CS302 – Analysis and Design of Algorithms

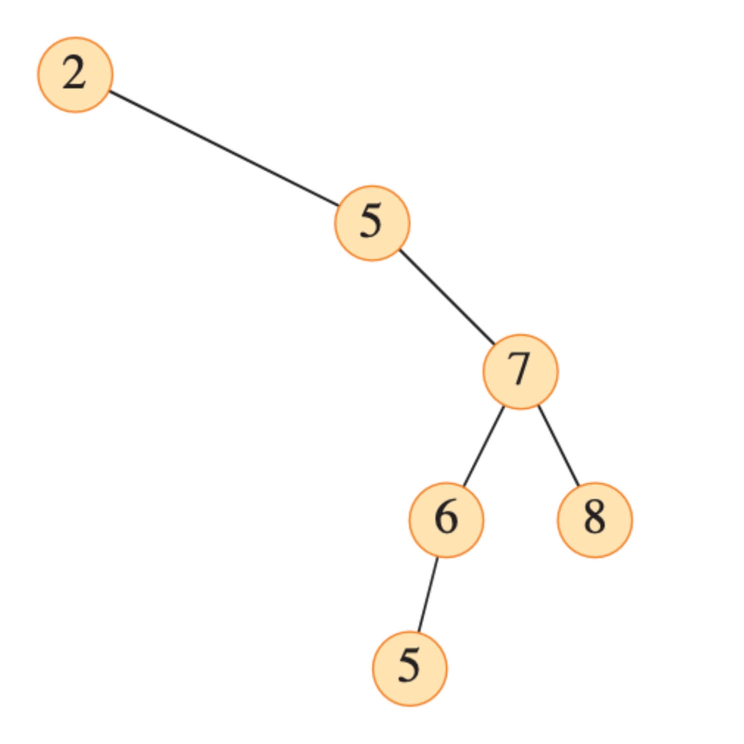
Red-Black Tree and Complexity Classes

Content

Content
Red-Black Trees
Rotations
Insertion
Deletion
Complexity Classes

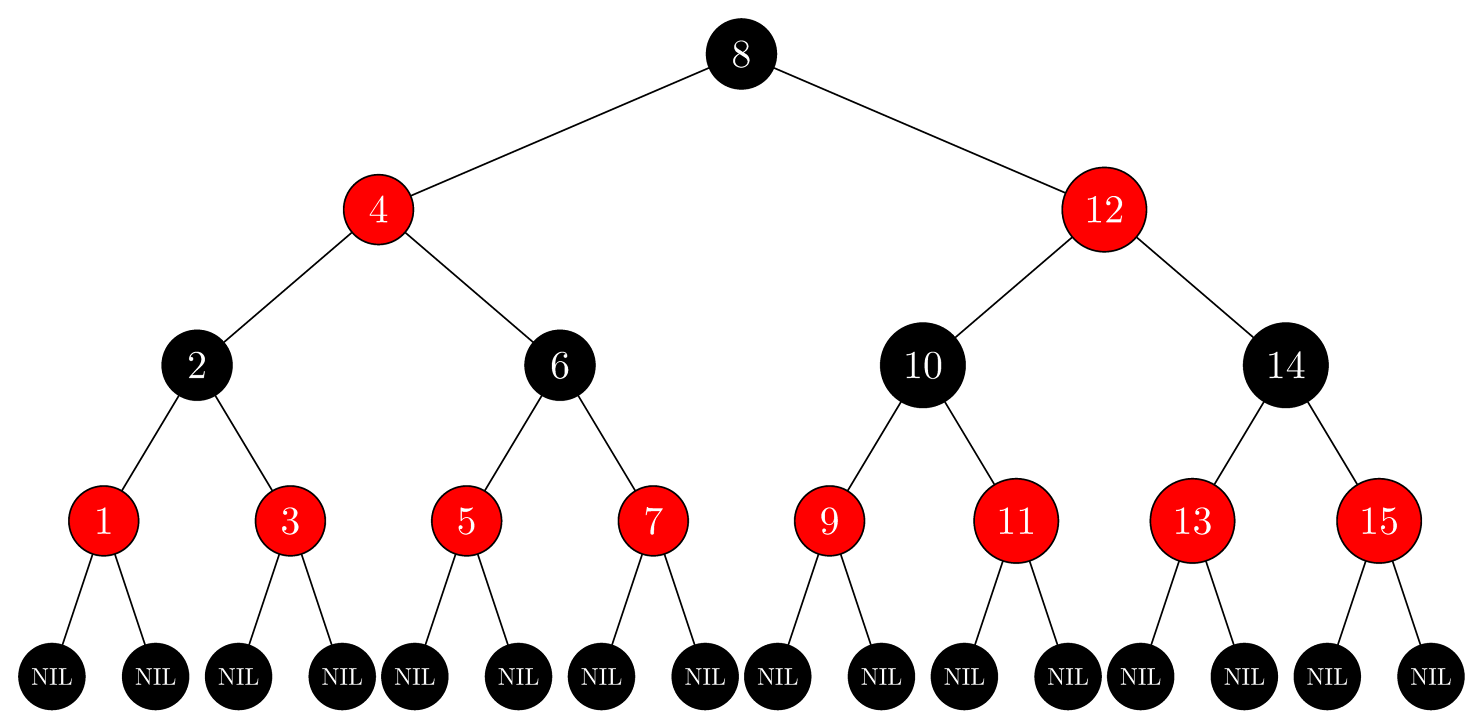
Red-Black Trees

- In BST, operations take $O(h)$ time, where $h = \lg n$ is the height of the tree.
- An unbalanced tree may run no faster than a linked list.



Red-Black Trees

- Red-black trees are binary search tree schemes that are balanced.
- They guarantee $O(\lg n)$ operations in the worst case.



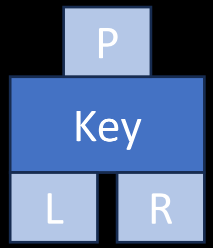
Red-Black Trees

- A red-black tree has an extra bit of storage per node: its color.
  - RED node
  - BLACK node

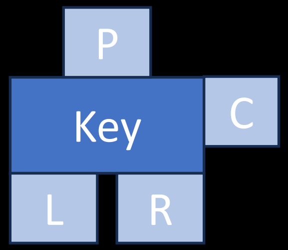
- The height of a red-black tree with n keys is at most $2\lg(n+1) = O(\lg \mathrm{n})$.

Red-Black Trees

- BST node



- Red-black tree node

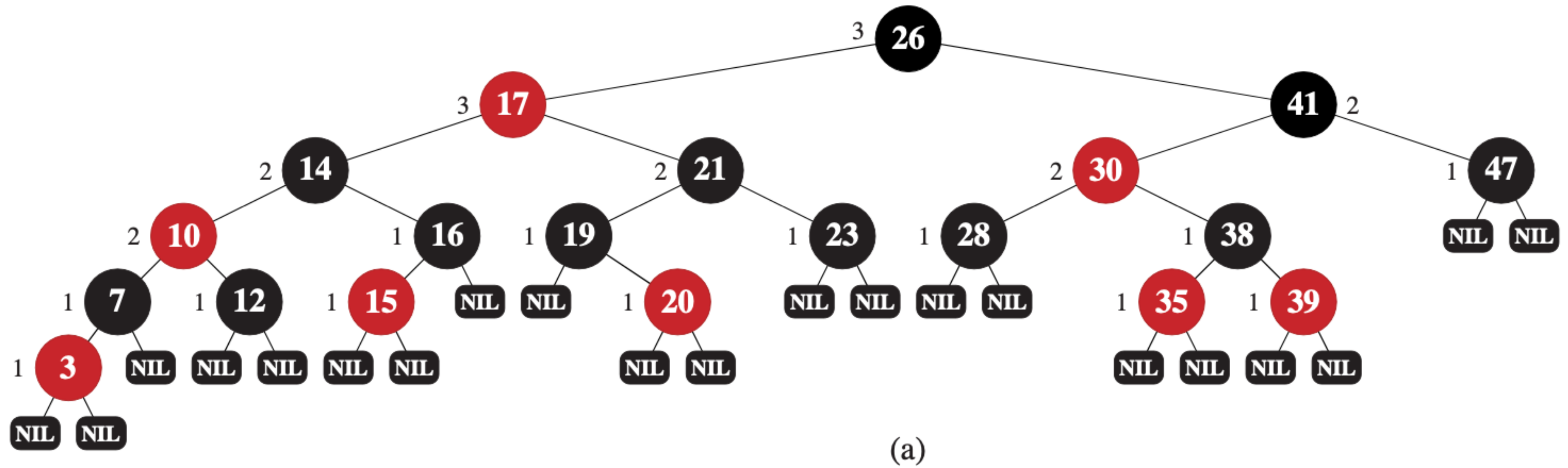


Red-Black Trees

- Properties of RB trees:
  1. Every node is either red or black.
  2. The root is black.
  3. Every leaf (NIL) is black.
  4. If a node is red, then both its children are black.
  5. For each node, all simple paths from the node to descendant leaves contain the same number of black nodes.

Red-Black Trees

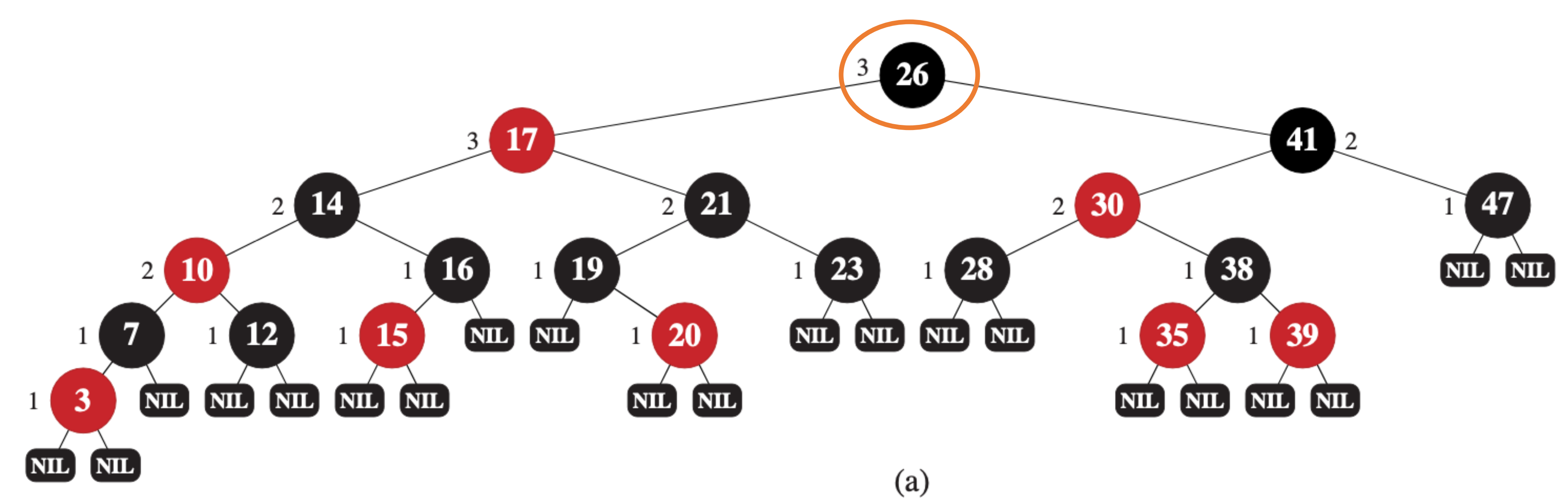
- Example
  - Every node is either red or black.



(a)

Red-Black Trees

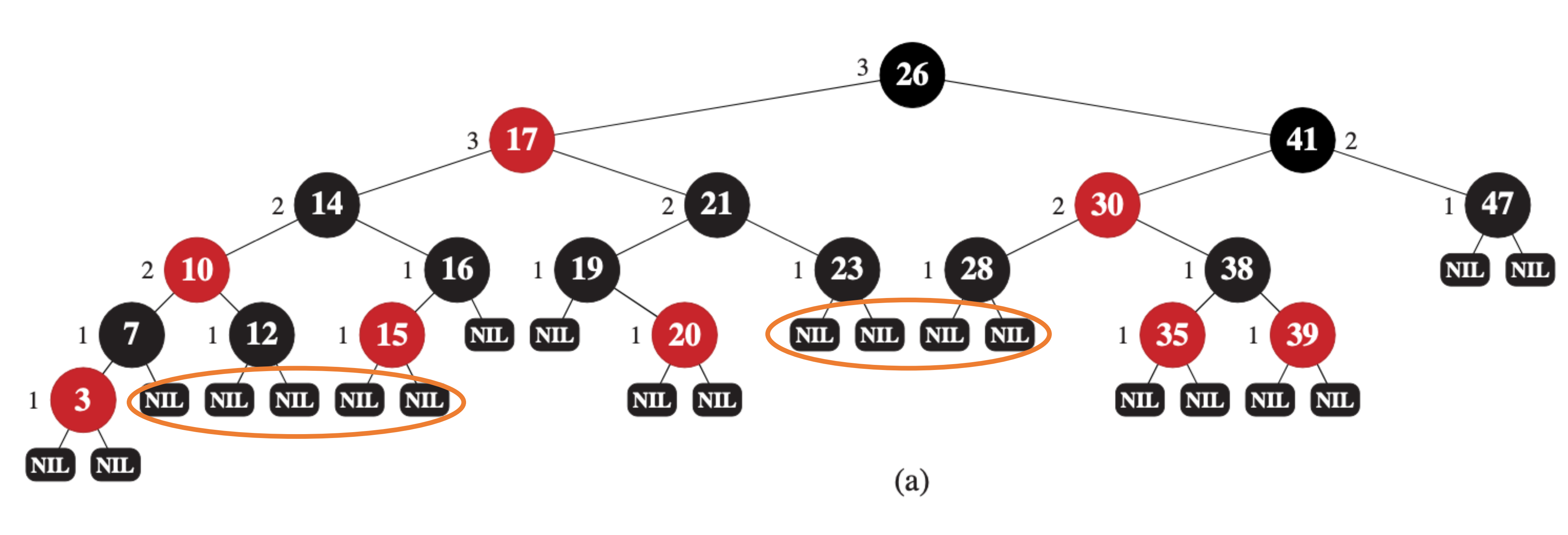
- Example
  - The root is black.



(a)

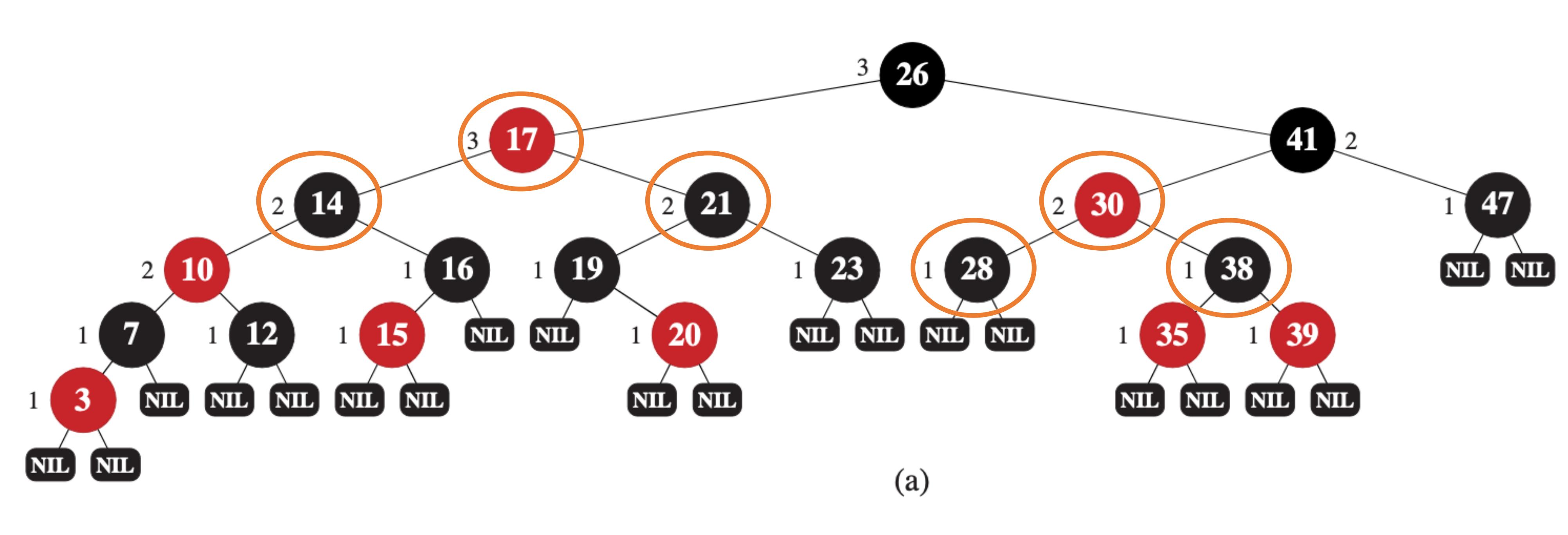
Red-Black Trees

- Example
  - Every leaf (NIL) is black.



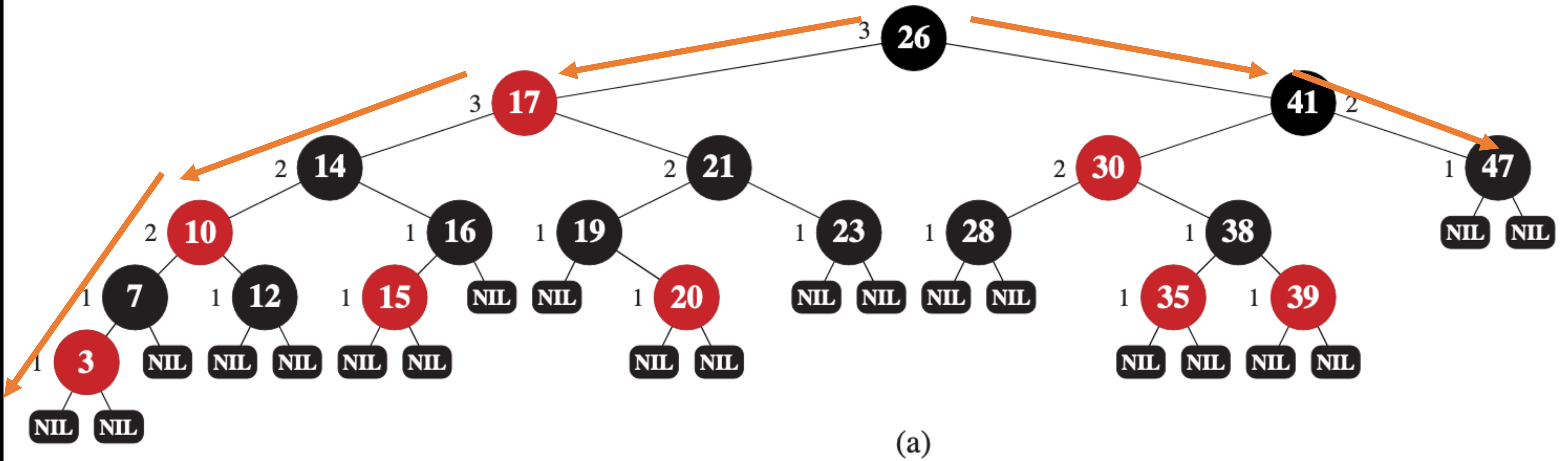
Red-Black Trees

- Example
  - If a node is red, then both its children are black.



Red-Black Trees

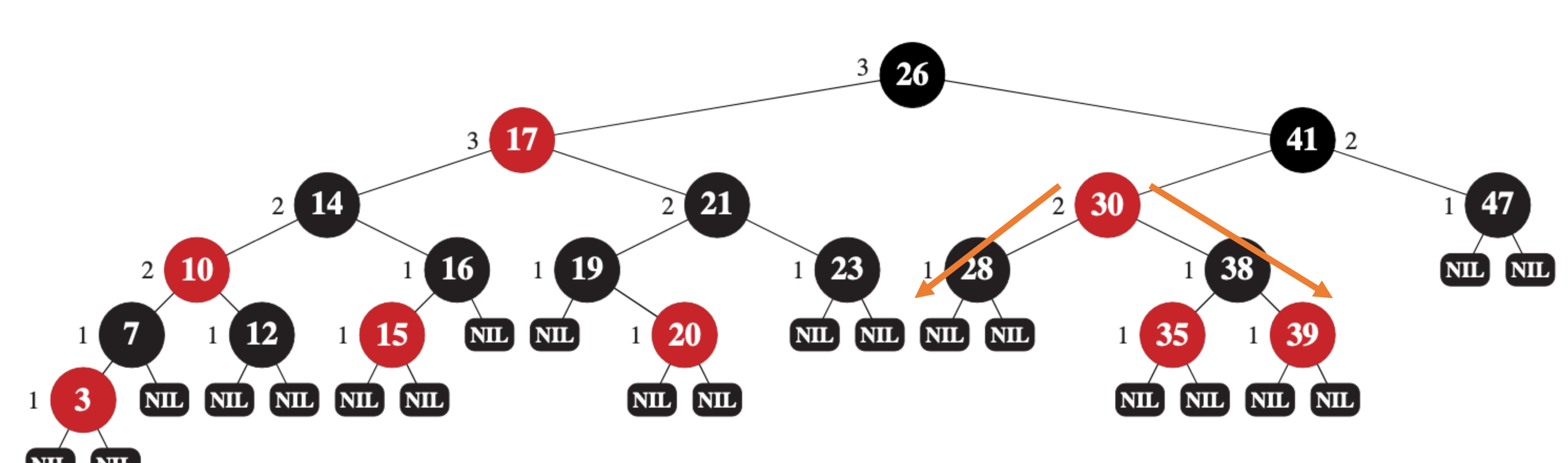
- Example
  - For each node, all simple paths from the node to descendant leaves contain the same number of black nodes.



(a)

Red-Black Trees

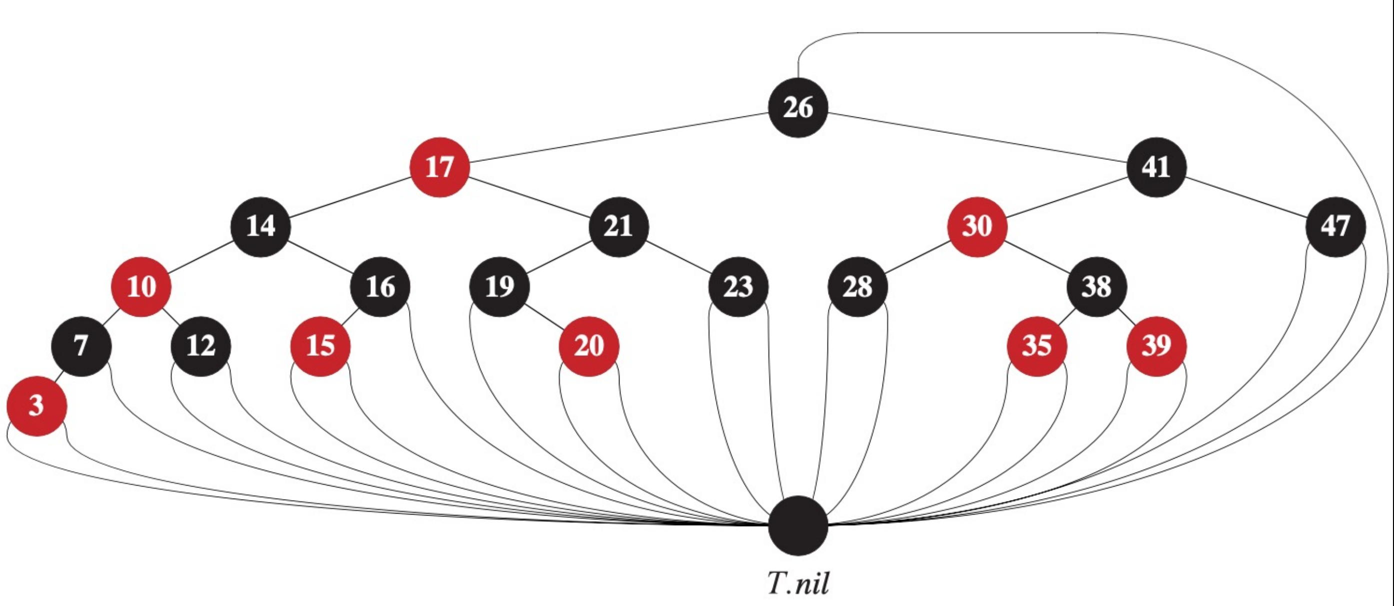
- Example
  - For each node, all simple paths from the node to descendant leaves contain the same number of black nodes.



(a)

Red-Black Trees

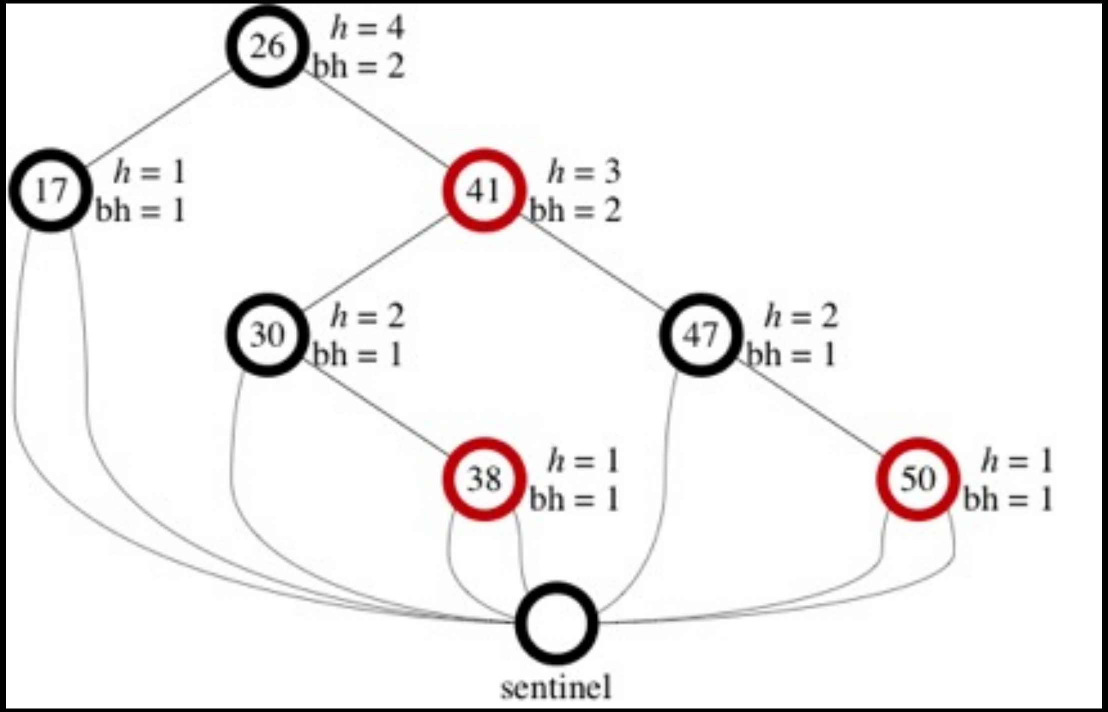
- Since all leaf nodes have their left and right are NILs, assign a sentinel node to all of them.
  - Sentinel node is black→ T.nil
  - The parent of the root point to the sentinel



(b)

Red-Black Trees

- $bh(x)$ refers to the black-height of the node x.
- Black-height is the number of black nodes on any path from node x to a leaf.
    - Node x is not counted

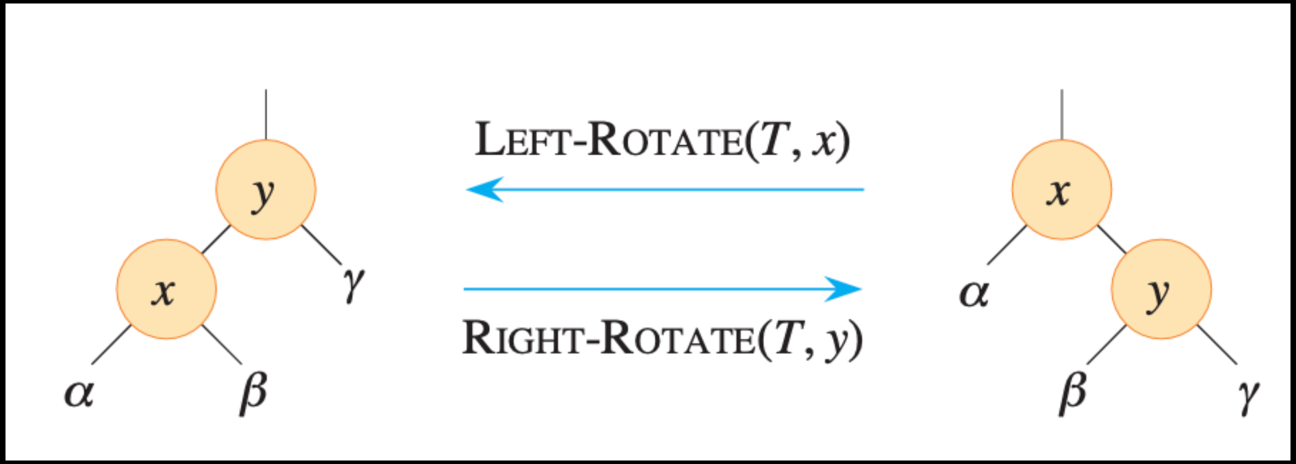


Content

Content
Red-Black Trees
Rotations
Insertion
Deletion
Complexity Classes

Rotations

- Rotation operation preserves the tree property when inserting or deleting elements.
- There are two rotations: left rotation and right rotation.
  - Both take $O(1)$, because they only change the pointers



Rotations

- Left rotation

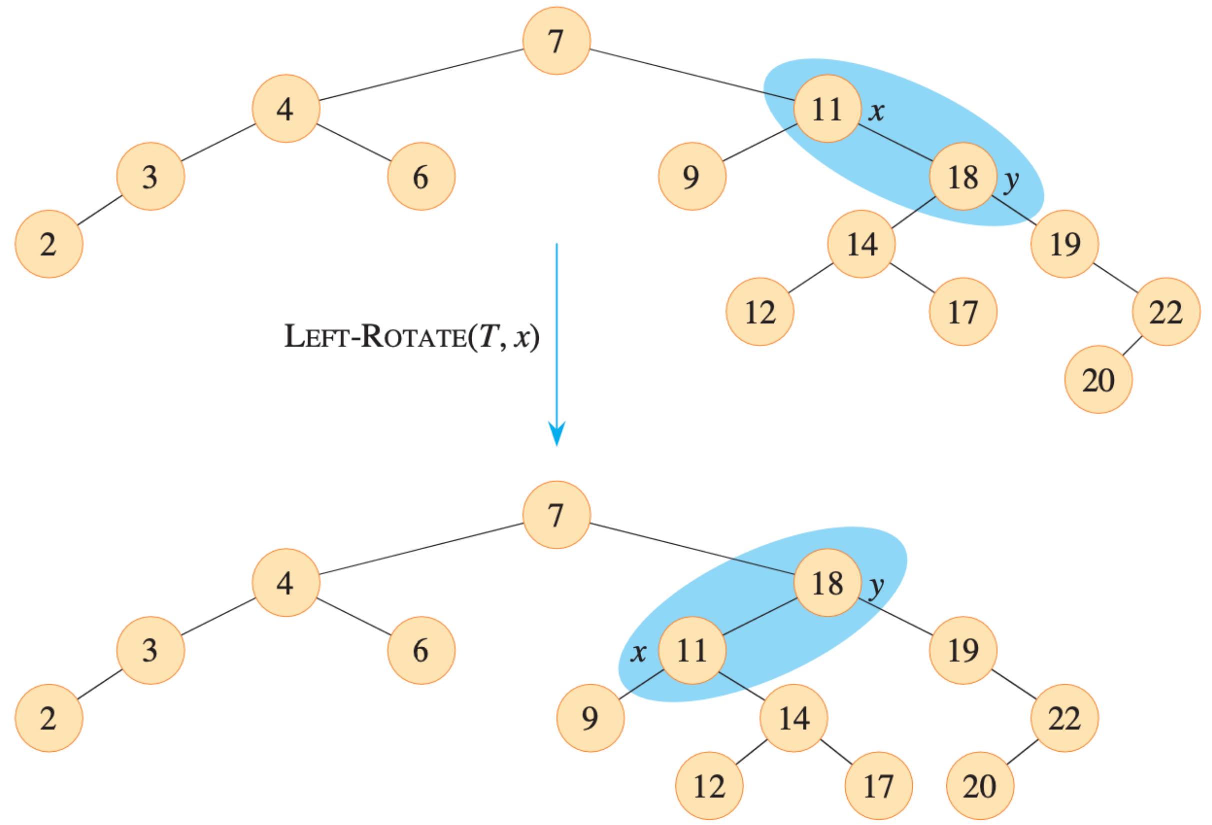
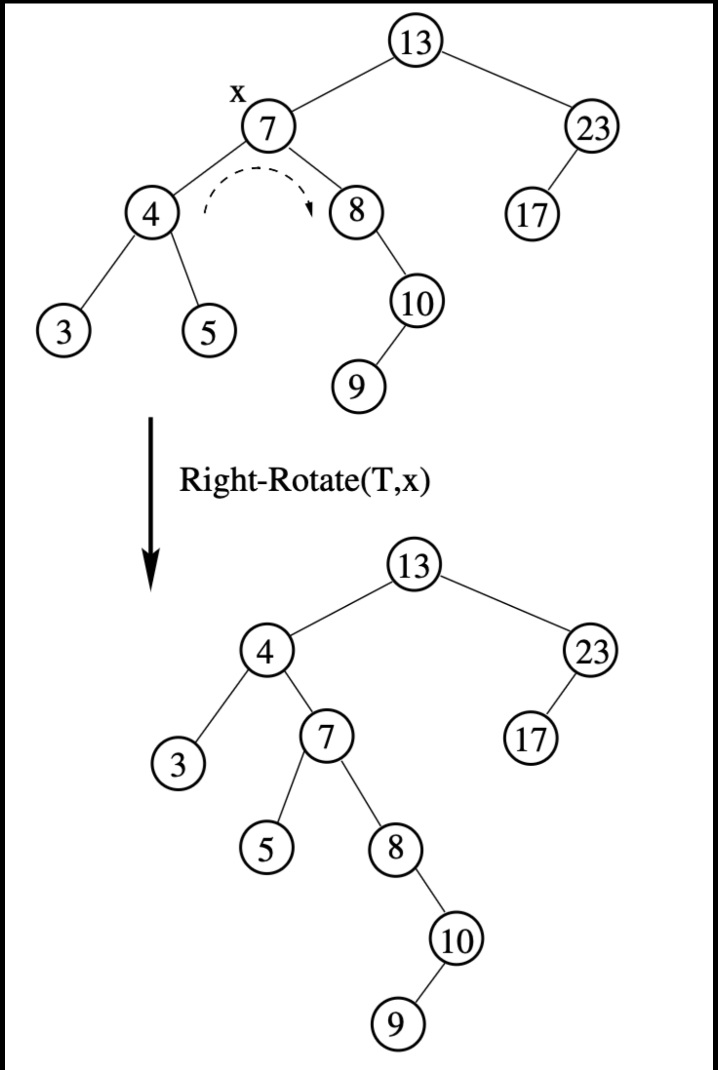


Figure 13.3 An example of how the procedure LEFT-ROTATE(T, x) modifies a binary search tree. Inorder tree walks of the input tree and the modified tree produce the same listing of key values.

Rotations

- Right rotation



Content

Content
Red-Black Trees
Rotations
Insertion
Deletion
Complexity Classes

Insertion

- To insert a node z in an RB-tree T:
    1. Set z's color to red
    2. BST-Insert(T, z)
    3. If the parent of z is black
        1. Stop
    4. If the parent of z is red
        1. Fix the tree

Insertion – Fix RB-tree

- While $z.p.color = red$:

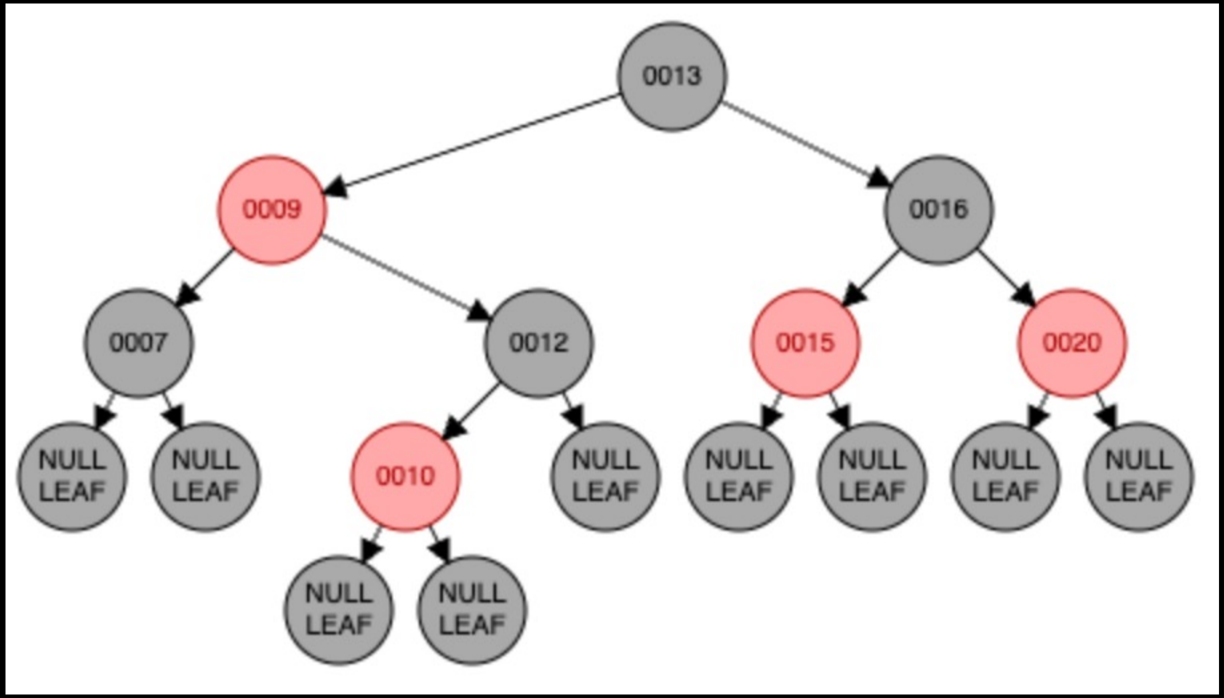
1. If $z.p$ is a left child
    1. y = right parent's sibling (uncle) of z.
    2. If y$.color = red$:
        1. $z.p.color = black$
        2. $y.color = black$
        3. $z.p.p.color = red$
    3. If $y.color = black$
        1. If z is a right child
            1. $z = z.p$
            2. LR(T, z)
        2. $z.p.color = black$
        3. $z.p.p = red$
        4. RR($T, z.p.p$)

2. If $z.p$ is a right child
    1. y = left parent's sibling (uncle) of z.
    2. If $y.color = red$:
        1. $z.p.color = black$
        2. $y.color = black$
        3. $z.p.p.color = red$
        4. $z = z.p.p$
    3. If $y.color = black$:
        1. If z is a left child
            1. $z = z.p$
            2. RR(T, z)
        2. $z.p.color = black$
        3. $z.p.p.color = red$
        4. LR($T, z.p.p$)

$T.root.color = black$

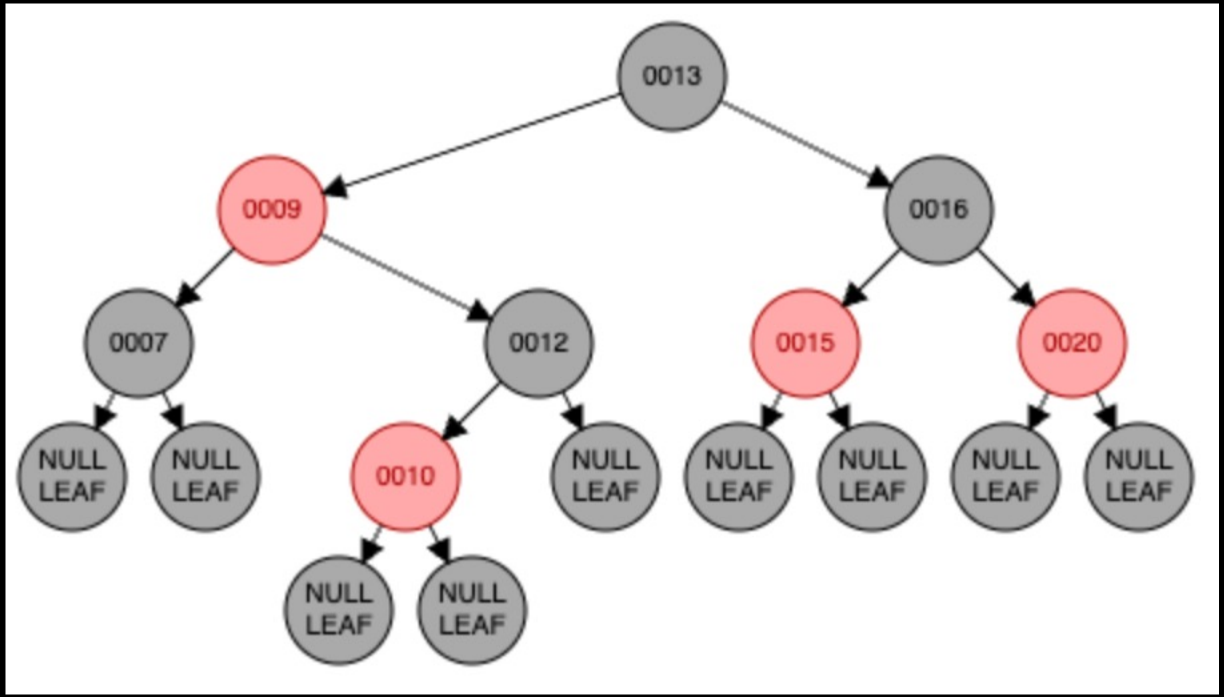
Insertion

- Example: insert 11



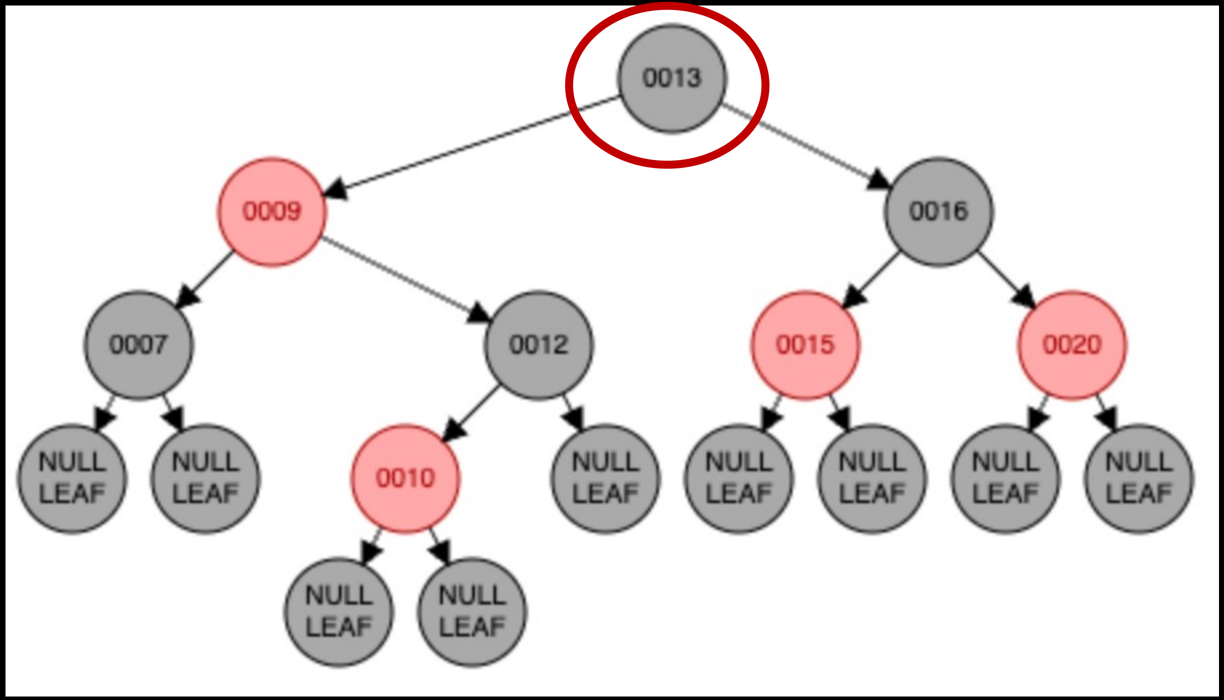
Insertion


Inserting 0011
0011


0013
0009
0016
0007
0012
0015
0020
NULL LEAF
NULL LEAF
0010
NULL LEAF
NULL LEAF
NULL LEAF
NULL LEAF
NULL LEAF
NULL LEAF
NULL LEAF

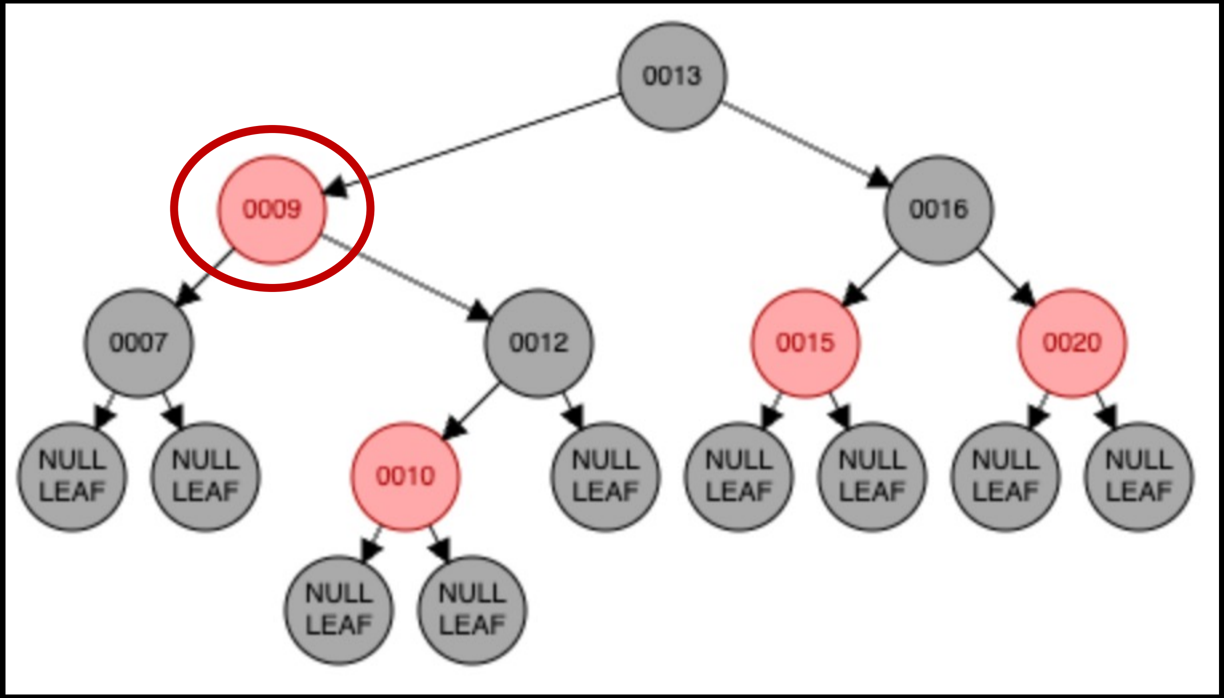
Insertion


Inserting 0011
0011


0013
0009
0016
0007
0012
0015
0020
NULL LEAF
NULL LEAF
0010
NULL LEAF
NULL LEAF
NULL LEAF
NULL LEAF
NULL LEAF
NULL LEAF
NULL LEAF

Insertion

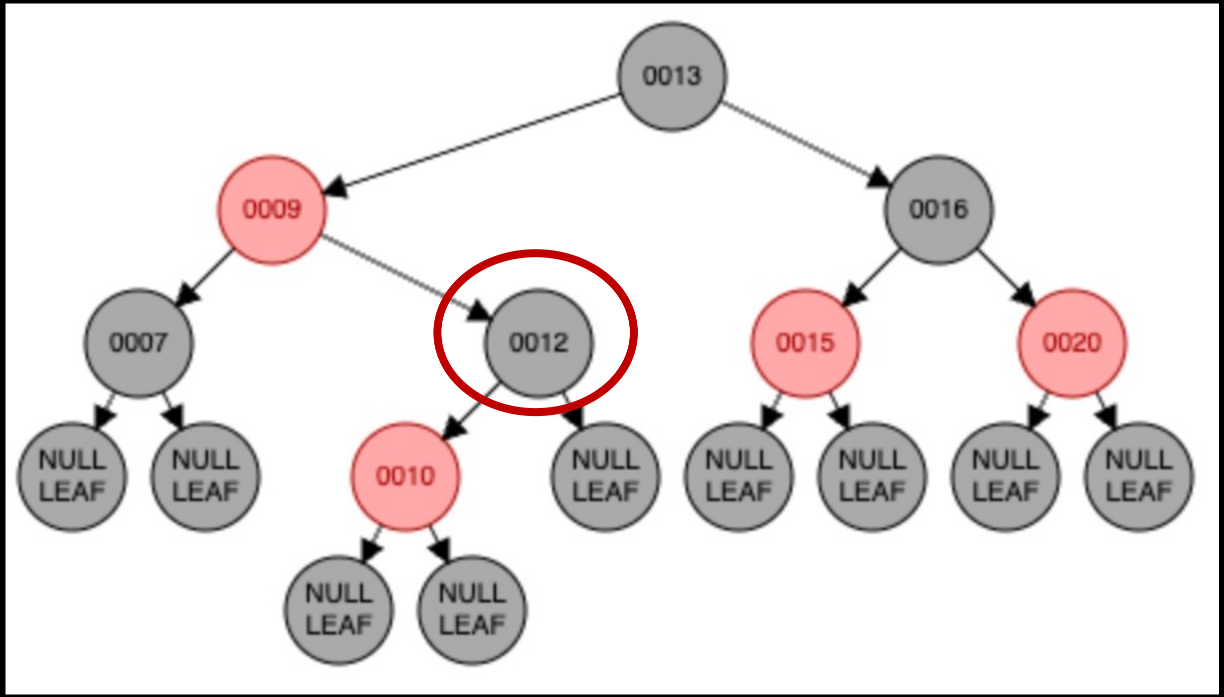
Inserting 0011
0011



0013
0009
0016
0007
0012
0015
0020
NULL LEAF
NULL LEAF
0010
NULL LEAF
NULL LEAF
NULL LEAF
NULL LEAF
NULL LEAF
NULL LEAF
NULL LEAF

Insertion

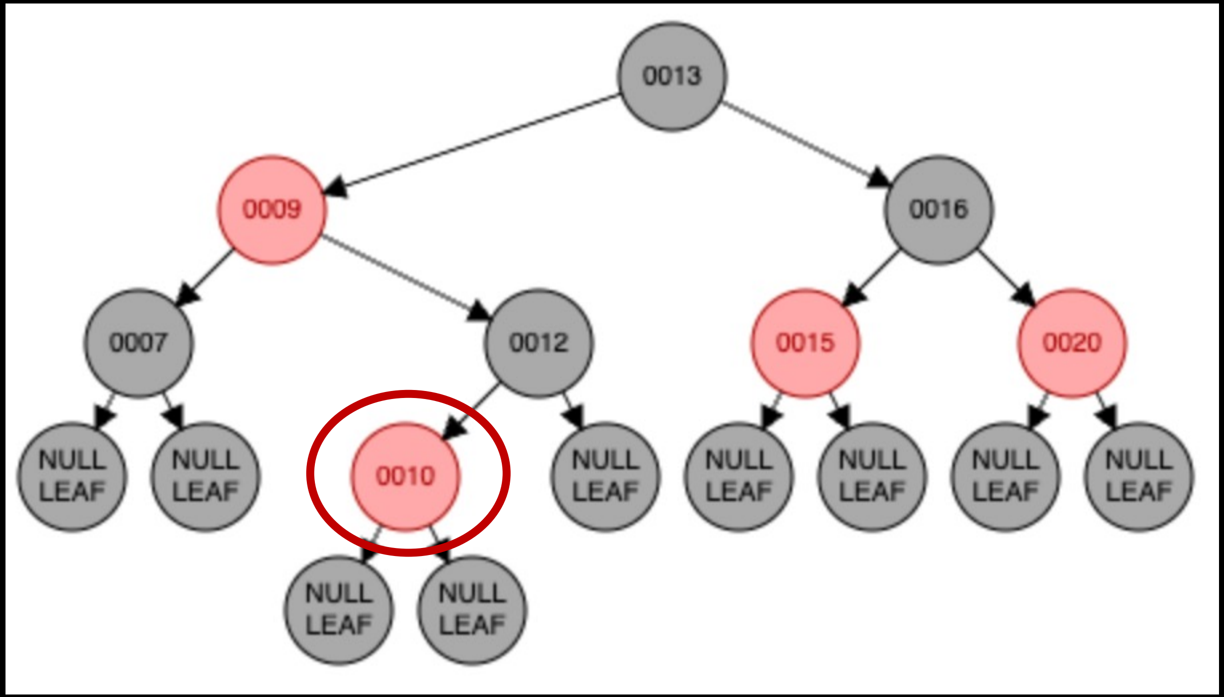
Inserting 0011
0011



0013
0009
0016
0007
0012
0015
0020
NULL LEAF
NULL LEAF
0010
NULL LEAF
NULL LEAF
NULL LEAF
NULL LEAF
NULL LEAF
NULL LEAF
NULL LEAF

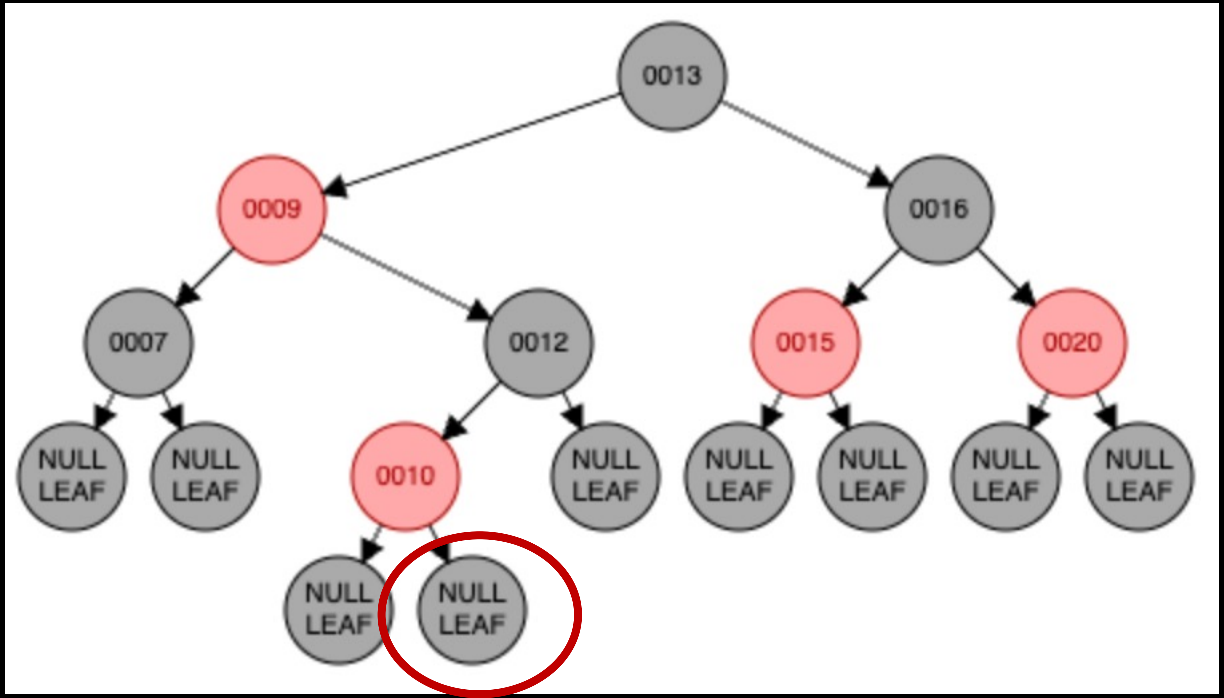
Insertion


Inserting 0011
0011


0013
0009
0016
0007
0012
0015
0020
NULL LEAF
NULL LEAF
0010
NULL LEAF
NULL LEAF
NULL LEAF
NULL LEAF
NULL LEAF
NULL LEAF
NULL LEAF

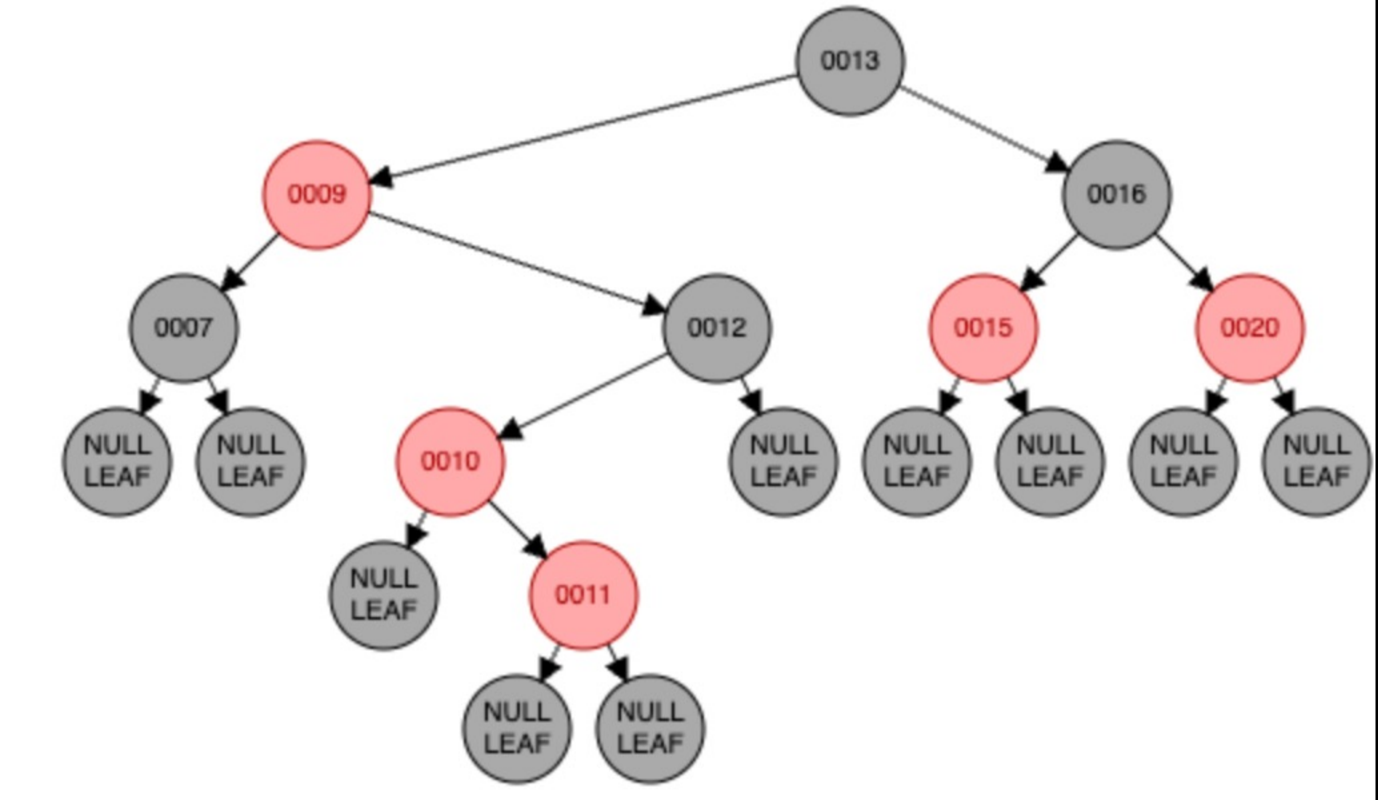
Insertion


Inserting 0011
0011


0013
0009
0016
0007
0012
0015
0020
NULL LEAF
NULL LEAF
0010
NULL LEAF
NULL LEAF
NULL LEAF
NULL LEAF
NULL LEAF
NULL LEAF
NULL LEAF

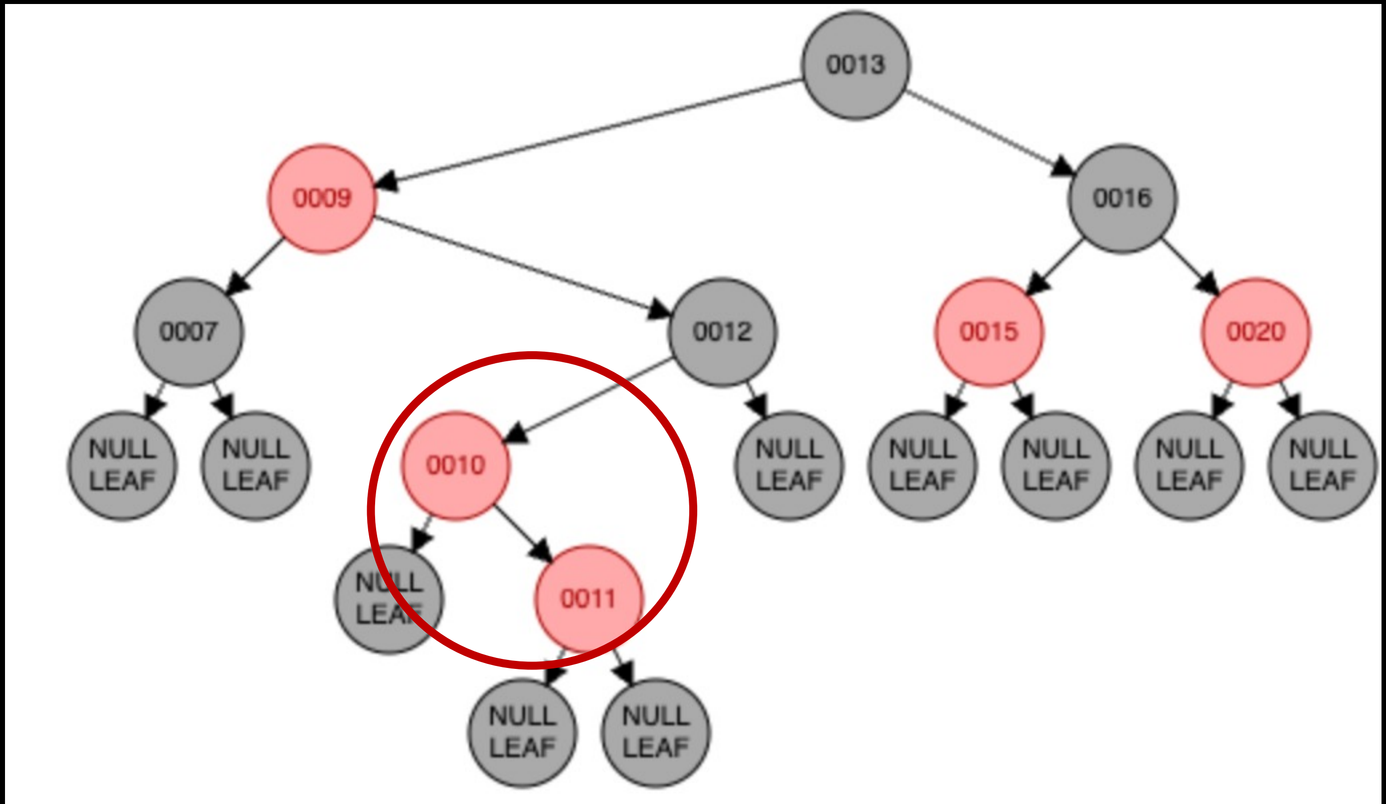
Insertion

- z and parent are both red. z is a right child, parent is a left child



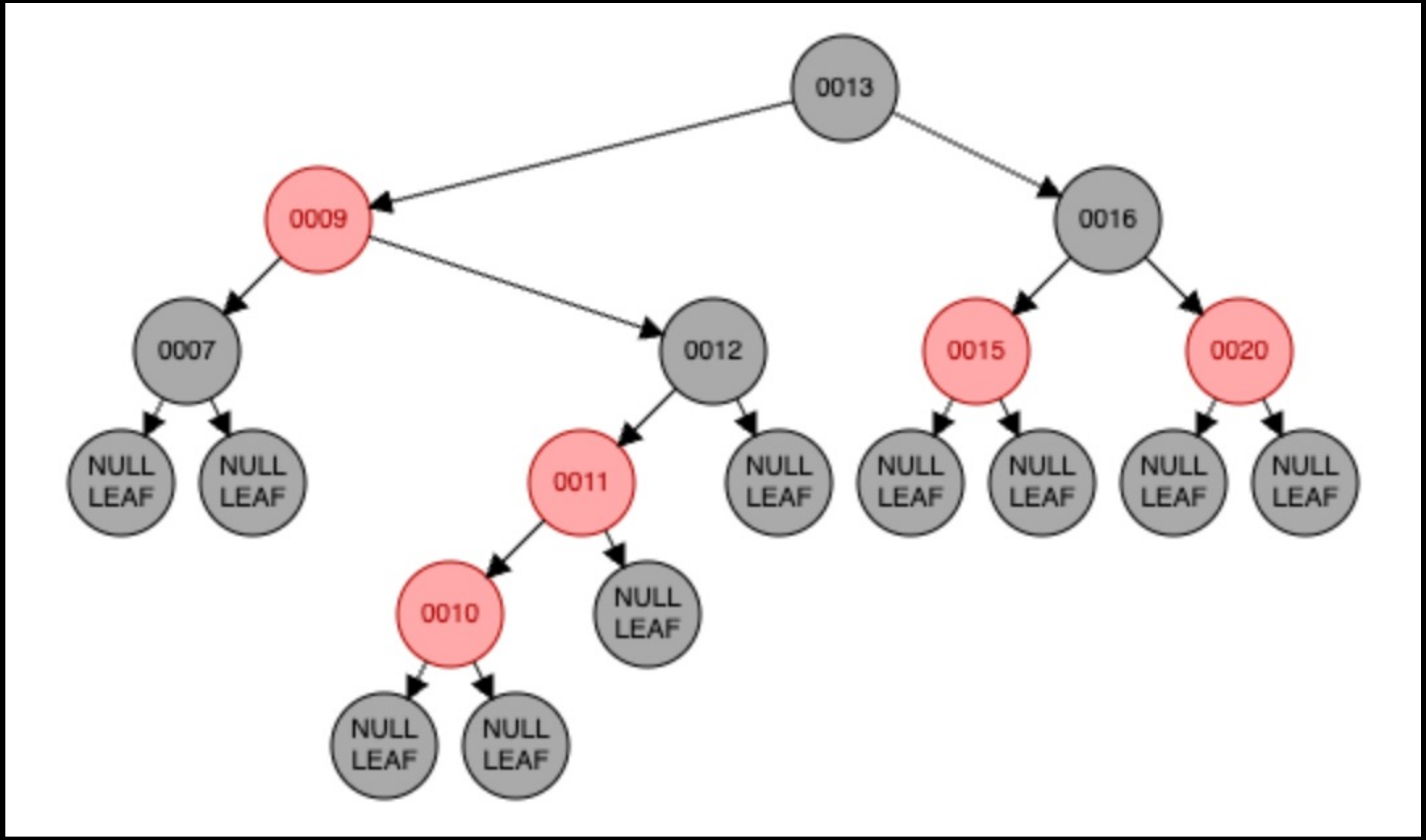
Insertion

- Single left rotation



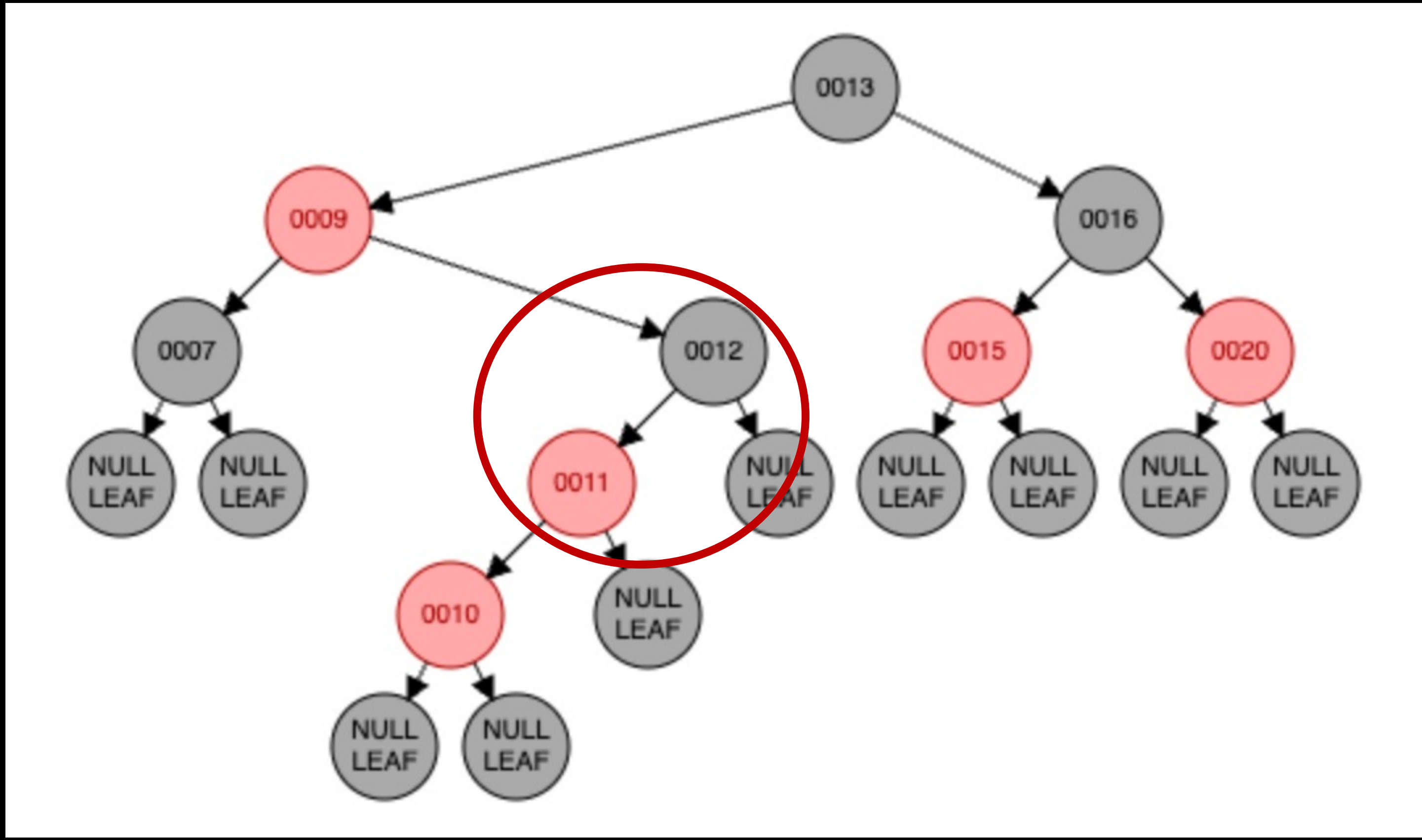
Insertion

- z and parent are red. z is left child, parent is left child



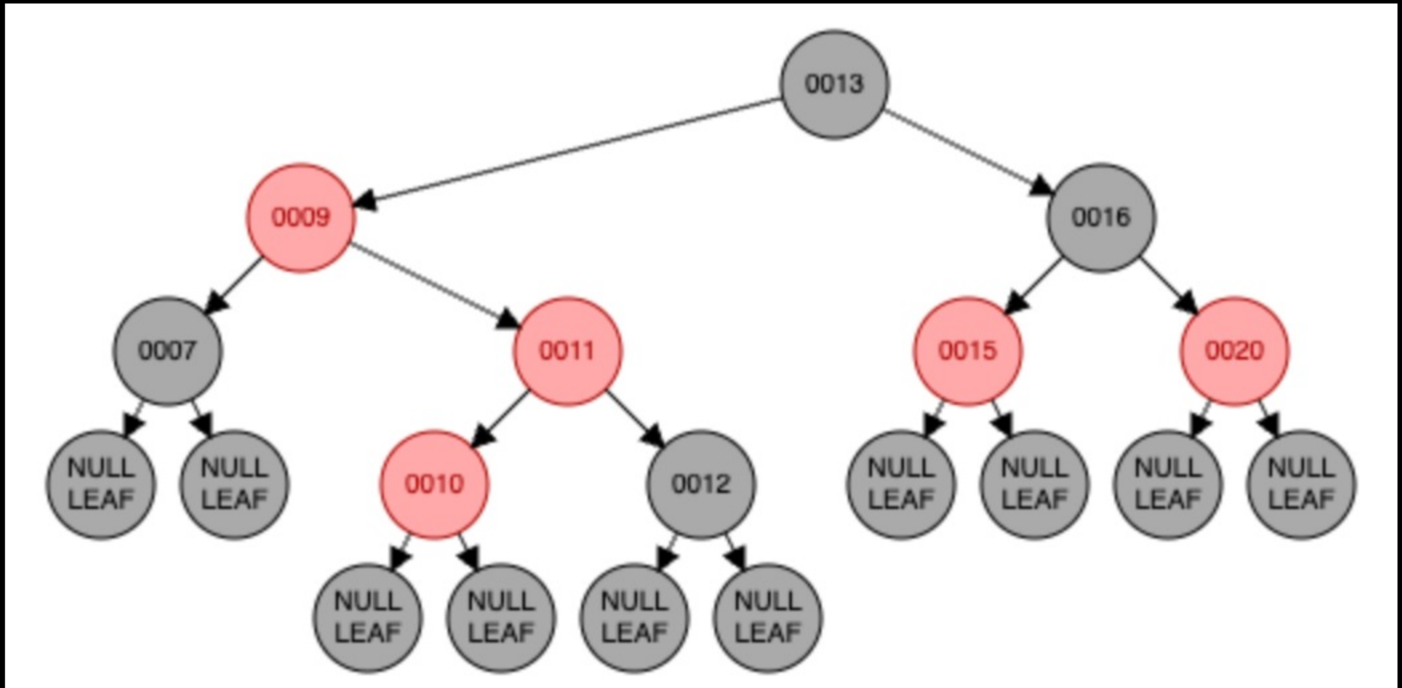
Insertion

- Single right rotation



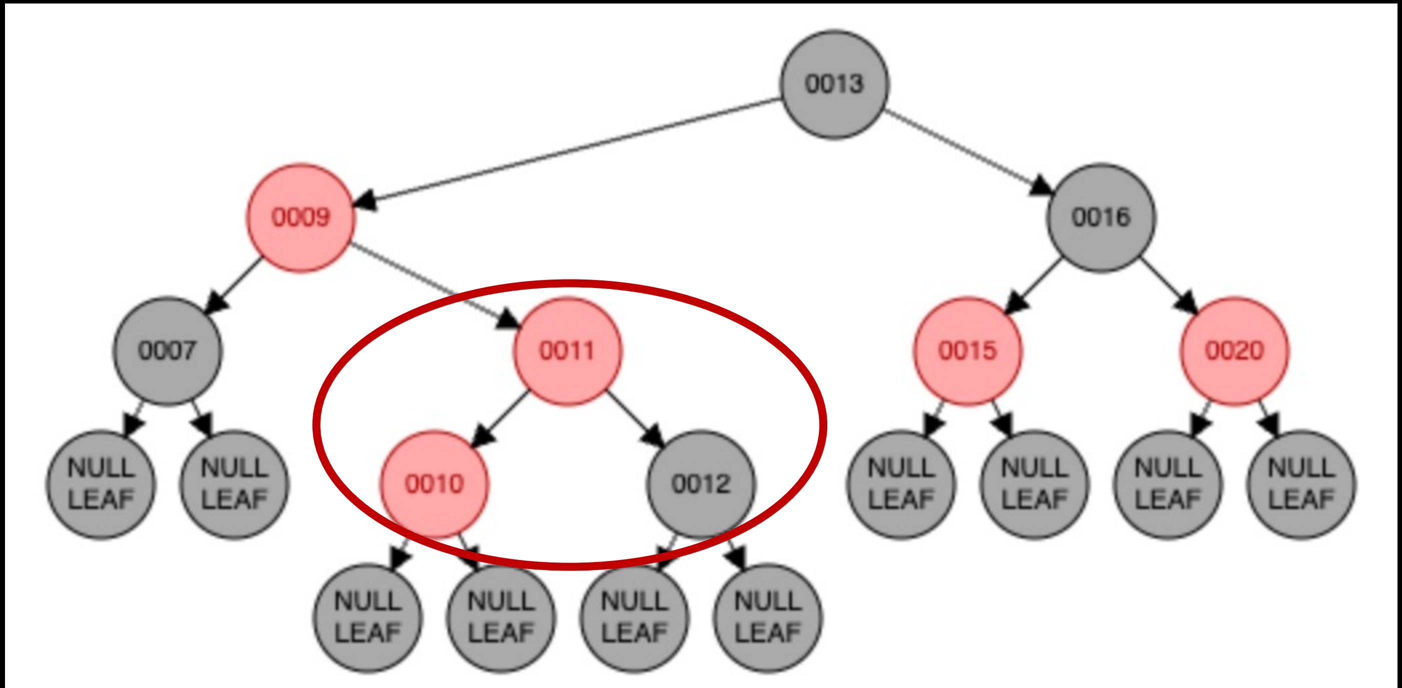
Insertion

- Single right rotation



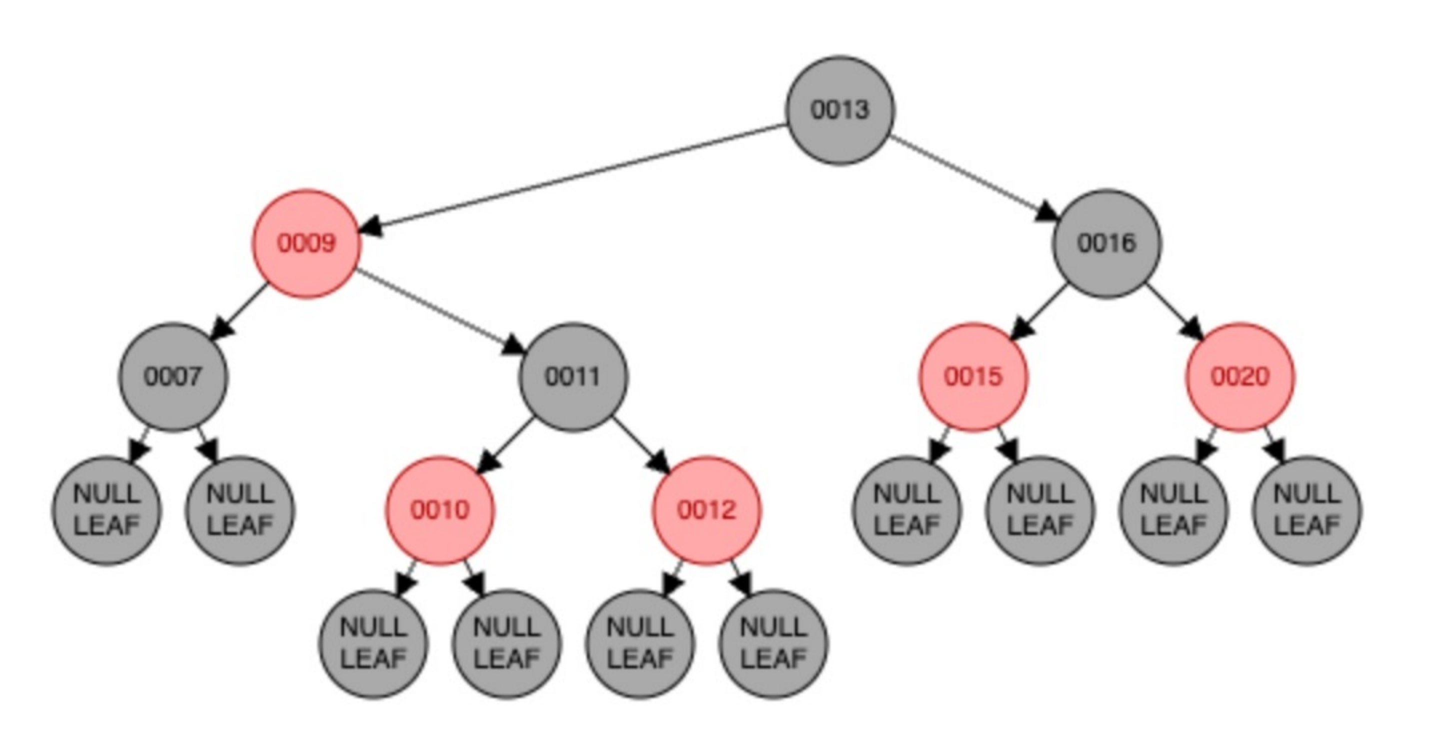
Insertion

- Recolor the nodes



Insertion

- Recolor the nodes



Insertion

- Insert the values [10, 1, 17, 4, 2, 0, 15] in an RB-tree.

Insertion

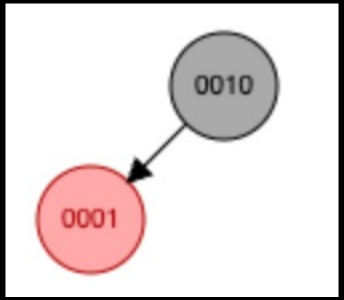
[10, 1, 17, 4, 2, 0, 15] – The tree is empty, so 10 is a black root node

Insertion

[10, 1, 17, 4, 2, 0, 15] – 1 is created as a red node.

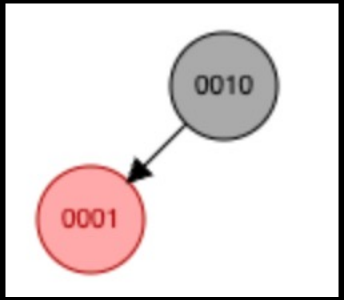
Insertion

[10, 1, 17, 4, 2, 0, 15] – 1 is less than 10, so it's inserted on the left of 10.



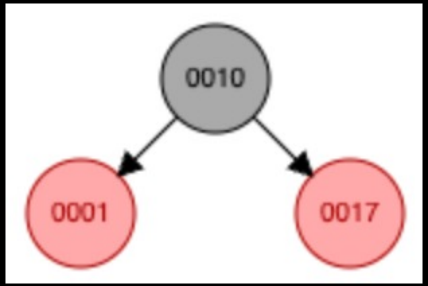
Insertion

[10, 1, 17, 4, 2, 0, 15] – 17 is created as a red node.



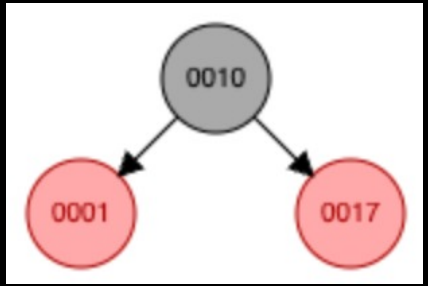
Insertion

[10, 1, 17, 4, 2, 0, 15] – 17 is greater than 10, so it’s inserted on the right.



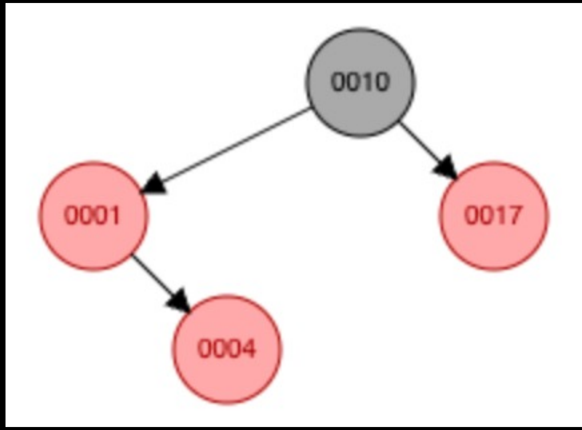
Insertion

[10, 1, 17, 4, 2, 0, 15] – 4 is created as a red node.



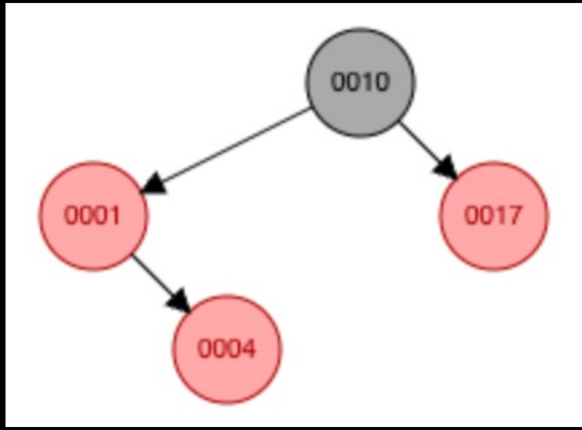
Insertion

[10, 1, 17, 4, 2, 0, 15] – 4 is less than 10 and greater than 1.



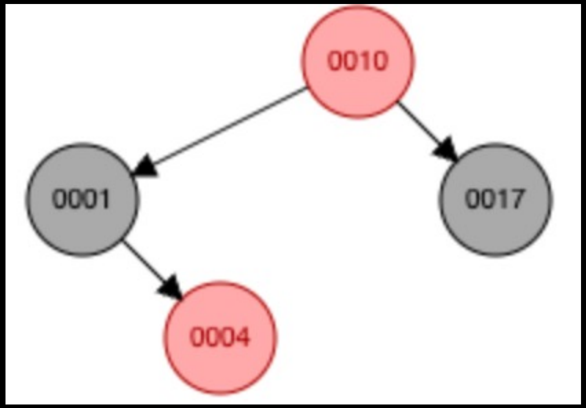
Insertion

[10, 1, 17, 4, 2, 0, 15] – 4 and 1 are both red. Uncle of 4 (17) is red.



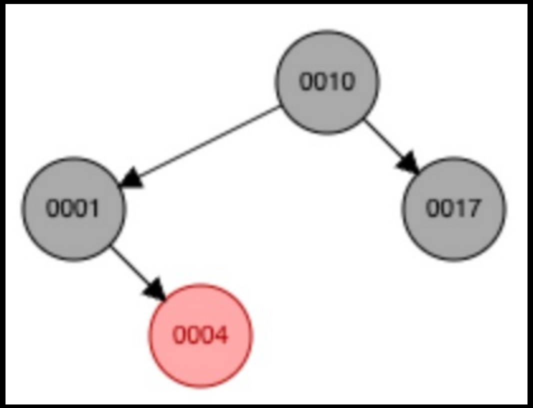
Insertion

[10, 1, 17, 4, 2, 0, 15] – Recolor the nodes from the grandparent.



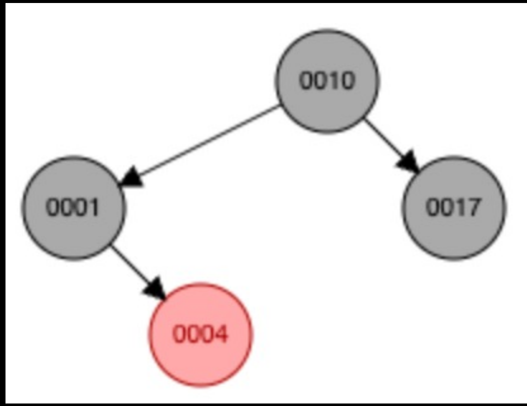
Insertion

[10, 1, 17, 4, 2, 0, 15] – Root is red, set it to black.



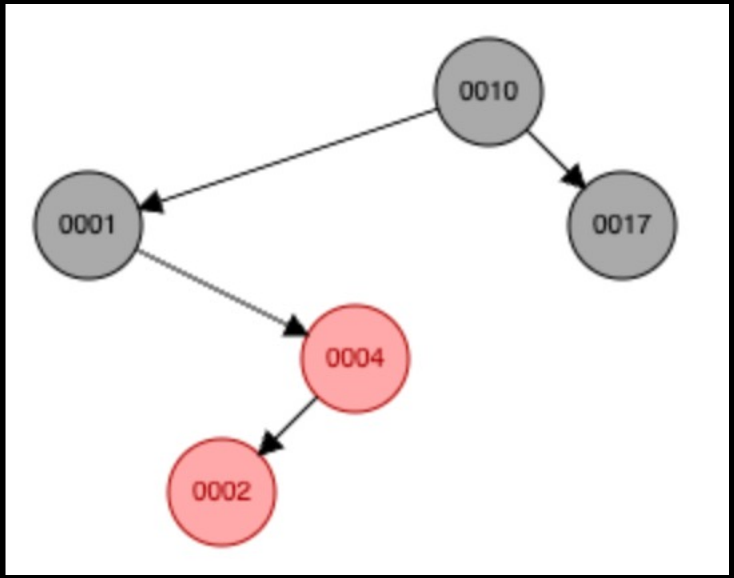
Insertion

[10, 1, 17, 4, 2, 0, 15] – 2 is created as a red node.



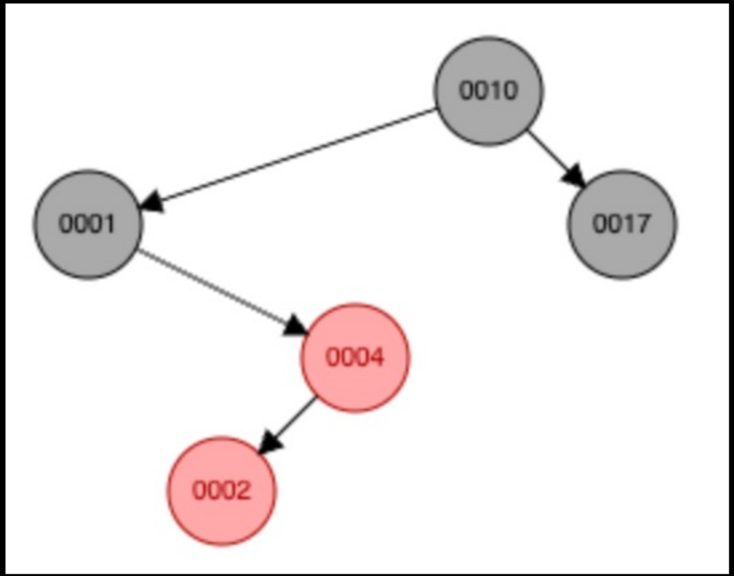
Insertion

[10, 1, 17, 4, 2, 0, 15] – 2<10, 2>1, 2<4.



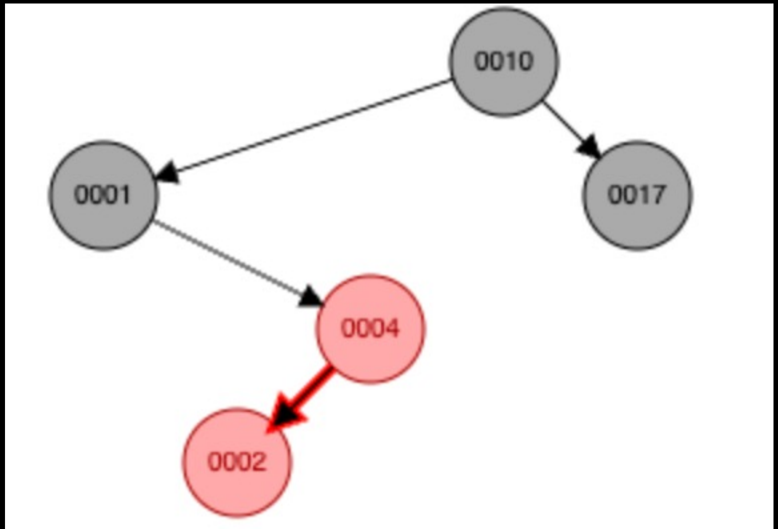
Insertion

[10, 1, 17, 4, 2, 0, 15] – 2 and 4 are both red.



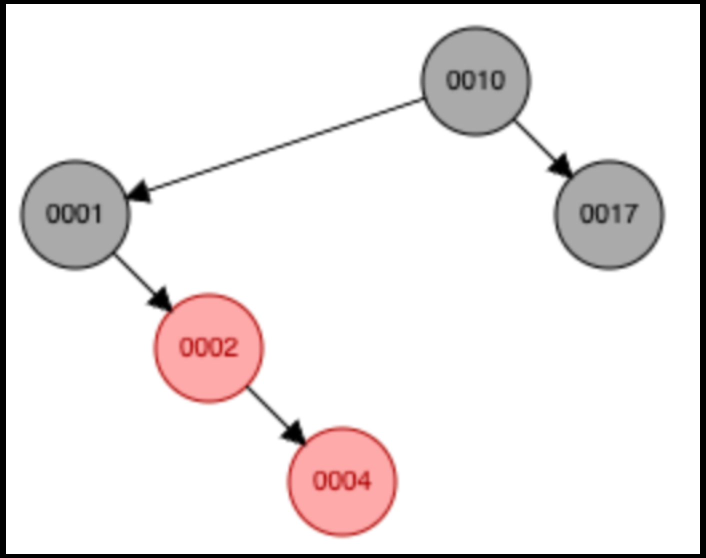
Insertion

[10, 1, 17, 4, 2, 0, 15] – 2 is a left child, 4 is a right child → RR



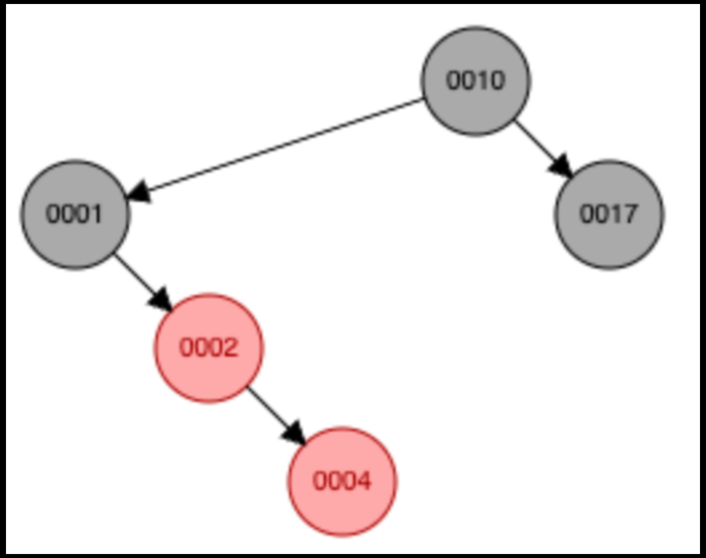
Insertion

[10, 1, 17, 4, 2, 0, 15] – Right rotate



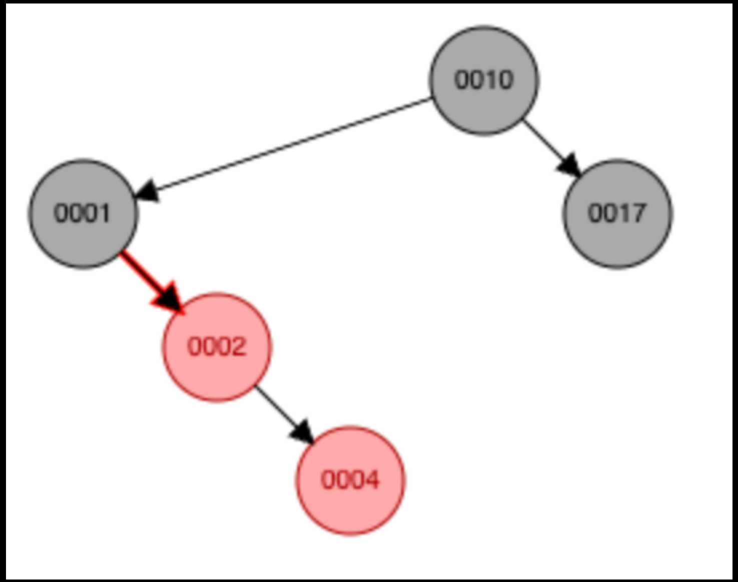
Insertion

[10, 1, 17, 4, 2, 0, 15] – 2 and 4 are red, they are both right child



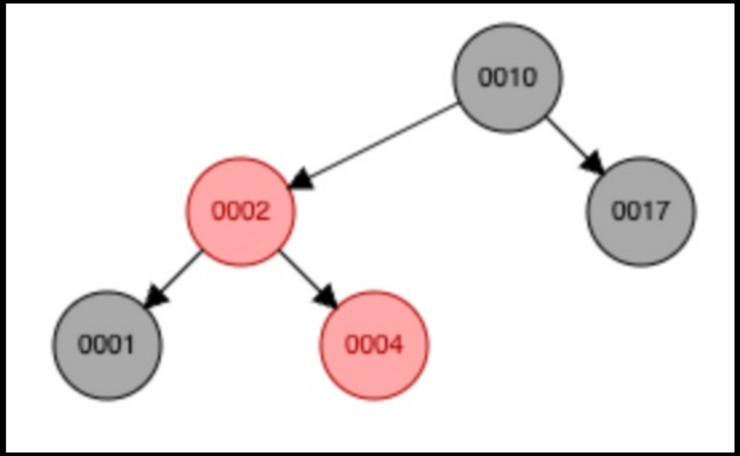
Insertion

[10, 1, 17, 4, 2, 0, 15] – 2 and 4 are red, they are both right child→ LR



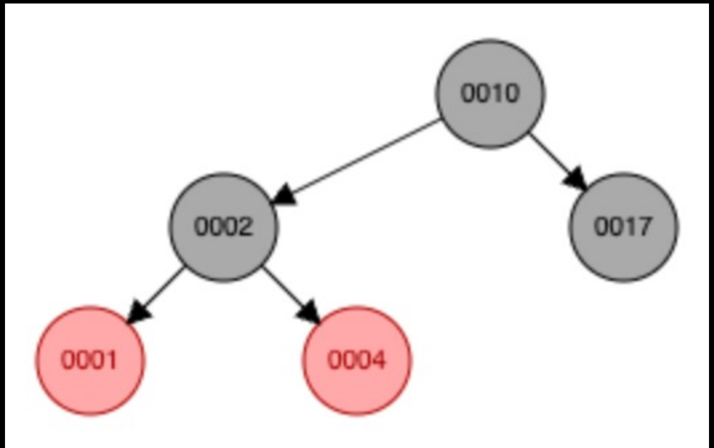
Insertion

[10, 1, 17, 4, 2, 0, 15] – Left rotate



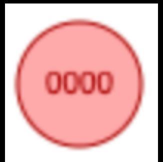
Insertion

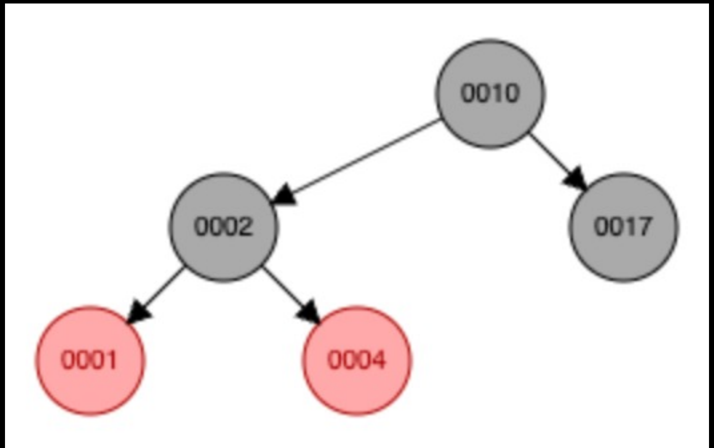
[10, 1, 17, 4, 2, 0, 15] – Recolor the nodes



Insertion

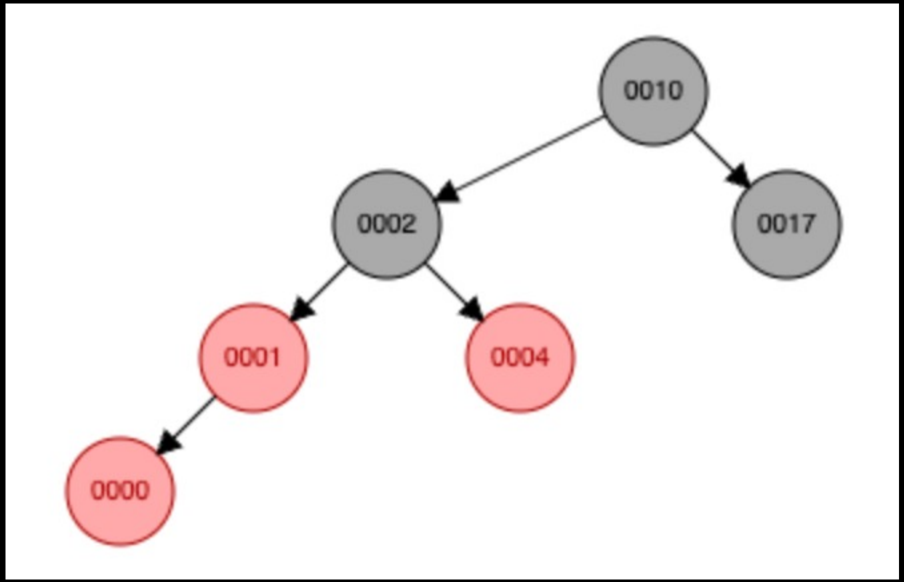
[10, 1, 17, 4, 2, 0, 15] – 0 is created as a red node





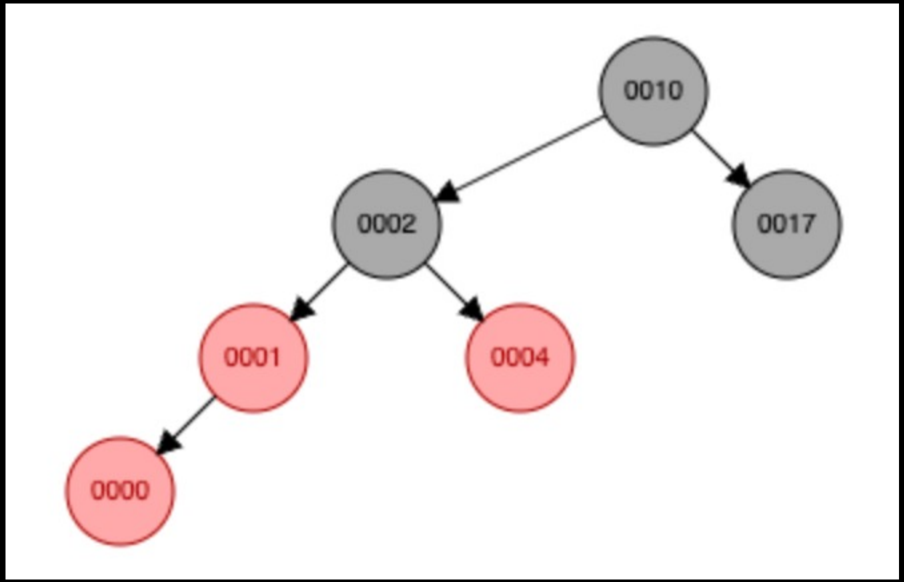
Insertion

[10, 1, 17, 4, 2, 0, 15] – 0<10, 0<2, 0<1



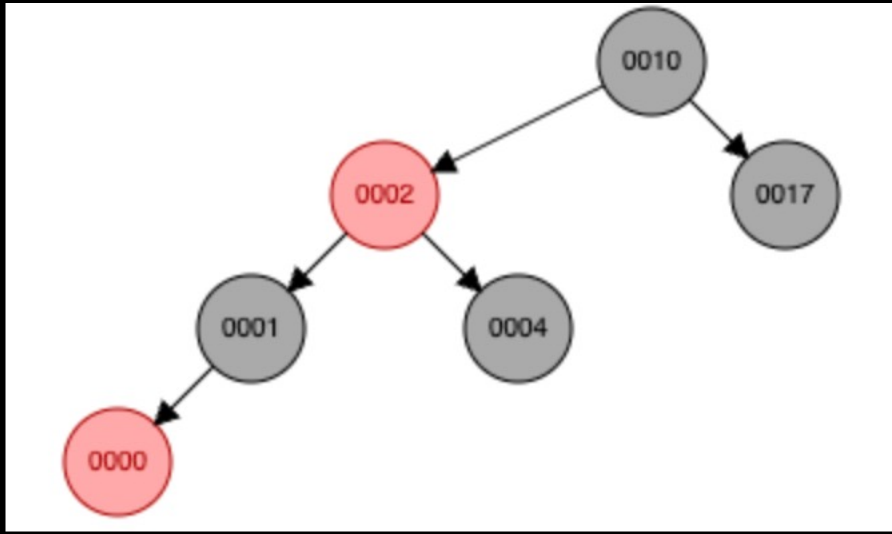
Insertion

[10, 1, 17, 4, 2, 0, 15] – 0 and 1 are both red, uncle (4) is red.



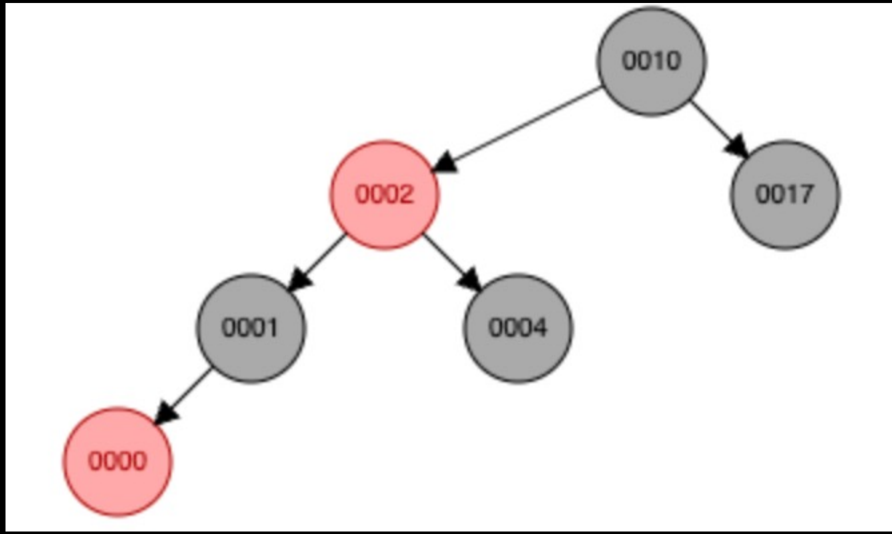
Insertion

[10, 1, 17, 4, 2, 0, 15] – Recolor from the grandparent



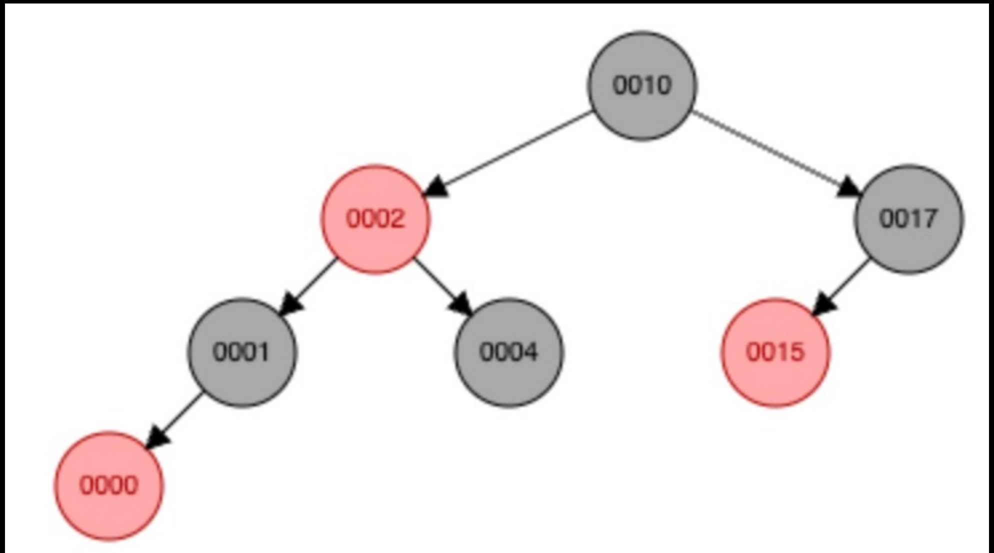
Insertion

[10, 1, 17, 4, 2, 0, 15] – 15 is created as a red node.



Insertion

[10, 1, 17, 4, 2, 0, 15] – 15>10, 15<17.



Content

Content
Red-Black Trees
Rotations
Insertion
Deletion
Complexity Classes

Deletion

- RB-delete(T, z)

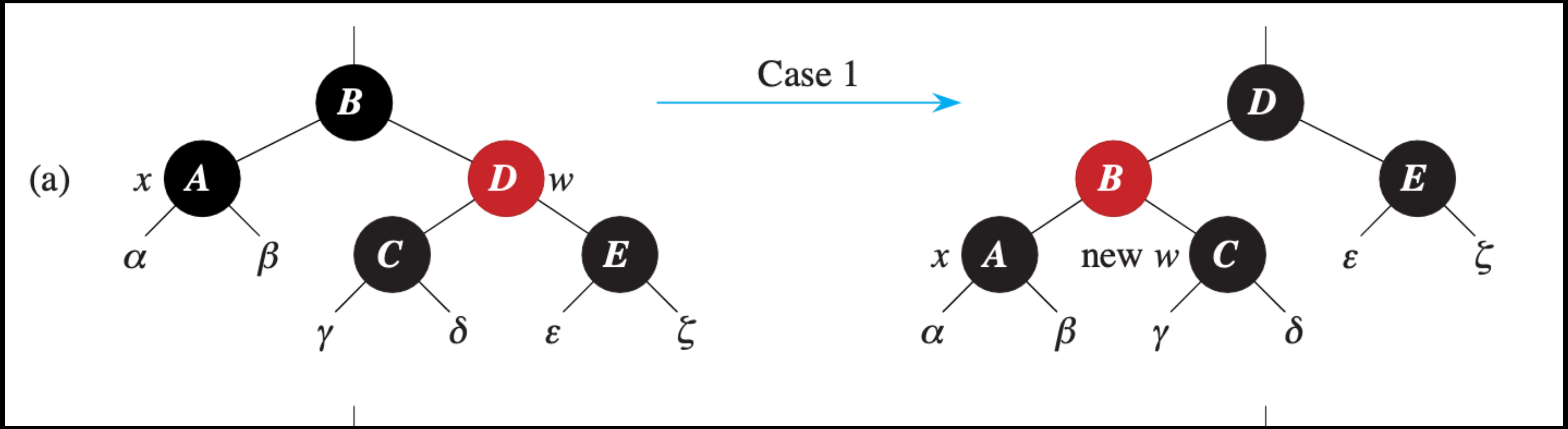
1. $y = z$
2. y-original-color = $y.color$
3. If $z.left = T.nil$:
   1. Replace z by its right child
4. Else If $z.right = T.nil$:
   1. Replace z by its left child

5. Else, $y = Tree - Minimum(z.right)$
   1. y-original-color = $y.color$
   2. $x = y.right$
   3. If $y \neq z.right$
      1. Replace y with its right child
   4. Else, $x.p = y$
   5. Replace z with y
   6. $y.color = z.color$
6. If y-original-color = black
   1. Fix the tree

Deletion

RB-DELETE-FIXUP(T,x)

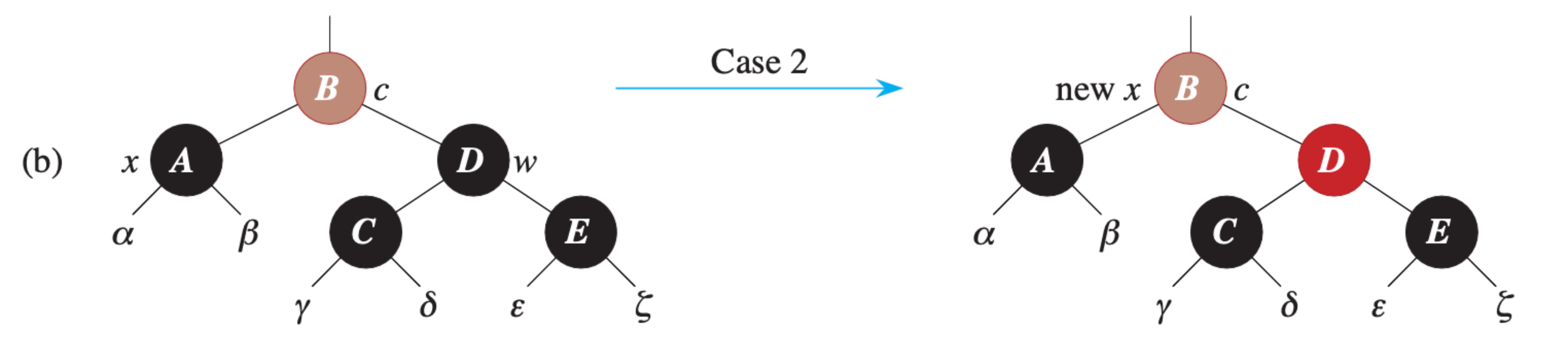
- While $x \neq T.root$ and $x.color = black$
  1. If x is a left child:
     1. $w =$ right sibling of x
     2. If $w.color = red$: → Case 1
        1. $w.color = black$
        2. $x.p.color = red$
        3. LR($T, x.p$)
        4. $w = x.p.right$



Deletion

RB-DELETE-FIXUP(T,x)

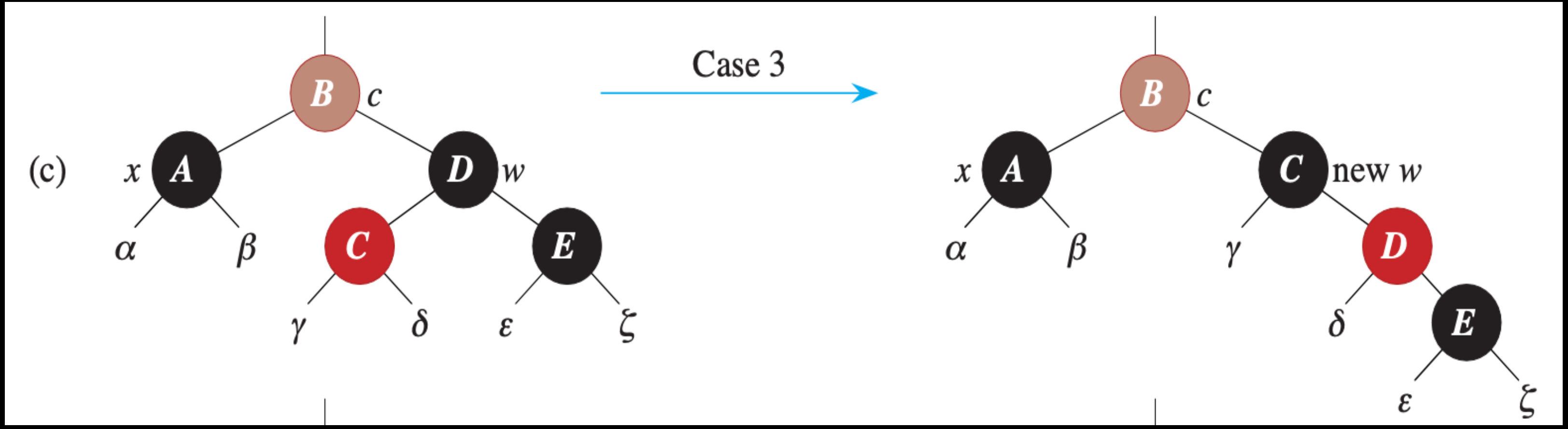
- While $x \neq T.root$ and $x.color = black$
  1. If x is a left child:
     1. $w =$ right sibling of x
     2. If $w.left.color = black$ and $w.right.color = black$: → Case 2
        1. $w.color = red$
        2. $x = x.p$



Deletion

RB-DELETE-FIXUP(T,x)

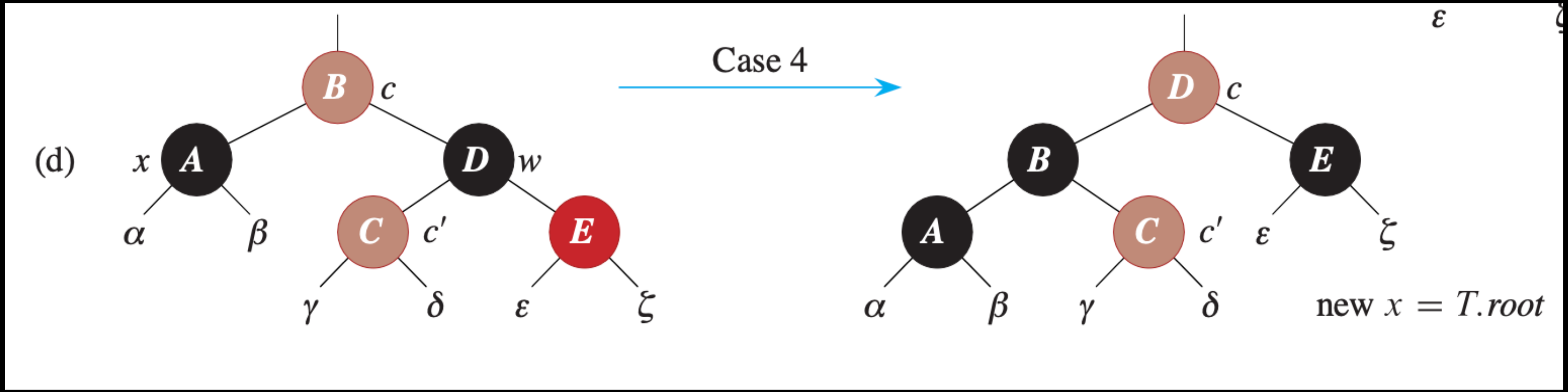
- While $x \neq T.root$ and $x.color = black$
  1. If x is a left child:
     1. $w =$ right sibling of x
     2. Else:
        1. If $w.color = black$: → Case 3
           1. $w.left.color = black$
           2. $w.color = red$
           3. RR(T, w)
           4. $w = x.p.right$



Deletion

RB-DELETE-FIXUP(T,x)

- While $x \neq T.root$ and $x.color = black$
  1. If x is a left child:
     1. w = right sibling of x
     2. Else: → Case 4
        1. $w.color = x.p.color$
        2. $x.p.color = black$
        3. $w.right.color = black$
        4. LR($T, x.p$)
        5. $x = T.root$



Deletion

RB-DELETE-FIXUP(T,x)

- While $x \neq T.root$ and $x.color = black$
  1. If x is a right child:
     1. $w = x.p.left$ → *Apply the previous 4 cases but exchanging right and left*
     2. If $w.color = red$: → Case 1
        1. $w.color = black$
        2. $x.p.color = red$
        3. RR($T, x.p$)
        4. $w = x.p.left$

Deletion

RB-DELETE-FIXUP(T,x)

- While $x \neq T.root$ and $x.color = black$
  1. If x is a right child:
     1. $w = x.p.left$ → *Apply the previous 4 cases but exchanging right and left*
     2. If $w.right.color = black$ and $w.left.color = black$: → Case 2
        1. $w.color = red$
        2. $x = x.p$

Deletion

RB-DELETE-FIXUP(T,x)

- While $x \neq T.root$ and $x.color = black$
  1. If x is a right child:
     1. $w = x.p.left$ → *Apply the previous 4 cases but exchanging right and left*
     2. Else:
        1. If $w.color = black$: → Case 3
           1. $w.right.color = black$
           2. $w.color = red$
           3. LR(T, w)
           4. $w = x.p.left$

Deletion

RB-DELETE-FIXUP(T,x)

- While $x \neq T.root$ and $x.color = black$
  1. If x is a right child:
     1. $w = x.p.left$ → *Apply the previous 4 cases but exchanging right and left*
     2. Else: → Case 4
        1. $w.color = x.p.color$
        2. $x.p.color = black$
        3. $w.left.color = black$
        4. RR($T, x.p$)
        5. $x = T.root$

Deletion

RB-DELETE-FIXUP(T,x)

- While $x \neq T.root$ and $x.color = black$
  - $\ldots$
- $x.color = black$

Content

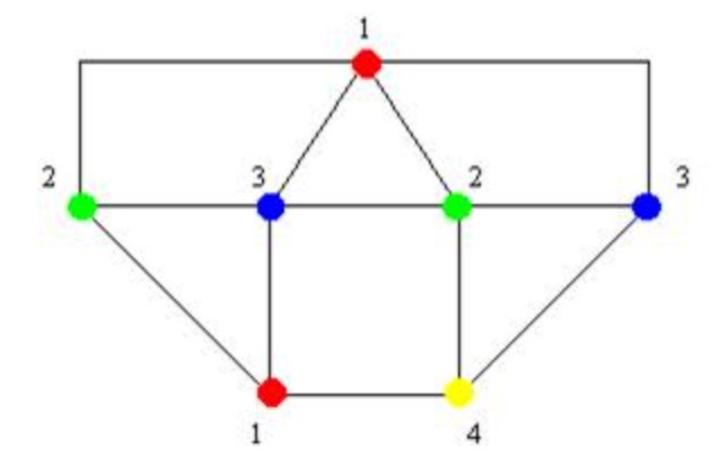
Content
Red-Black Trees
Rotations
Insertion
Deletion
Complexity Classes

Complexity Classes

- **Types of problems**
  - **Tractable**: Problems that can be solved in polynomial time
  - **Intractable**: Problems that as they grow large, we are unable to solve them in reasonable time
  - **Decision**: A problem with a True/False answer
  - **Optimization**: A problem which asks, “What is the optimal solution to problem X?”

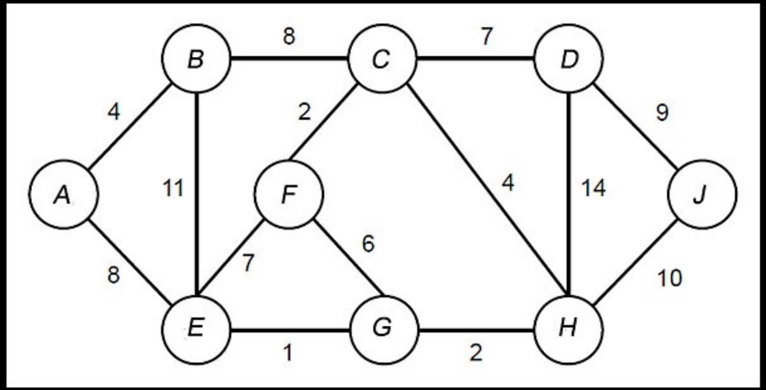
Complexity Classes

- Reasonable time = Polynomial time
  - On an input size n, the worst-case running time is $O(n^c)$ for some constant c.
  - Polynomial time: $O(n^2), O(n^3), O(1), O(n \lg n)$
  - Non-polynomial time: $O(2^n), O(n^n), O(n!)$
- Intractable problems: there is no efficient algorithm to solve.
  - Graph-coloring: coloring vertices such that no two adjacent vertices have the same color.



Complexity Classes

- Optimization problems: what is the best (optimal) solution?
  - Minimum spanning tree (MST): a spanning tree that has the minimum weight among all the possible spanning trees.

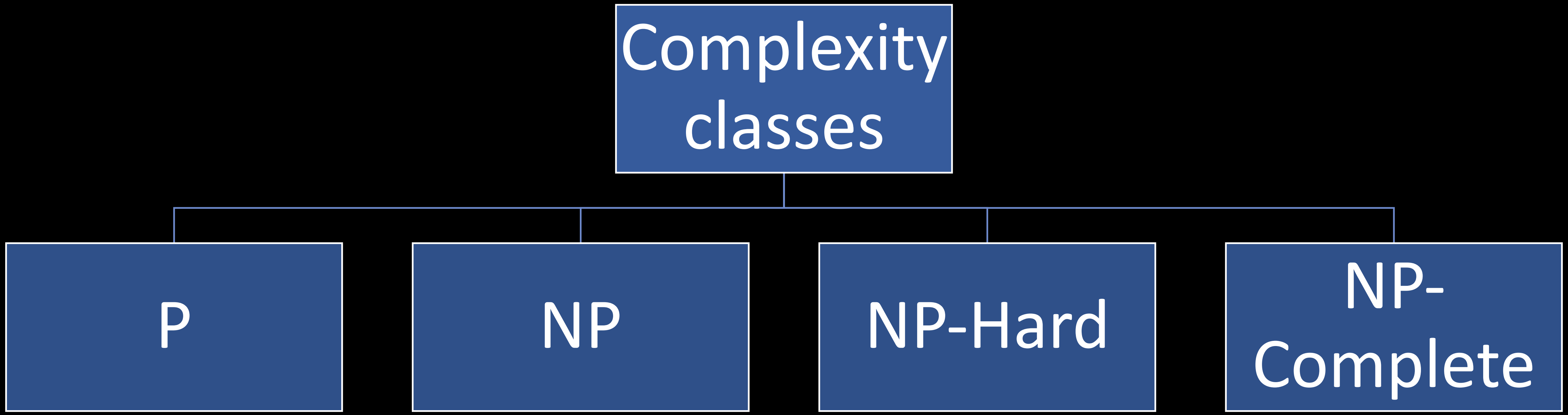


- Decision problems: Yes, or No?
  - Does a graph have an MST of weight $\leq w$?

Complexity Classes

- **Deterministic algorithms**: always compute the same answer and do the same sequence of computations.
  - **Predictability**: The same input will always result in the same output.
  - **Fixed Steps**: The algorithm follows a clear set of rules or steps.
  - **No Randomness**: There is no element of randomness or probability in the process.
- **Non-deterministic algorithms**: algorithms that “guess” the right solution.
  - **Multiple outcomes**: The same input might result in different outputs on different runs.
  - **Probabilistic Elements**: Often involves randomness or probability in decision-making.
  - **Parallel Path Exploration**: Can explore multiple solution paths simultaneously.

Complexity Classes



Complexity Classes

- **P (Polynomial Time)**: Problems **solvable** in polynomial time on deterministic algorithms.
    - Sorting algorithms.

- **NP (Non-deterministic Polynomial Time)**: Problems **verifiable** in polynomial time, solved using non-deterministic algorithms.
    - Hard to find an optimal solution.
    - Solutions can be verified in polynomial time.
    - Example: graph-coloring algorithms.

Complexity Classes

- Sometimes you can solve a problem by reducing it to a different problem. E.g., solving Problem B by solving Problem A.
  - If I can solve A in polynomial time, then I can construct a solution to B in polynomial time that is based on the solution of A.

- A problem is **NP-hard** if problems in NP are reducible to it.
  - Ex: Hamiltonian Cycle – a path in a graph that visits each vertex exactly once.
- Example: Travelling salesman problem can be reduced to Hamiltonian Cycle.

Complexity Classes

- **NP-Complete**: A problem that is NP-hard and NP.
- Open question: $P = NP$? → Is every problem whose solution can be verified quickly (in polynomial time) also solvable quickly (in polynomial time)?
- If $P = NP$, then all NP-complete problems would have efficient (polynomial-time) solutions
  - This has not been proven.
- Most computer scientists believe that $P \neq NP$.
  - This has not been proven.

Complexity Classes

- **P:** The easiest and can be solved efficiently.
- **NP:** Have a solution that can be verified quickly but finding it may be hard.
- **NP-complete:** The hardest problems in NP, and if one of them can be solved efficiently, so can all NP problems.
- **NP-hard** problems are at least as difficult as NP-complete problems, but they may not be decision problems and may not be in NP at all.