }
         // rotate the grandparent and finish
         color parent of z black
         color grandparent of z red
         rotate grand parent of z
        }
  } // end while
 color root black
}
```



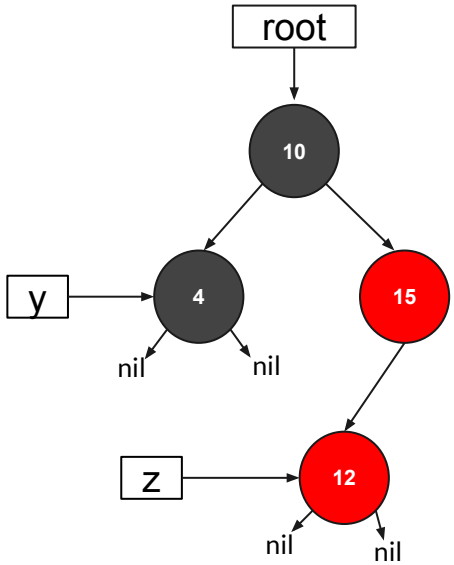
Fixup pseudocode: Case 2

```
RB-Insert-fixup(T,z) {
 while(z's parent is Red) {
    set y to be z's uncle
    if uncle y is Red {
          color parent and uncle black
          color grandparent red
          set z to grandparent
    }
    else {  // the uncle is black
         if (triangle) {
                   set z to parent
                   rotate to parent
                  }
         // rotate the grandparent and finish
         color parent of z black
         color grandparent of z red
         rotate grand parent of z
        }
  } // end while
 color root black
}
```



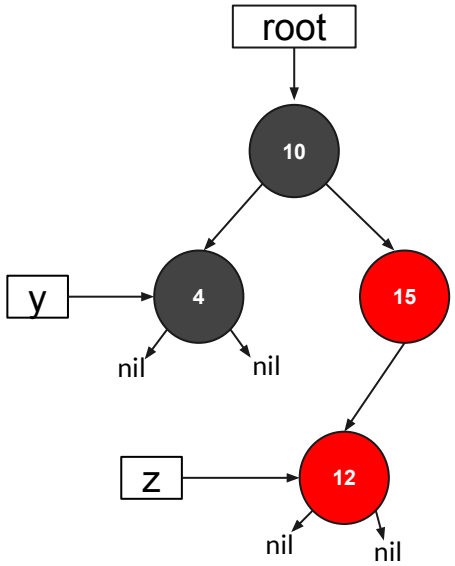
Fixup pseudocode: Case 2

```
RB-Insert-fixup(T,z) {
 while(z's parent is Red) {
    set y to be z's uncle
    if uncle y is Red {
          color parent and uncle black
          color grandparent red
          set z to grandparent
    }
    else {  // the uncle is black
         if (triangle) {
                  set z to parent
                  rotate to parent
                }
         // rotate the grandparent and finish
         color parent of z black
         color grandparent of z red
         rotate grand parent of z
       }
  } // end while
 color root black
}
```



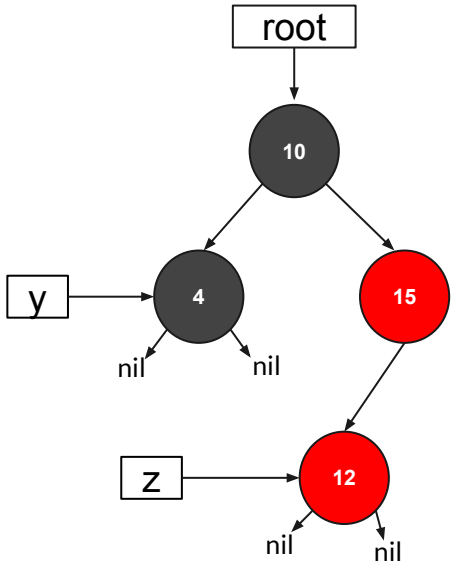
Fixup pseudocode: Case 2

```
RB-Insert-fixup(T,z) {
 while(z's parent is Red) {
    set y to be z's uncle
    if uncle y is Red {
          color parent and uncle black
          color grandparent red
          set z to grandparent
    }
    else {  // the uncle is black
         if (triangle) {
                   set z to parent
                   rotate to parent
                  }
         // rotate the grandparent and finish
         color parent of z black
         color grandparent of z red
         rotate grand parent of z
        }
  } // end while
 color root black
}
```



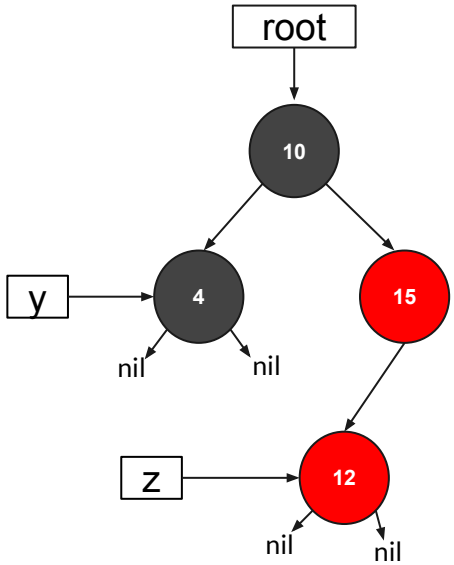
Fixup pseudocode: Case 2

```
RB-Insert-fixup(T,z) {
 while(z's parent is Red) {
    set y to be z's uncle
    if uncle y is Red {
          color parent and uncle black
          color grandparent red
          set z to grandparent
    }
    else {  // the uncle is black
          if (triangle) {
                    set z to parent
                    rotate to parent
                  }
          // rotate the grandparent and finish
          color parent of z black
          color grandparent of z red
          rotate grand parent of z
        }
  } // end while
 color root black
}
```



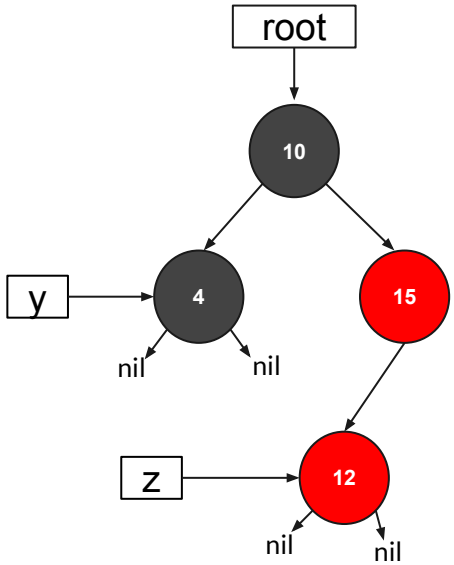
Fixup pseudocode: Case 2

```
RB-Insert-fixup(T,z) {
 while(z's parent is Red) {
    set y to be z's uncle
    if uncle y is Red {
          color parent and uncle black
          color grandparent red
          set z to grandparent
    }
    else {  // the uncle is black
         if (triangle) {
                   set z to parent
                   rotate to parent
                  }
         // rotate the grandparent and finish
         color parent of z black
         color grandparent of z red
         rotate grand parent of z
        }
  } // end while
 color root black
}
```



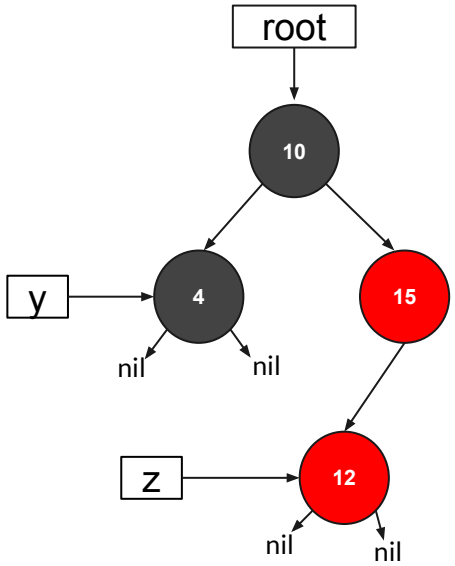
Fixup pseudocode: Case 2

```
RB-Insert-fixup(T,z) {
 while(z's parent is Red) {
    set y to be z's uncle
    if uncle y is Red {
          color parent and uncle black
          color grandparent red
          set z to grandparent
    }
    else {  // the uncle is black
         if (triangle) {
                   set z to parent
                   rotate to parent
                  }
         // rotate the grandparent and finish
         color parent of z black
         color grandparent of z red
         rotate grand parent of z
        }
  } // end while
 color root black
}
```



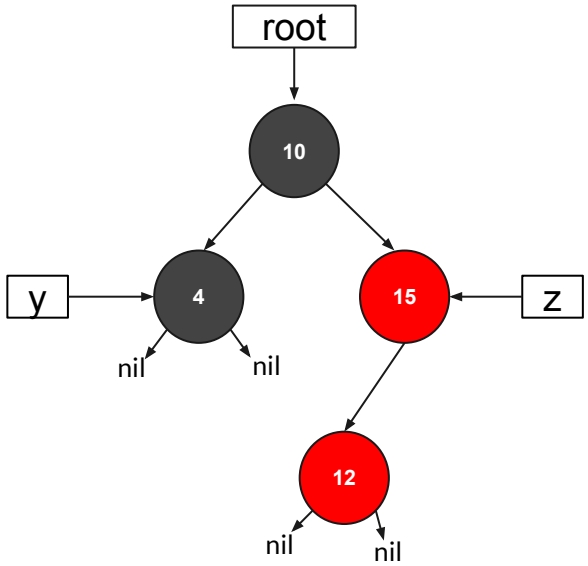
Fixup pseudocode: Case 2

```
RB-Insert-fixup(T,z) {
 while(z's parent is Red) {
    set y to be z's uncle
    if uncle y is Red {
          color parent and uncle black
          color grandparent red
          set z to grandparent
    }
    else {  // the uncle is black
         if (triangle) {
                   set z to parent
                   rotate to parent
                  }
         // rotate the grandparent and finish
         color parent of z black
         color grandparent of z red
         rotate grand parent of z
        }
  } // end while
 color root black
}
```



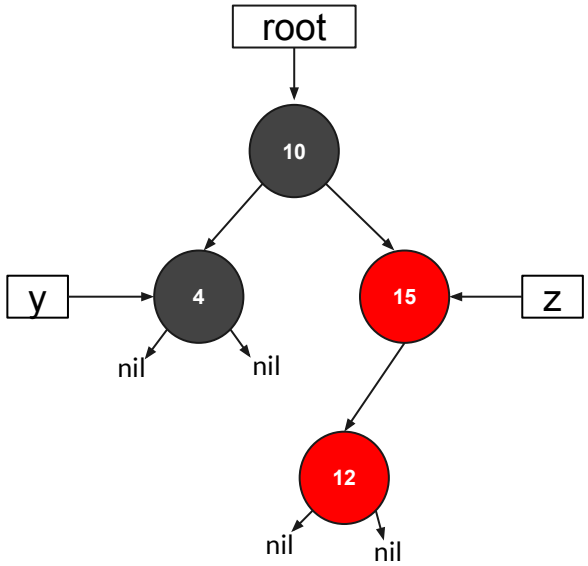
Fixup pseudocode: Case 2

```
RB-Insert-fixup(T,z) {
 while(z's parent is Red) {
    set y to be z's uncle
    if uncle y is Red {
          color parent and uncle black
          color grandparent red
          set z to grandparent
    }
    else {  // the uncle is black
         if (triangle) {
                  set z to parent
                  rotate to parent
                }
         // rotate the grandparent and finish
         color parent of z black
         color grandparent of z red
         rotate grand parent of z
        }
  } // end while
 color root black
}
```



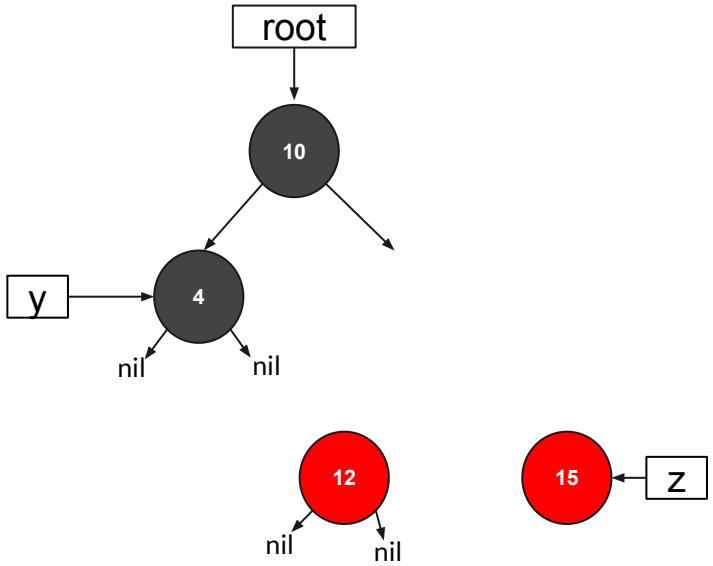
Fixup pseudocode: Case 2

```
RB-Insert-fixup(T,z) {
 while(z's parent is Red) {
    set y to be z's uncle
    if uncle y is Red {
          color parent and uncle black
          color grandparent red
          set z to grandparent
    }
    else {  // the uncle is black
         if (triangle) {
                   set z to parent
                   rotate to parent
                  }
         // rotate the grandparent and finish
         color parent of z black
         color grandparent of z red
         rotate grand parent of z
        }
  } // end while
 color root black
}
```



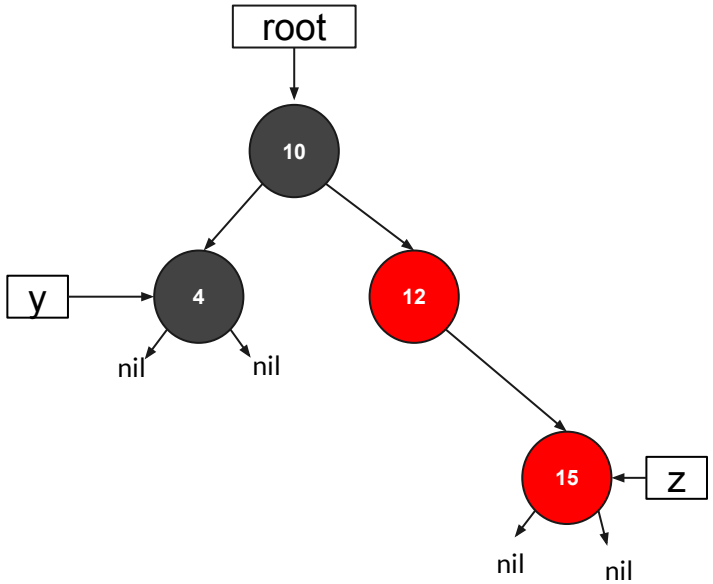
Fixup pseudocode: Case 2

```
RB-Insert-fixup(T,z) {
 while(z's parent is Red) {
    set y to be z's uncle
    if uncle y is Red {
          color parent and uncle black
          color grandparent red
          set z to grandparent
    }
    else {  // the uncle is black
         if (triangle) {
                   set z to parent
                   rotate to parent
                  }
         // rotate the grandparent and finish
         color parent of z black
         color grandparent of z red
         rotate grand parent of z
        }
  } // end while
 color root black
}
```



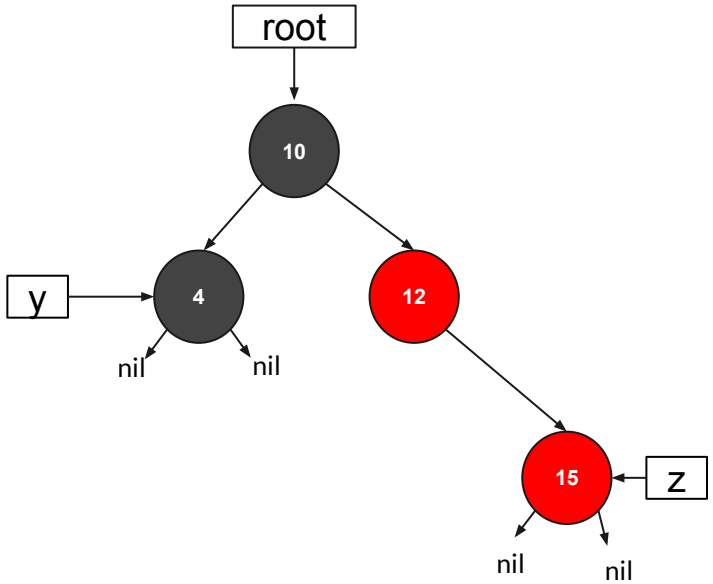
Fixup pseudocode: Case 2

```
RB-Insert-fixup(T,z) {
 while(z's parent is Red) {
    set y to be z's uncle
    if uncle y is Red {
          color parent and uncle black
          color grandparent red
          set z to grandparent
    }
    else {  // the uncle is black
         if (triangle) {
                   set z to parent
                   rotate to parent
                  }
         // rotate the grandparent and finish
         color parent of z black
         color grandparent of z red
         rotate grand parent of z
        }
  } // end while
 color root black
}
```



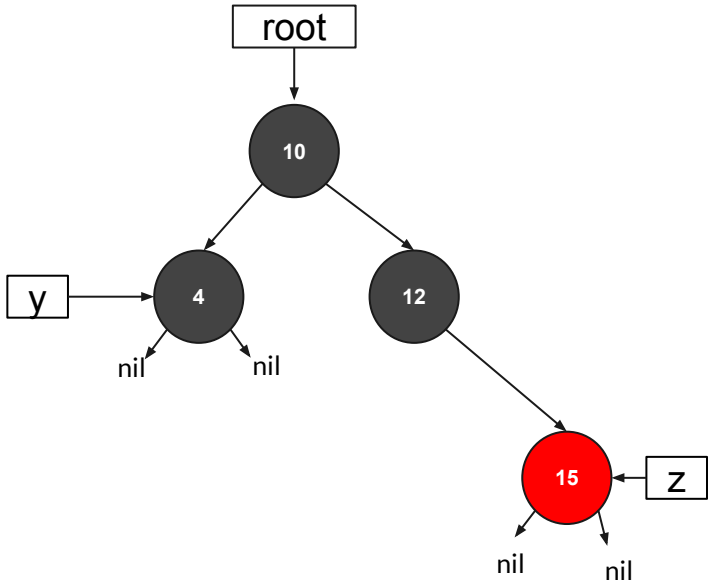
Fixup pseudocode: Case 2

```
RB-Insert-fixup(T,z) {
 while(z's parent is Red) {
    set y to be z's uncle
    if uncle y is Red {
          color parent and uncle black
          color grandparent red
          set z to grandparent
    }
    else {  // the uncle is black
         if (triangle) {
                   set z to parent
                   rotate to parent
                  }
         // rotate the grandparent and finish
         color parent of z black
         color grandparent of z red
         rotate grand parent of z
        }
  } // end while
 color root black
}
```



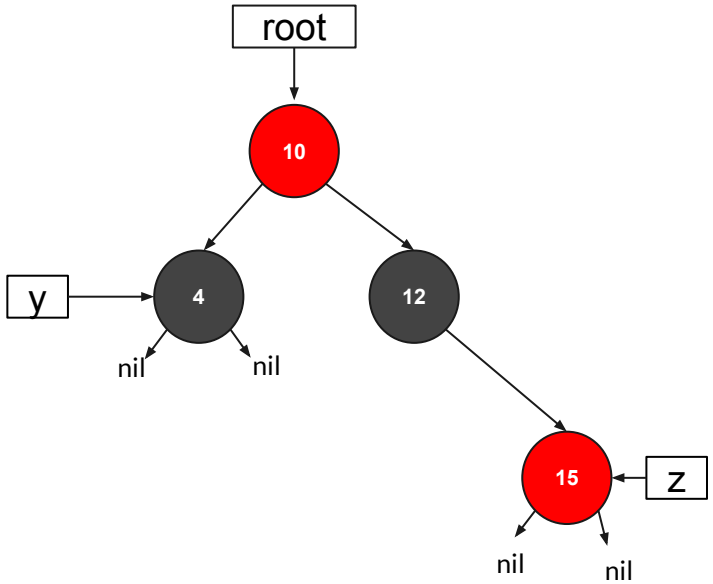
Fixup pseudocode: Case 2

```
RB-Insert-fixup(T,z) {
 while(z's parent is Red) {
    set y to be z's uncle
    if uncle y is Red {
          color parent and uncle black
          color grandparent red
          set z to grandparent
    }
    else {  // the uncle is black
         if (triangle) {
                   set z to parent
                   rotate to parent
                  }
         // rotate the grandparent and finish
         color parent of z black
         color grandparent of z red
         rotate grand parent of z
        }
  } // end while
 color root black
}
```



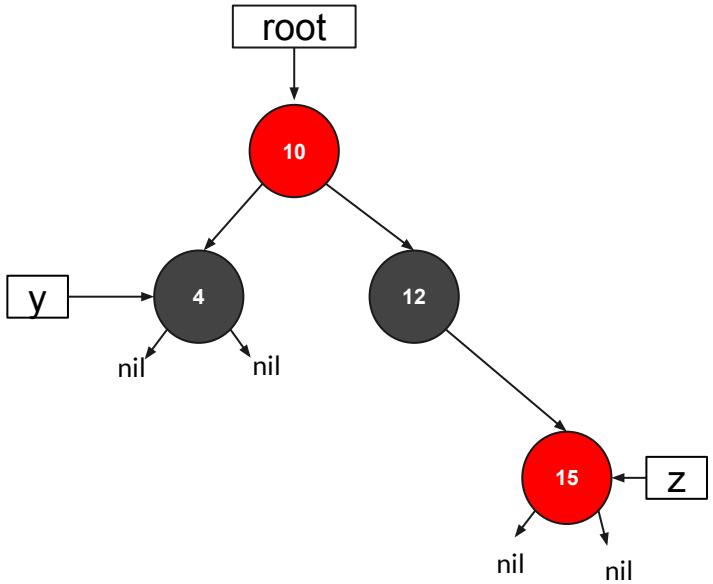
Fixup pseudocode: Case 2

```
RB-Insert-fixup(T,z) {
 while(z's parent is Red) {
    set y to be z's uncle
    if uncle y is Red {
          color parent and uncle black
          color grandparent red
          set z to grandparent
    }
    else {  // the uncle is black
         if (triangle) {
                   set z to parent
                   rotate to parent
                  }
         // rotate the grandparent and finish
         color parent of z black
         color grandparent of z red
         rotate grand parent of z
        }
  } // end while
 color root black
}
```



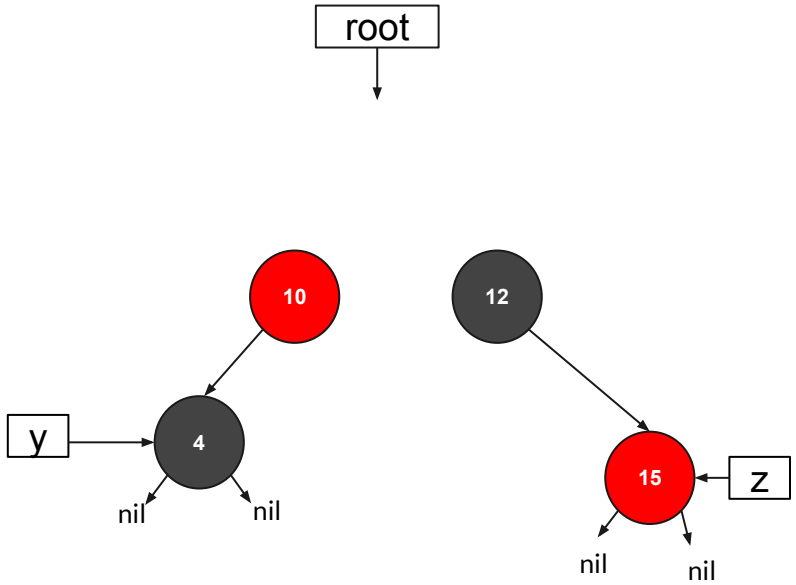
Fixup pseudocode: Case 2

```
RB-Insert-fixup(T,z) {
 while(z's parent is Red) {
    set y to be z's uncle
    if uncle y is Red {
          color parent and uncle black
          color grandparent red
          set z to grandparent
    }
    else {  // the uncle is black
         if (triangle) {
                   set z to parent
                   rotate to parent
                  }
         // rotate the grandparent and finish
         color parent of z black
         color grandparent of z red
         rotate grand parent of z
        }
  } // end while
 color root black
}
```



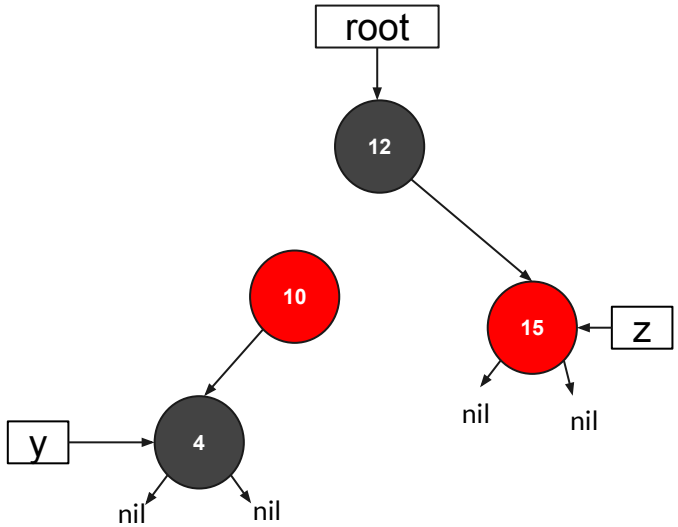
Fixup pseudocode: Case 2

```
RB-Insert-fixup(T,z) {
 while(z's parent is Red) {
    set y to be z's uncle
    if uncle y is Red {
          color parent and uncle black
          color grandparent red
          set z to grandparent
    }
    else {  // the uncle is black
         if (triangle) {
                   set z to parent
                   rotate to parent
                  }
         // rotate the grandparent and finish
         color parent of z black
         color grandparent of z red
         rotate grand parent of z
        }
  } // end while
 color root black
}
```



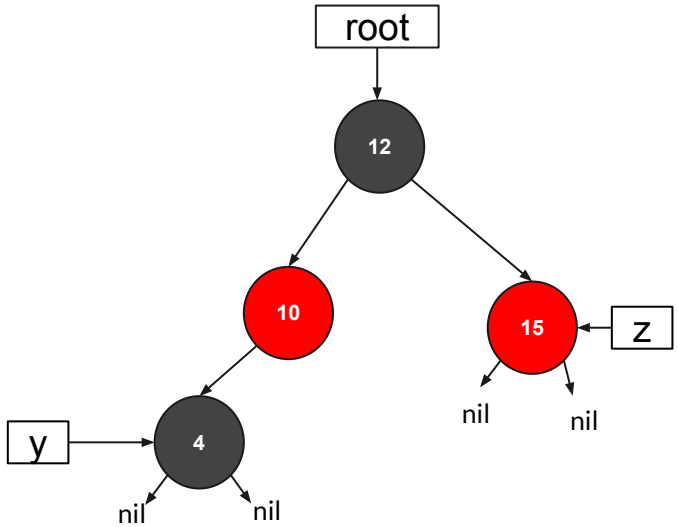
Fixup pseudocode: Case 2

```
RB-Insert-fixup(T,z) {
 while(z's parent is Red) {
    set y to be z's uncle
    if uncle y is Red {
          color parent and uncle black
          color grandparent red
          set z to grandparent
    }
    else {  // the uncle is black
         if (triangle) {
                  set z to parent
                  rotate to parent
                 }
         // rotate the grandparent and finish
         color parent of z black
         color grandparent of z red
         rotate grand parent of z
        }
  } // end while
 color root black
}
```



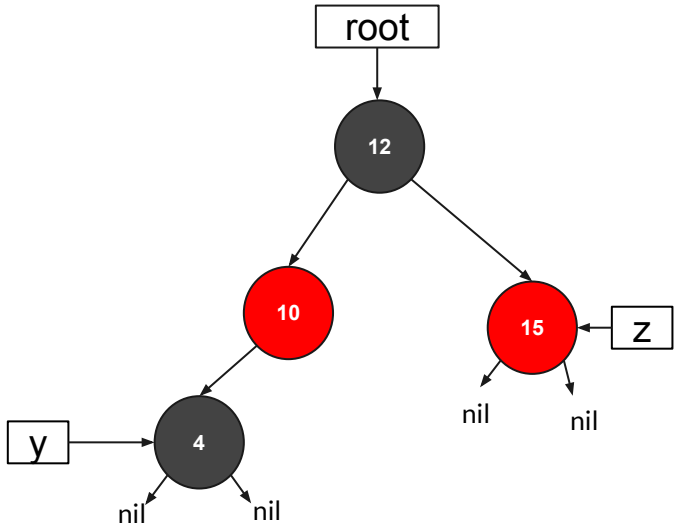
Fixup pseudocode: Case 2

```
RB-Insert-fixup(T,z) {
 while(z's parent is Red) {
    set y to be z's uncle
    if uncle y is Red {
          color parent and uncle black
          color grandparent red
          set z to grandparent
    }
    else {  // the uncle is black
         if (triangle) {
                   set z to parent
                   rotate to parent
                  }
         // rotate the grandparent and finish
         color parent of z black
         color grandparent of z red
         rotate grand parent of z
        }
  } // end while
 color root black
}
```



Fixup pseudocode: Case 2

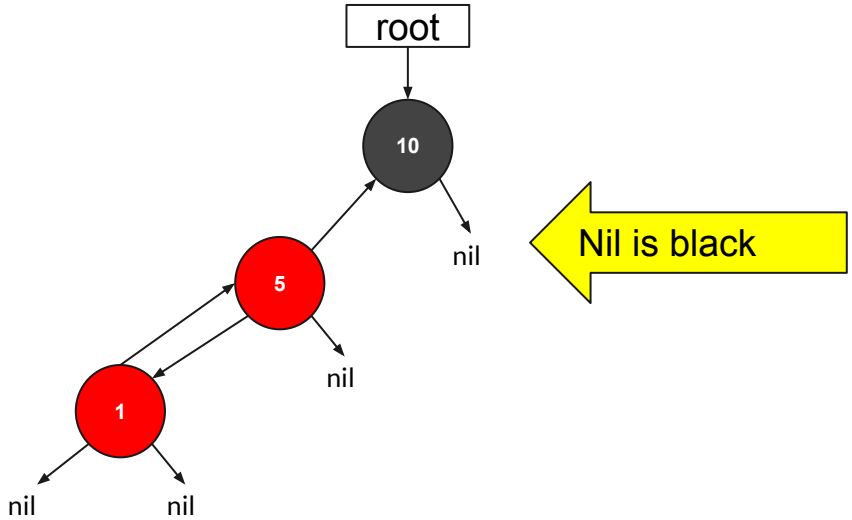
```
RB-Insert-fixup(T,z) {
 while(z's parent is Red) {
    set y to be z's uncle
    if uncle y is Red {
          color parent and uncle black
          color grandparent red
          set z to grandparent
    }
    else {  // the uncle is black
         if (triangle) {
                   set z to parent
                   rotate to parent
                  }
         // rotate the grandparent and finish
         color parent of z black
         color grandparent of z red
         rotate grand parent of z
        }
 } // end while
 color root black
}
```



Fixup pseudocode: case 3

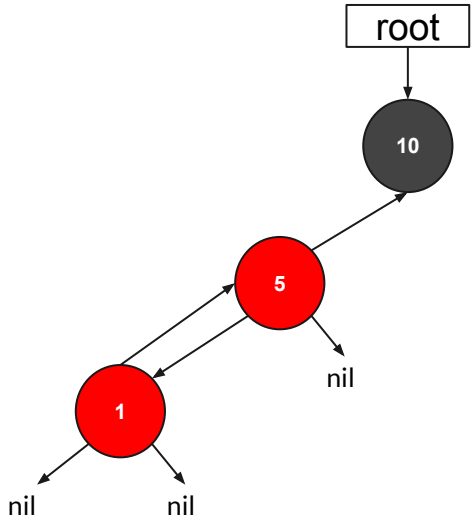
Fixup pseudocode: Case 3

```
RB-Insert-fixup(T,z) {
 while(z's parent is Red) {
    set y to be z's uncle
    if uncle y is Red {
          color parent and uncle black
          color grandparent red
          set z to grandparent
    }
    else {  // the uncle is black
         if (triangle) {
                   set z to parent
                   rotate to parent
                  }
         // rotate the grandparent and finish
         color parent of z black
         color grandparent of z red
         rotate grand parent of z
        }
  } // end while
 color root black
}
```



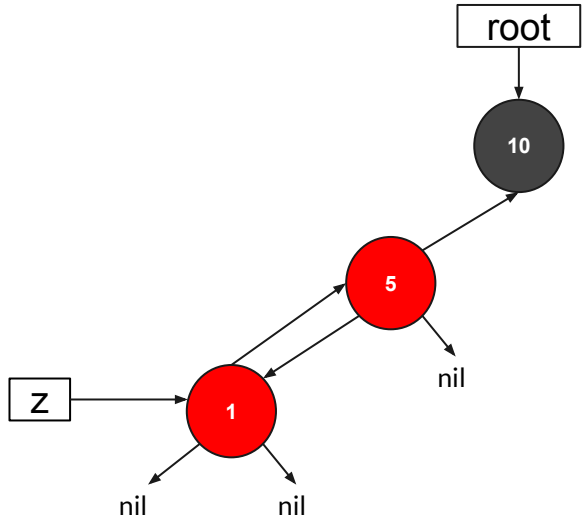
Fixup pseudocode: Case 3

```
RB-Insert-fixup(T,z) {
 while(z's parent is Red) {
    set y to be z's uncle
    if uncle y is Red {
          color parent and uncle black
          color grandparent red
          set z to grandparent
    }
    else {  // the uncle is black
         if (triangle) {
                   set z to parent
                   rotate to parent
                  }
         // rotate the grandparent and finish
         color parent of z black
         color grandparent of z red
         rotate grand parent of z
        }
  } // end while
 color root black
}
```



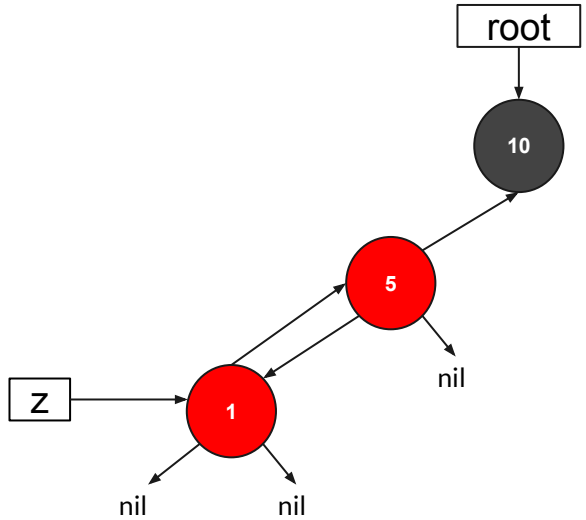
Fixup pseudocode: Case 3

```
RB-Insert-fixup(T,z) {
 while(z's parent is Red) {
    set y to be z's uncle
    if uncle y is Red {
          color parent and uncle black
          color grandparent red
          set z to grandparent
    }
    else {  // the uncle is black
          if (triangle) {
                    set z to parent
                    rotate to parent
                  }
          // rotate the grandparent and finish
          color parent of z black
          color grandparent of z red
          rotate grand parent of z
        }
  } // end while
 color root black
}
```



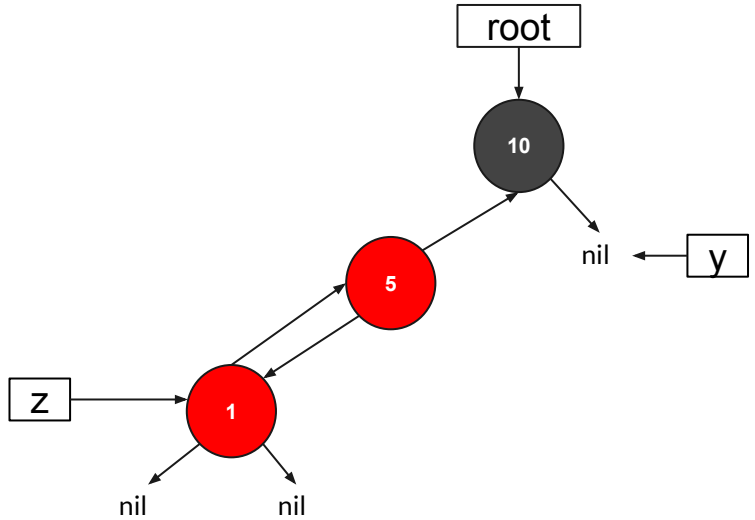
Fixup pseudocode: Case 3

```
RB-Insert-fixup(T,z) {
 while(z's parent is Red) {
    set y to be z's uncle
    if uncle y is Red {
          color parent and uncle black
          color grandparent red
          set z to grandparent
    }
    else {  // the uncle is black
          if (triangle) {
                    set z to parent
                    rotate to parent
                  }
          // rotate the grandparent and finish
          color parent of z black
          color grandparent of z red
          rotate grand parent of z
        }
  } // end while
 color root black
}
```



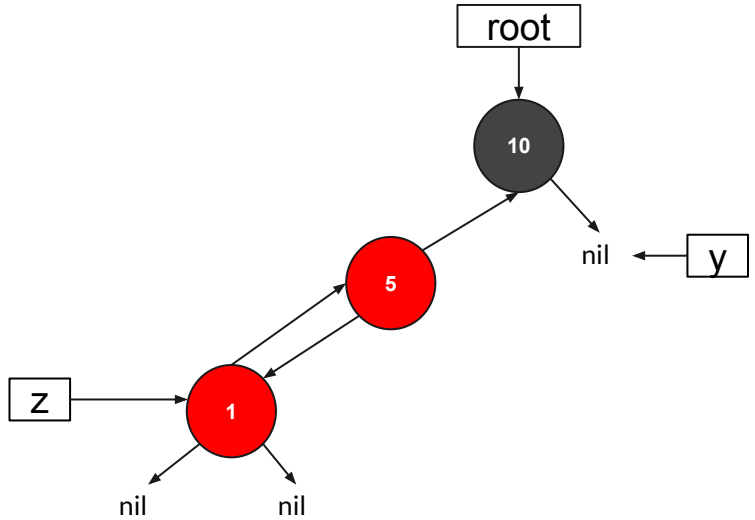
Fixup pseudocode: Case 3

```
RB-Insert-fixup(T,z) {
 while(z's parent is Red) {
    set y to be z's uncle
    if uncle y is Red {
          color parent and uncle black
          color grandparent red
          set z to grandparent
    }
    else {  // the uncle is black
          if (triangle) {
                    set z to parent
                    rotate to parent
                  }
          // rotate the grandparent and finish
          color parent of z black
          color grandparent of z red
          rotate grand parent of z
        }
  } // end while
 color root black
}
```



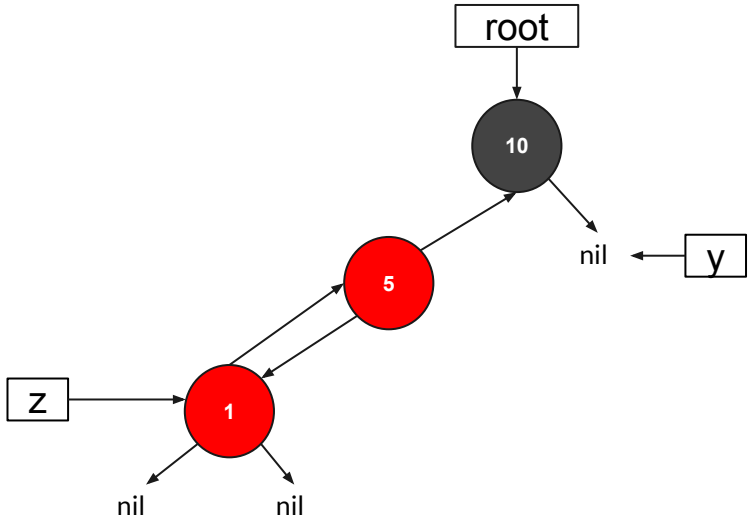
Fixup pseudocode: Case 3

```
RB-Insert-fixup(T,z) {
 while(z's parent is Red) {
    set y to be z's uncle
    if uncle y is Red {
          color parent and uncle black
          color grandparent red
          set z to grandparent
    }
    else {  // the uncle is black
         if (triangle) {
                  set z to parent
                  rotate to parent
                }
         // rotate the grandparent and finish
         color parent of z black
         color grandparent of z red
         rotate grand parent of z
        }
  } // end while
 color root black
}
```



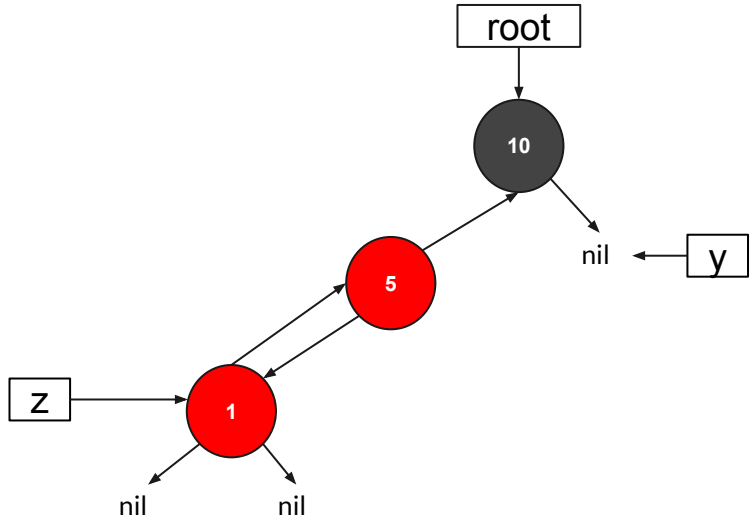
Fixup pseudocode: Case 3

```
RB-Insert-fixup(T,z) {
 while(z's parent is Red) {
    set y to be z's uncle
    if uncle y is Red {
          color parent and uncle black
          color grandparent red
          set z to grandparent
    }
    else {  // the uncle is black
         if (triangle) {
                  set z to parent
                  rotate to parent
                }
         // rotate the grandparent and finish
         color parent of z black
         color grandparent of z red
         rotate grand parent of z
        }
  } // end while
 color root black
}
```



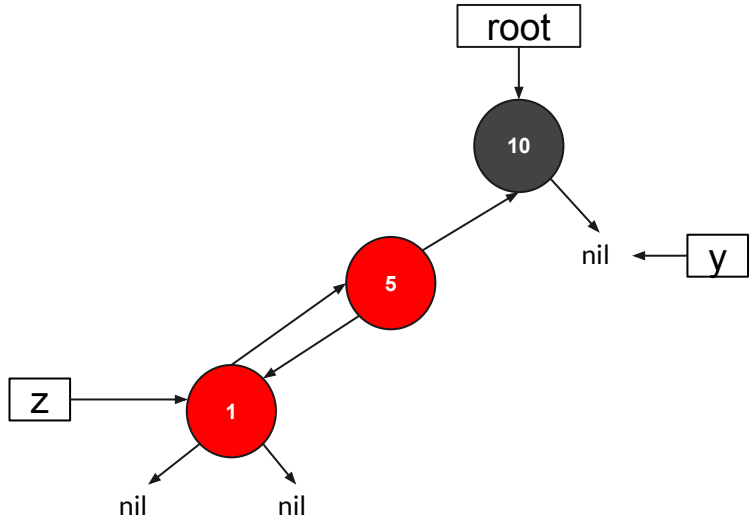
Fixup pseudocode: Case 3

```
RB-Insert-fixup(T,z) {
 while(z's parent is Red) {
    set y to be z's uncle
    if uncle y is Red {
          color parent and uncle black
          color grandparent red
          set z to grandparent
    }
    else {  // the uncle is black
          if (triangle) {
                set z to parent
                rotate to parent
               }
          // rotate the grandparent and finish
          color parent of z black
          color grandparent of z red
          rotate grand parent of z
        }
  } // end while
 color root black
}
```



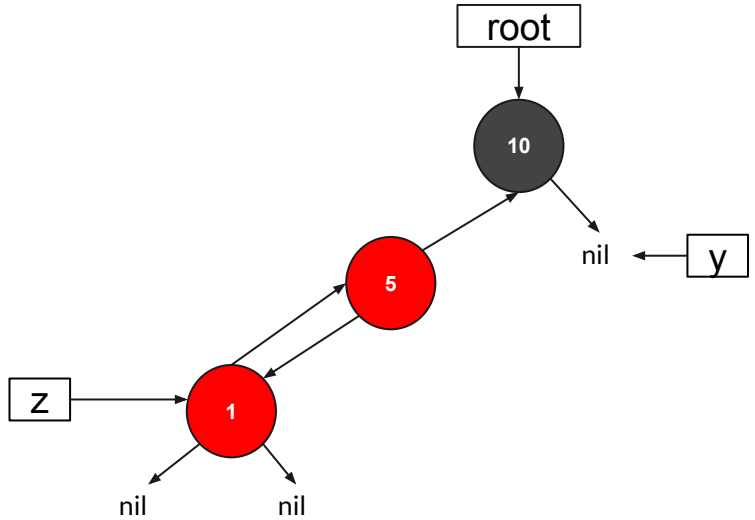
Fixup pseudocode: Case 3

```
RB-Insert-fixup(T,z) {
 while(z's parent is Red) {
    set y to be z's uncle
    if uncle y is Red {
          color parent and uncle black
          color grandparent red
          set z to grandparent
    }
    else {  // the uncle is black
          if (triangle) {
                    set z to parent
                    rotate to parent
                  }
          // rotate the grandparent and finish
          color parent of z black
          color grandparent of z red
          rotate grand parent of z
        }
  } // end while
 color root black
}
```



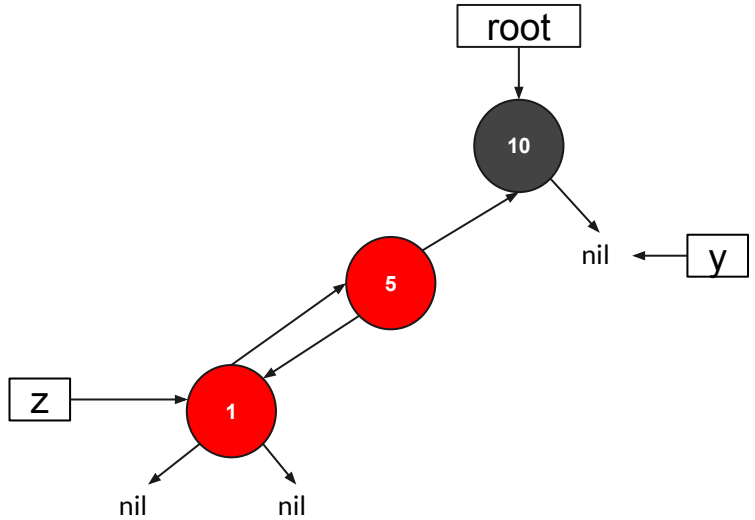
Fixup pseudocode: Case 3

```
RB-Insert-fixup(T,z) {
 while(z's parent is Red) {
    set y to be z's uncle
    if uncle y is Red {
          color parent and uncle black
          color grandparent red
          set z to grandparent
    }
    else {  // the uncle is black
         if (triangle) {
                    set z to parent
                    rotate to parent
                  }
         // rotate the grandparent and finish
         color parent of z black
         color grandparent of z red
         rotate grand parent of z
       }
  } // end while
 color root black
}
```



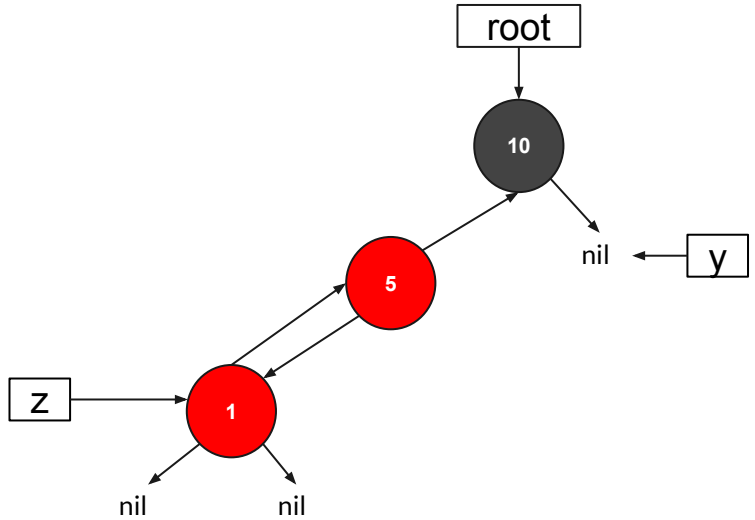
Fixup pseudocode: Case 3

```
RB-Insert-fixup(T,z) {
 while(z's parent is Red) {
    set y to be z's uncle
    if uncle y is Red {
          color parent and uncle black
          color grandparent red
          set z to grandparent
    }
    else {  // the uncle is black
         if (triangle) {
                    set z to parent
                    rotate to parent
                  }
         // rotate the grandparent and finish
         color parent of z black
         color grandparent of z red
         rotate grand parent of z
       }
  } // end while
 color root black
}
```



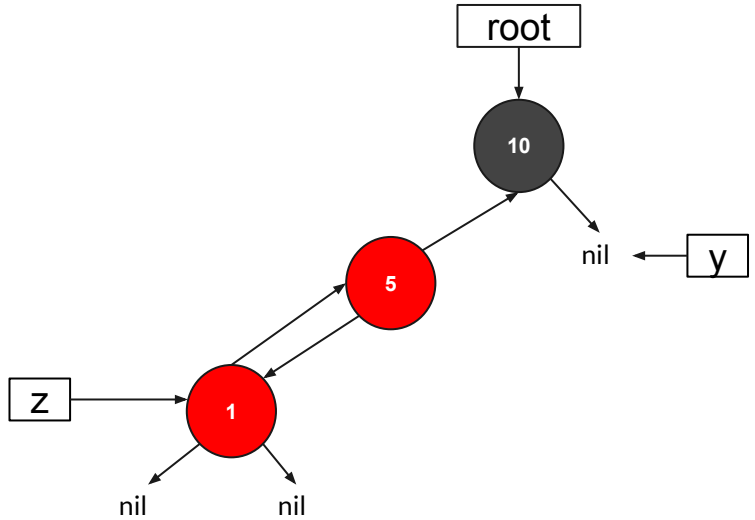
Fixup pseudocode: Case 3

```
RB-Insert-fixup(T,z) {
 while(z's parent is Red) {
    set y to be z's uncle
    if uncle y is Red {
          color parent and uncle black
          color grandparent red
          set z to grandparent
    }
    else {  // the uncle is black
         if (triangle) {
                    set z to parent
                    rotate to parent
                  }
         // rotate the grandparent and finish
         color parent of z black
         color grandparent of z red
         rotate grand parent of z
       }
  } // end while
 color root black
}
```



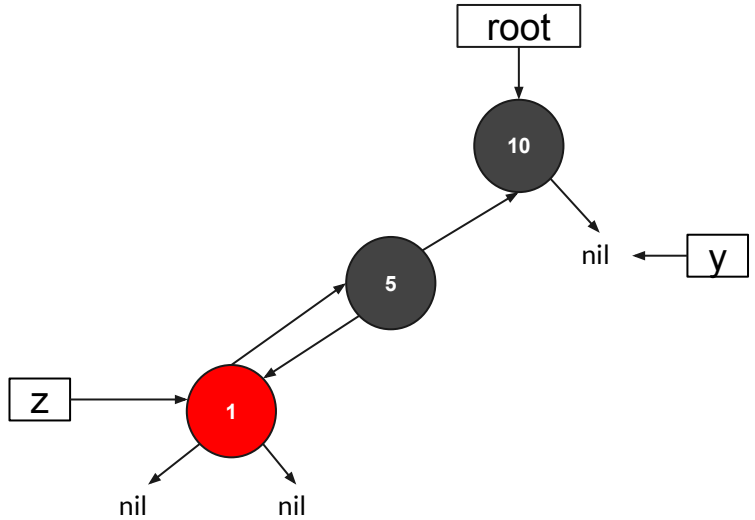
Fixup pseudocode: Case 3

```
RB-Insert-fixup(T,z) {
 while(z's parent is Red) {
    set y to be z's uncle
    if uncle y is Red {
          color parent and uncle black
          color grandparent red
          set z to grandparent
    }
    else {  // the uncle is black
         if (triangle) {
                  set z to parent
                  rotate to parent
                }
         // rotate the grandparent and finish
         color parent of z black
         color grandparent of z red
         rotate grand parent of z
        }
  } // end while
 color root black
}
```



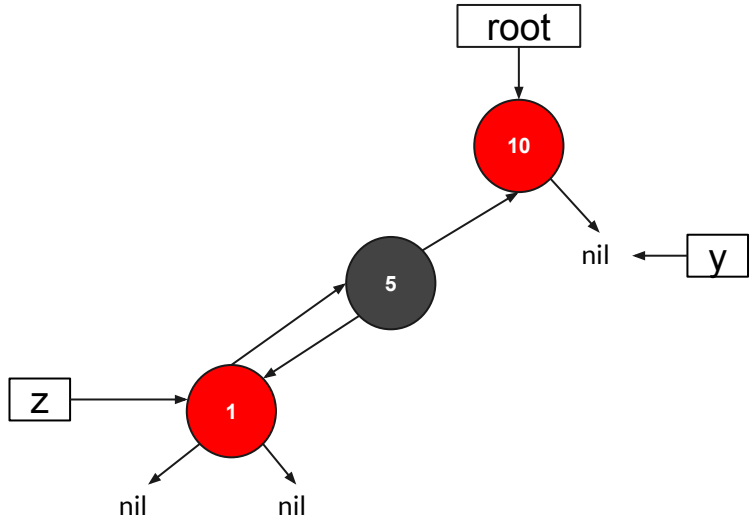
Fixup pseudocode: Case 3

```
RB-Insert-fixup(T,z) {
 while(z's parent is Red) {
    set y to be z's uncle
    if uncle y is Red {
          color parent and uncle black
          color grandparent red
          set z to grandparent
    }
    else {  // the uncle is black
         if (triangle) {
                  set z to parent
                  rotate to parent
                }
         // rotate the grandparent and finish
         color parent of z black
         color grandparent of z red
         rotate grand parent of z
        }
  } // end while
 color root black
}
```



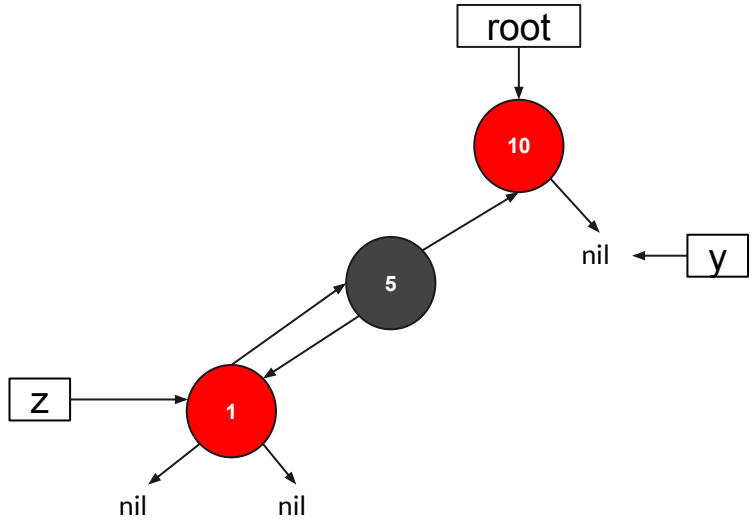
Fixup pseudocode: Case 3

```
RB-Insert-fixup(T,z) {
 while(z's parent is Red) {
    set y to be z's uncle
    if uncle y is Red {
          color parent and uncle black
          color grandparent red
          set z to grandparent
    }
    else {  // the uncle is black
          if (triangle) {
                    set z to parent
                    rotate to parent
                  }
          // rotate the grandparent and finish
          color parent of z black
          color grandparent of z red
          rotate grand parent of z
        }
  } // end while
 color root black
}
```



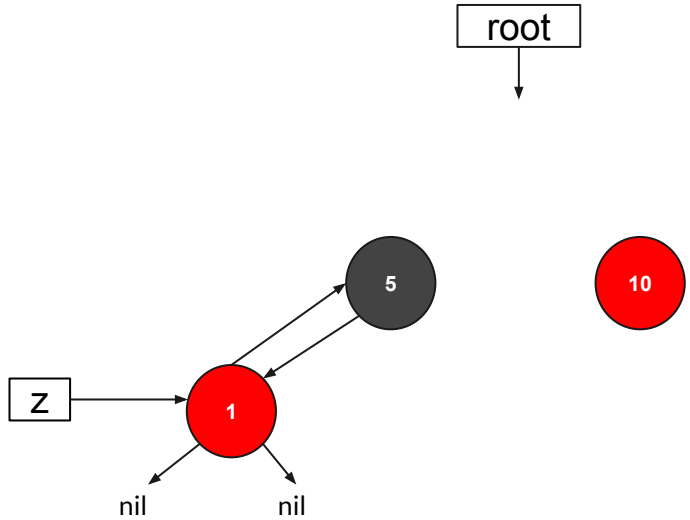
Fixup pseudocode: Case 3

```
RB-Insert-fixup(T,z) {
 while(z's parent is Red) {
    set y to be z's uncle
    if uncle y is Red {
          color parent and uncle black
          color grandparent red
          set z to grandparent
    }
    else {  // the uncle is black
         if (triangle) {
                  set z to parent
                  rotate to parent
                }
         // rotate the grandparent and finish
         color parent of z black
         color grandparent of z red
         rotate grand parent of z
       }
  } // end while
 color root black
}
```



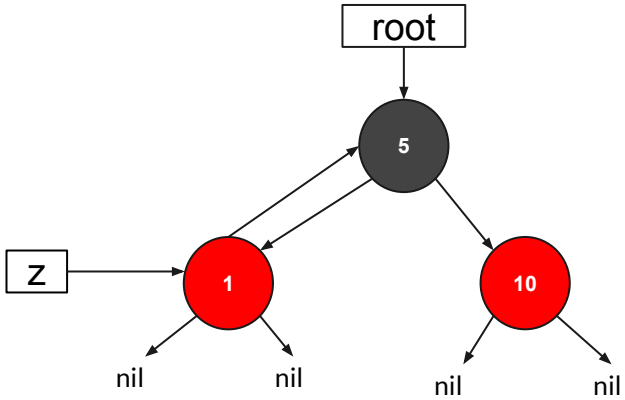
Fixup pseudocode: Case 3

```
RB-Insert-fixup(T,z) {
 while(z's parent is Red) {
    set y to be z's uncle
    if uncle y is Red {
          color parent and uncle black
          color grandparent red
          set z to grandparent
    }
    else {  // the uncle is black
         if (triangle) {
                  set z to parent
                  rotate to parent
                }
         // rotate the grandparent and finish
         color parent of z black
         color grandparent of z red
         rotate grand parent of z
       }
  } // end while
 color root black
}
```



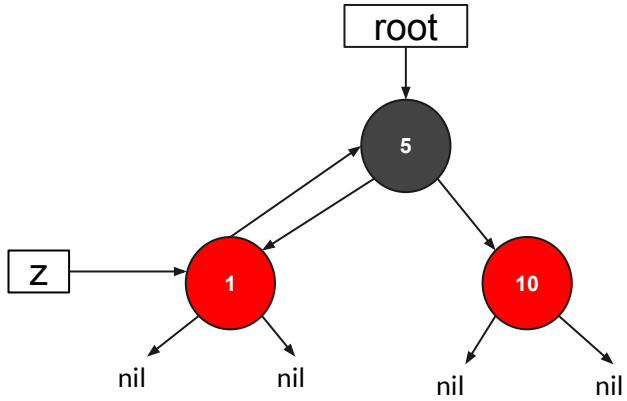
Fixup pseudocode: Case 3

```
RB-Insert-fixup(T,z) {
 while(z's parent is Red) {
    set y to be z's uncle
    if uncle y is Red {
          color parent and uncle black
          color grandparent red
          set z to grandparent
    }
    else {  // the uncle is black
         if (triangle) {
                  set z to parent
                  rotate to parent
                 }
         // rotate the grandparent and finish
         color parent of z black
         color grandparent of z red
         rotate grand parent of z
       }
  } // end while
 color root black
}
```



Fixup pseudocode: Case 3

```
RB-Insert-fixup(T,z) {
 while(z's parent is Red) {
    set y to be z's uncle
    if uncle y is Red {
          color parent and uncle black
          color grandparent red
          set z to grandparent
    }
    else {  // the uncle is black
         if (triangle) {
                  set z to parent
                  rotate to parent
                }
         // rotate the grandparent and finish
         color parent of z black
         color grandparent of z red
         rotate grand parent of z
       }
  } // end while
 color root black
}
```



Red-Black Tree Visualization

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

References

- Introduction to Operating Systems (Prof. Chester Rebeiro, IIT Madras)
- Understanding the Linux Kernel, 3rd Edition By Daniel P. Bovet, Marco Cesati
- Linux Kernel Development By Robert Love
- Red-black trees tutorial by Michael Sambol
- https://www.andrew.cmu.edu/user/mm6/95-771/examples/RedBlackTreeProject/dist/javadoc/redblacktreeproject/RedBlackTree.html