BITS Pilani
Pilani|Dubai|Goa|Hyderabad

Data Structures and Algorithms
CS F211

Vishal Gupta
Department of Computer Science and Information Systems
Birla Institute of Technology and Science
Pilani Campus, Pilani

Agenda: Red-Black Trees

Red-Black Trees

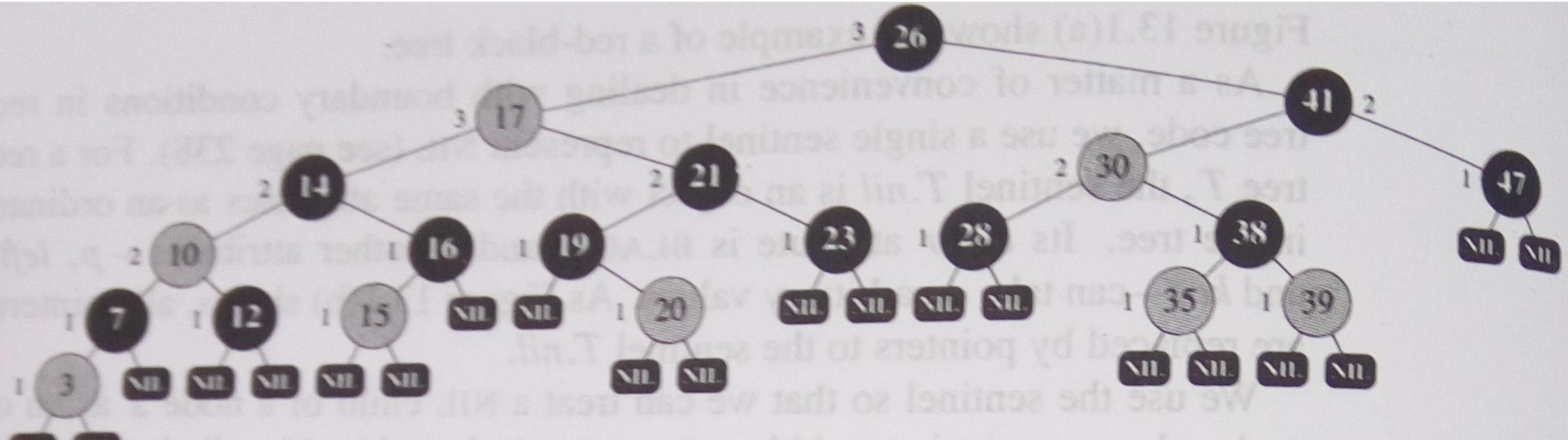
- *Red-black trees*:
  - Binary search trees augmented with node color
  - Operations designed to guarantee that the height $h = O(\lg n)$

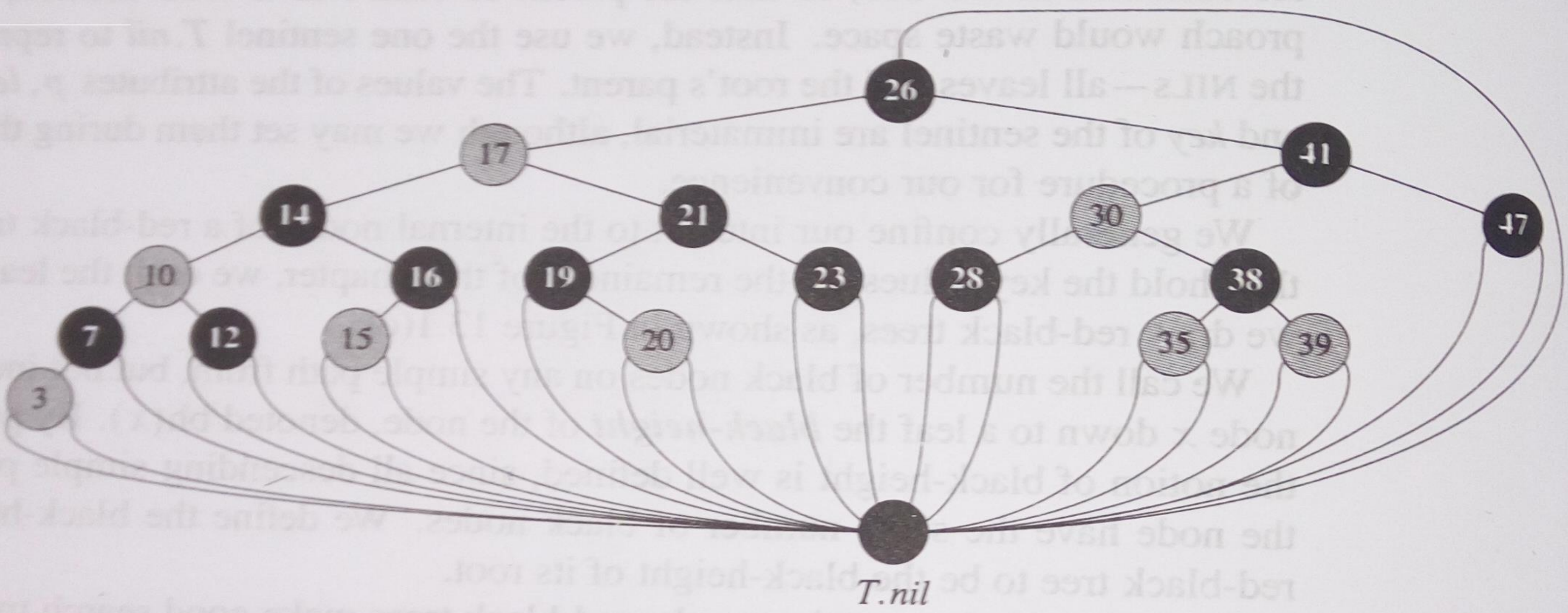
Red-Black Properties

- The *red-black properties*:
  1. Every node is either red or black
  2. Every leaf (NULL pointer) is black
     - Note: this means every “real” node has 2 children
  3. If a node is red, both children are black
     - Note: can’t have 2 consecutive reds on a path
  4. Every path from node to descendent leaf contains the same number of black nodes
  5. The root is always black

Red-Black Tree

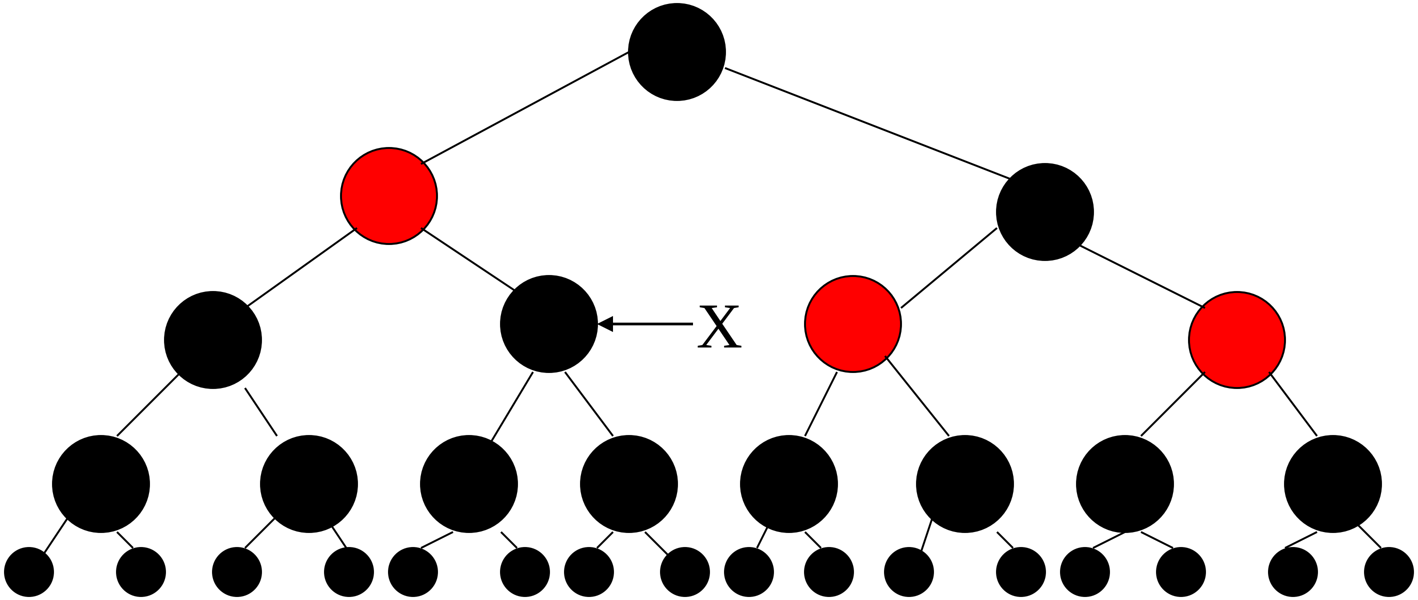
- Each node has the following attributes:
  - Color
  - Key
  - Left child
  - Right child
  - Parent
- If a child of the parent of a node does not exist, the corresponding pointer attribute of the node contains the value NIL.
- We shall regard these NIL’s as being pointers to leaves (external nodes) of the BST and the normal, key bearing nodes as being internal nodes of the tree.
- By constraining the node colors on any simple path from the root to a leaf, red black trees ensure that no such path is more than twice as long as any other path.


26
17
41
14
21
30
47
10
16
19
23
28
38
7
12
15
20
35
39
3
NIL
(a)


26
17
41
14
21
30
47
10
16
19
23
28
38
7
12
15
20
35
39
3
T.nil
(b)

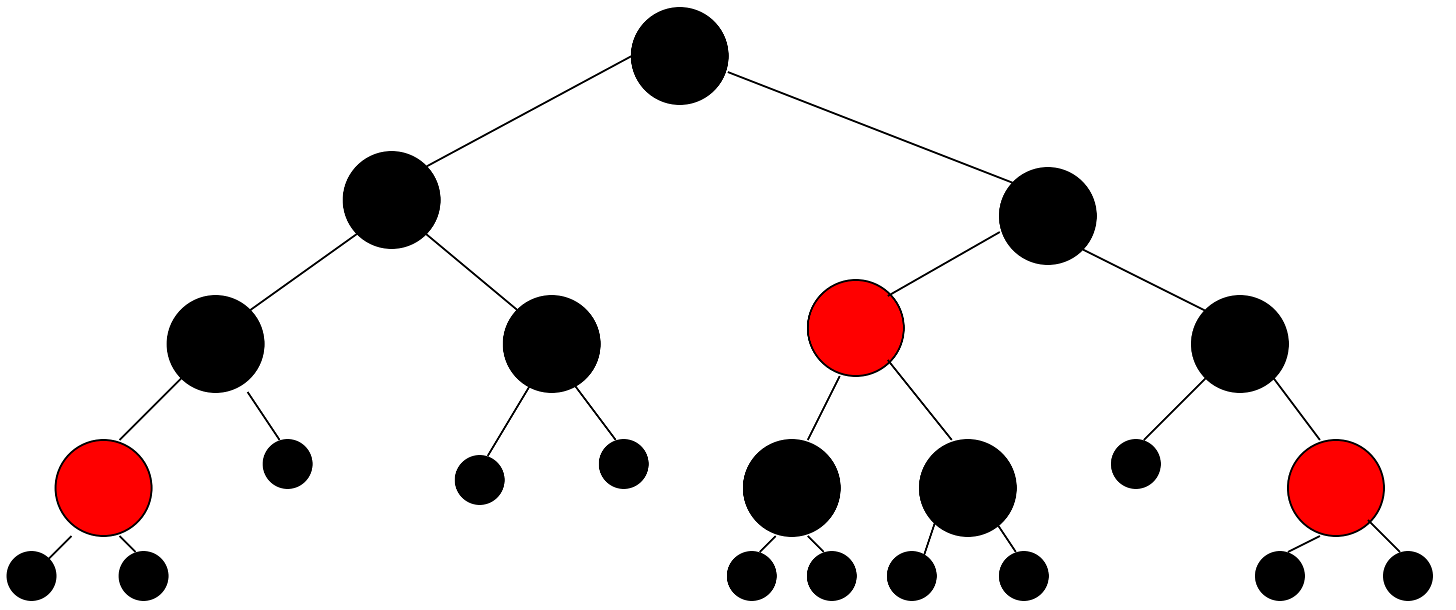
Black-Height

- ***Black-height:*** The black-height of a node, X, in a red-black tree is the number of Black nodes on any path to a NULL (or leaf), not counting X.



A Red-Black Tree with NULLs shown

Black-Height of the tree (the root) = 3
Black-Height of node “X” = 2

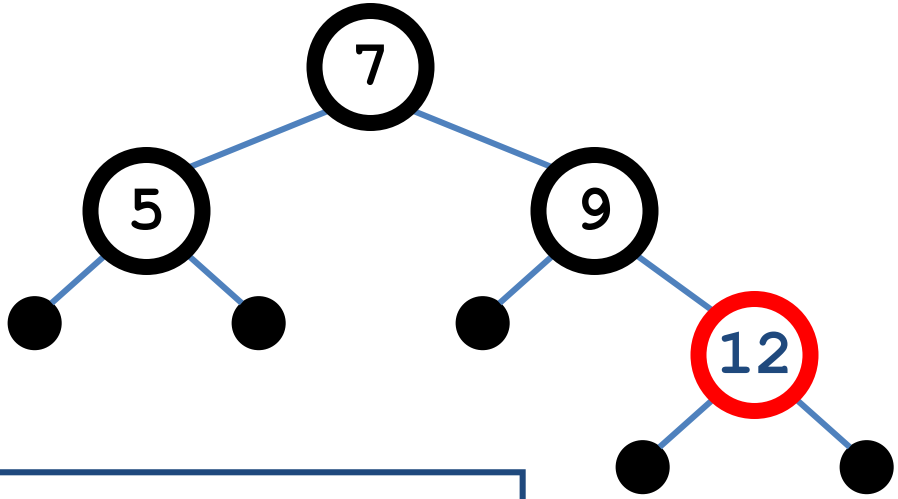


A Red-Black Tree with

Black-Height = _____

Red-Black Trees: An Example

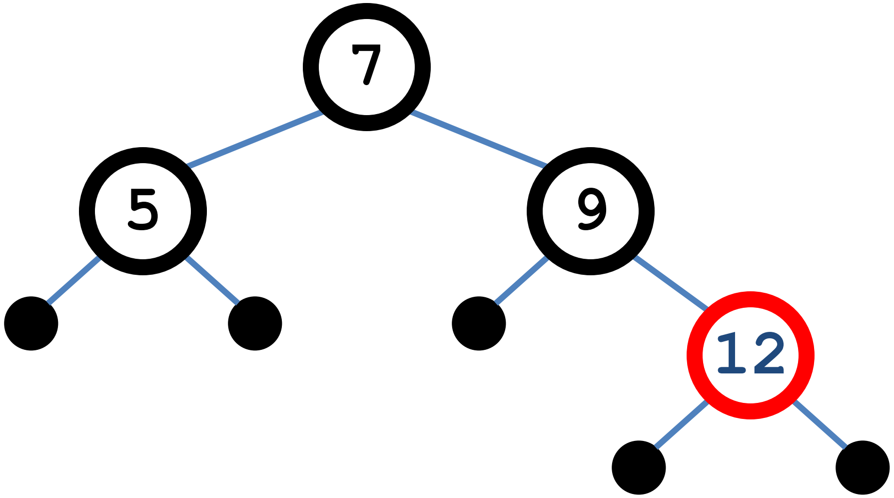
- *Color this tree:*



Red-black properties:
1. Every node is either red or black
2. Every leaf (NULL pointer) is black
3. If a node is red, both children are black
4. Every path from node to descendent leaf contains the same number of black nodes
5. The root is always black

Red-Black Trees: The Problem With Insertion

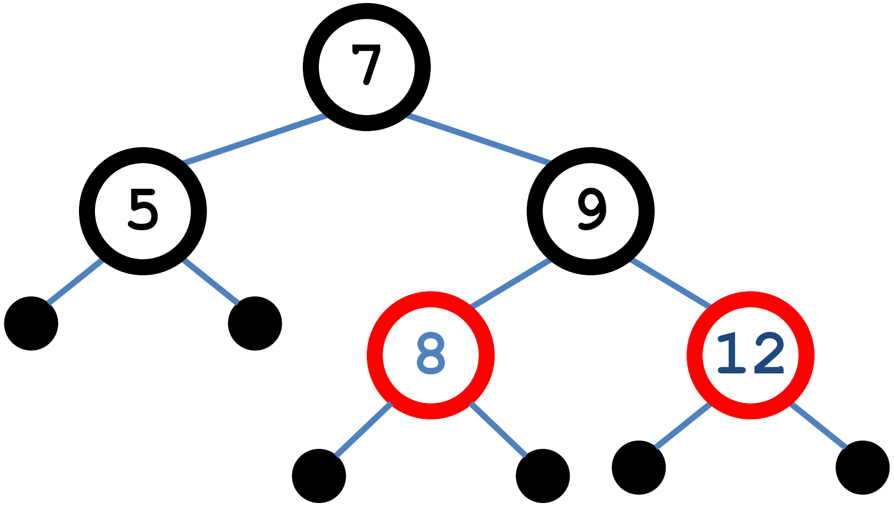
- Insert 8
  - *Where does it go?*



1. Every node is either red or black
2. Every leaf (NULL pointer) is black
3. If a node is red, both children are black
4. Every path from node to descendent leaf contains the same number of black nodes
5. The root is always black

Red-Black Trees: The Problem With Insertion

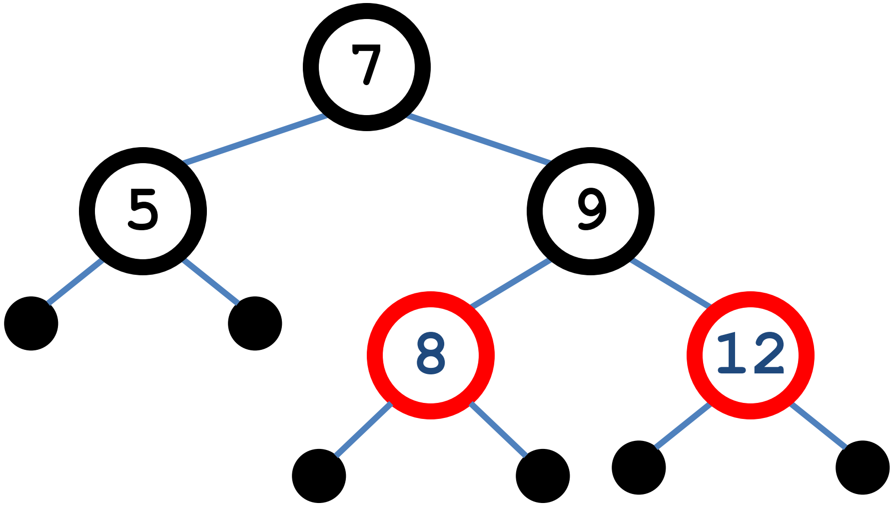
- Insert 8
  - *Where does it go?*
  - *What color should it be?*



1. Every node is either red or black
2. Every leaf (NULL pointer) is black
3. If a node is red, both children are black
4. Every path from node to descendent leaf contains the same number of black nodes
5. The root is always black

Red-Black Trees: The Problem With Insertion

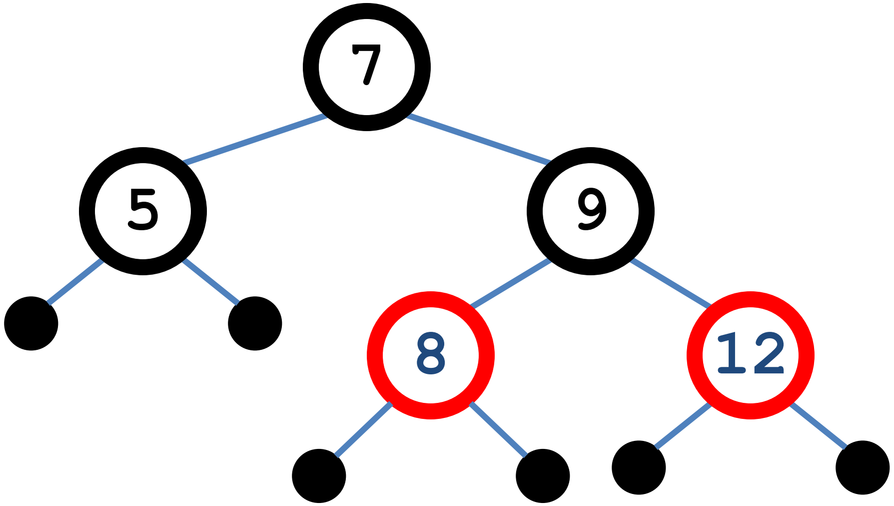
- Insert 8
  - *Where does it go?*
  - *What color should it be?*



1. Every node is either red or black
2. Every leaf (NULL pointer) is black
3. If a node is red, both children are black
4. Every path from node to descendent leaf contains the same number of black nodes
5. The root is always black

Red-Black Trees: The Problem With Insertion

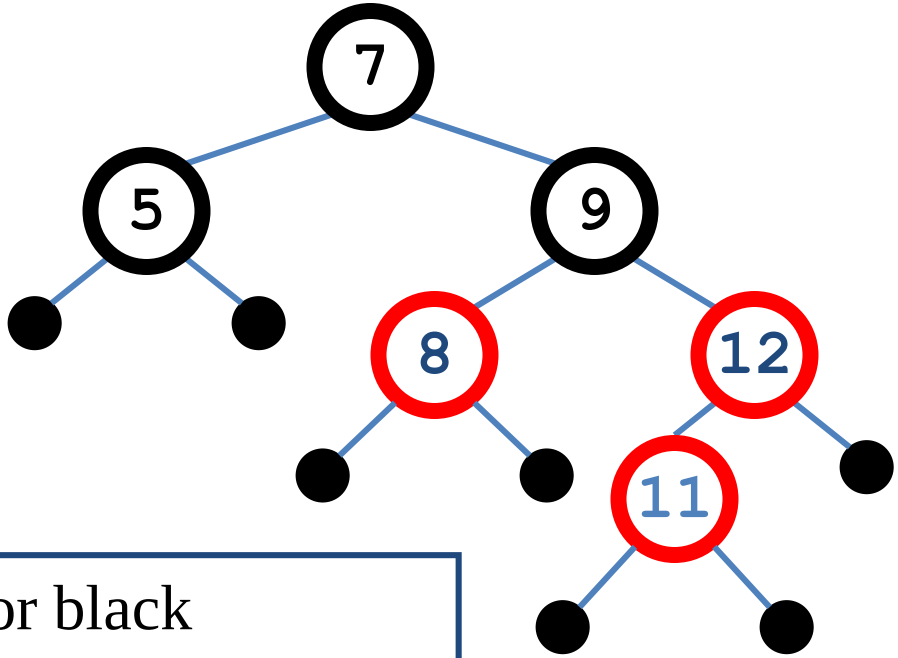
- Insert 11
  - *Where does it go?*



1. Every node is either red or black
2. Every leaf (NULL pointer) is black
3. If a node is red, both children are black
4. Every path from node to descendent leaf contains the same number of black nodes
5. The root is always black

Red-Black Trees: The Problem With Insertion

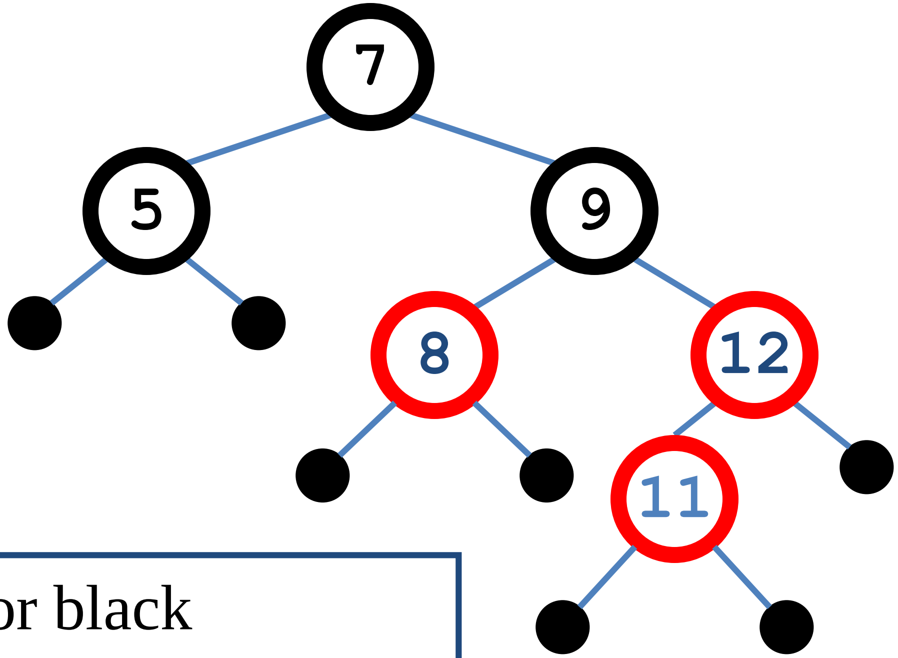
- Insert 11
  - *Where does it go?*
  - *What color?*



1. Every node is either red or black
2. Every leaf (NULL pointer) is black
3. If a node is red, both children are black
4. Every path from node to descendent leaf contains the same number of black nodes
5. The root is always black

Red-Black Trees: The Problem With Insertion

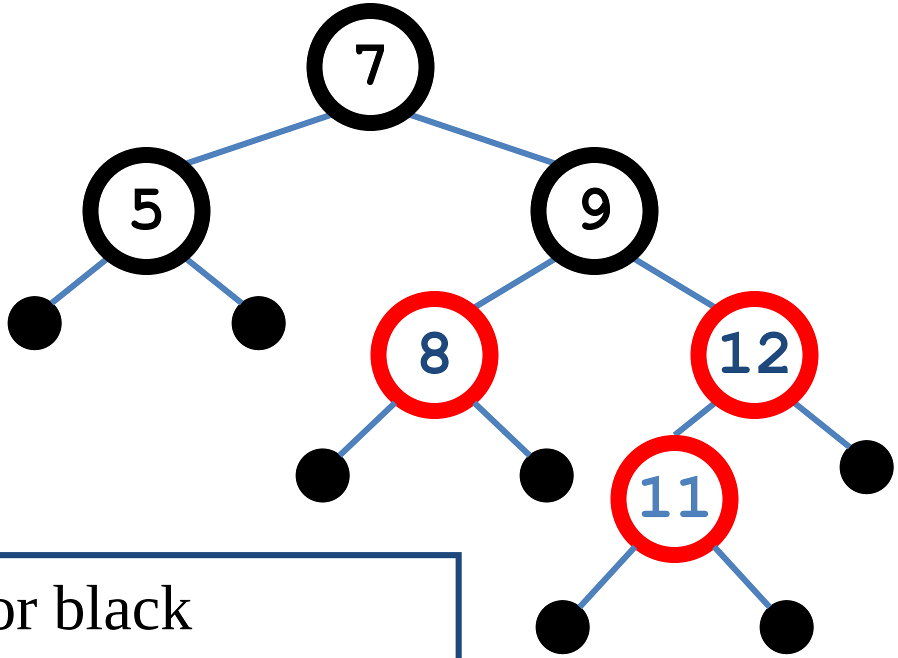
- Insert 11
  - *Where does it go?*
  - *What color?*
    - Can't be red! (#3)



1. Every node is either red or black
2. Every leaf (NULL pointer) is black
3. If a node is red, both children are black
4. Every path from node to descendent leaf contains the same number of black nodes
5. The root is always black

Red-Black Trees: The Problem With Insertion

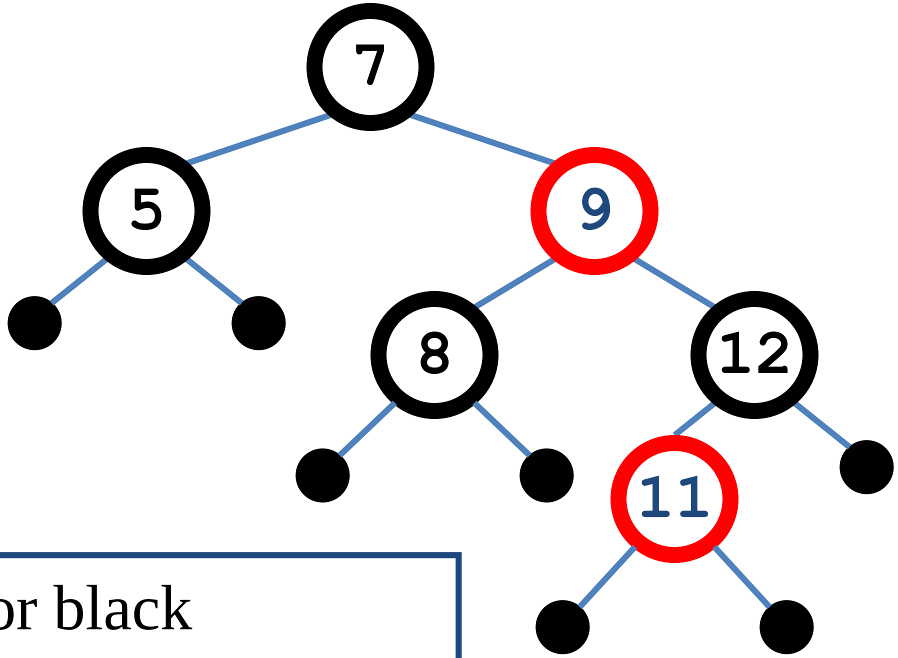
- Insert 11
  - *Where does it go?*
  - *What color?*
    - Can't be red! (#3)
    - Can't be black! (#4)



1. Every node is either red or black
2. Every leaf (NULL pointer) is black
3. If a node is red, both children are black
4. Every path from node to descendent leaf contains the same number of black nodes
5. The root is always black

Red-Black Trees: The Problem With Insertion

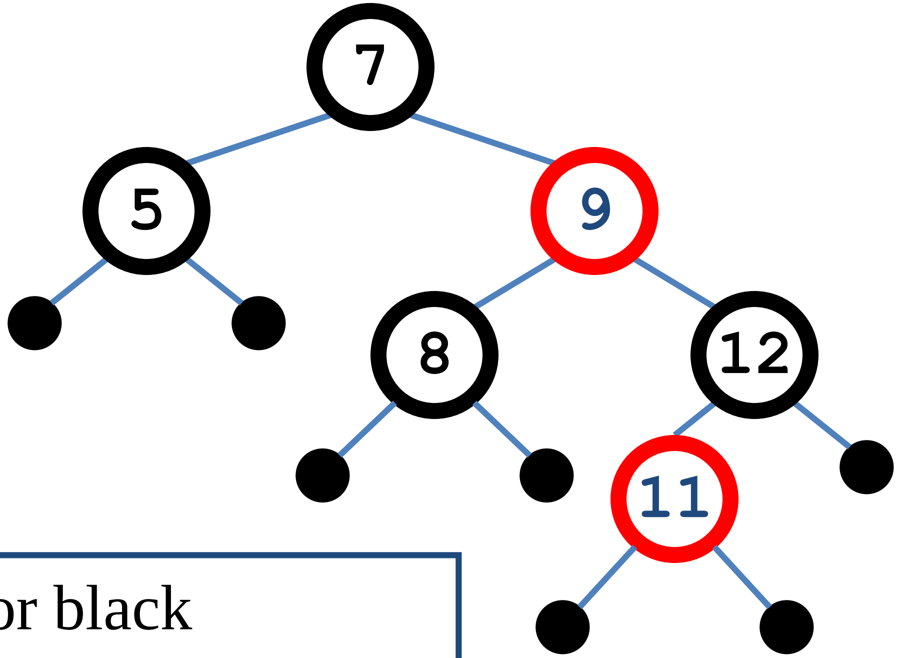
- Insert 11
  - *Where does it go?*
  - *What color?*
    - Solution: recolor the tree



1. Every node is either red or black
2. Every leaf (NULL pointer) is black
3. If a node is red, both children are black
4. Every path from node to descendent leaf contains the same number of black nodes
5. The root is always black

Red-Black Trees: The Problem With Insertion

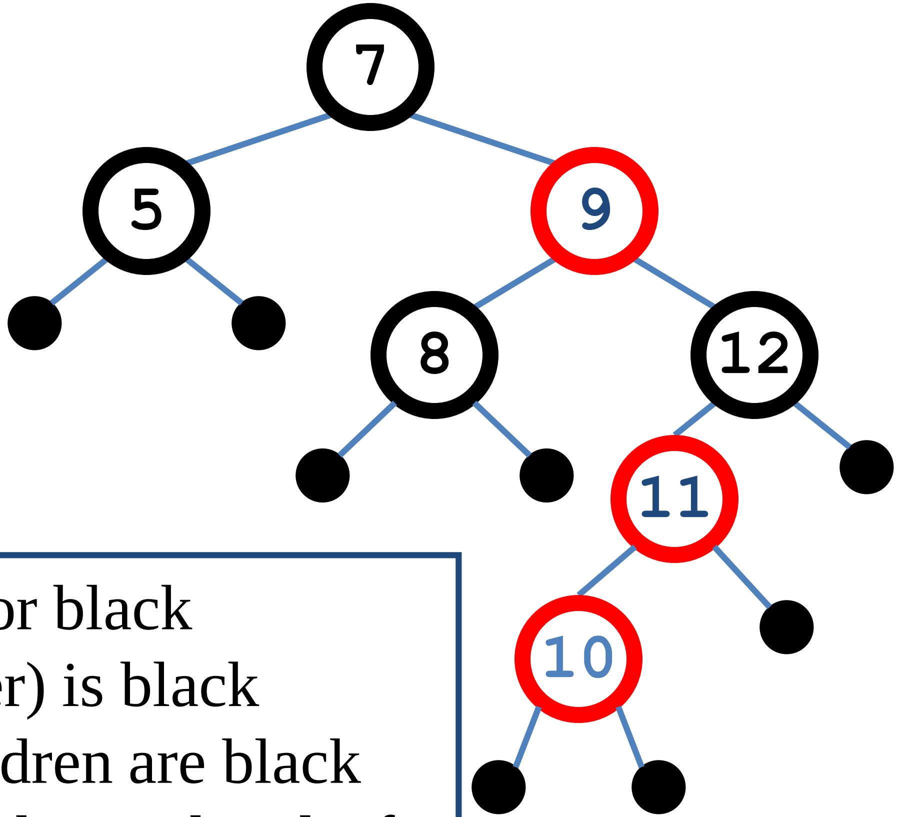
- Insert 10
  - *Where does it go?*



1. Every node is either red or black
2. Every leaf (NULL pointer) is black
3. If a node is red, both children are black
4. Every path from node to descendent leaf contains the same number of black nodes
5. The root is always black

Red-Black Trees: The Problem With Insertion

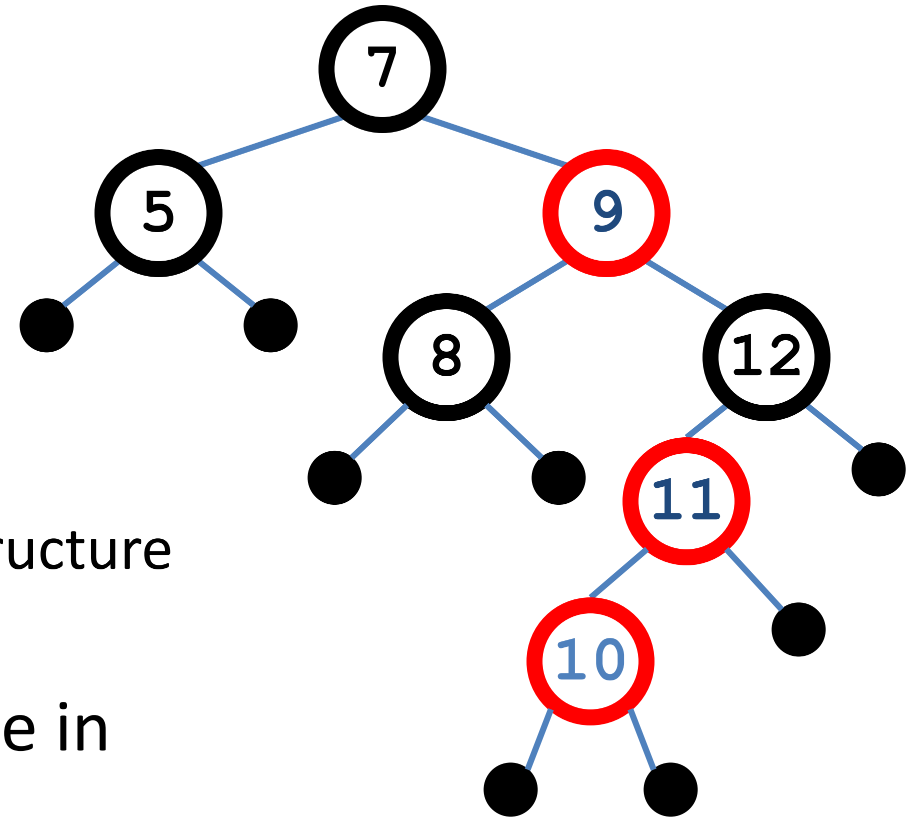
- Insert 10
  - *Where does it go?*
  - *What color?*



1. Every node is either red or black
2. Every leaf (NULL pointer) is black
3. If a node is red, both children are black
4. Every path from node to descendent leaf contains the same number of black nodes
5. The root is always black

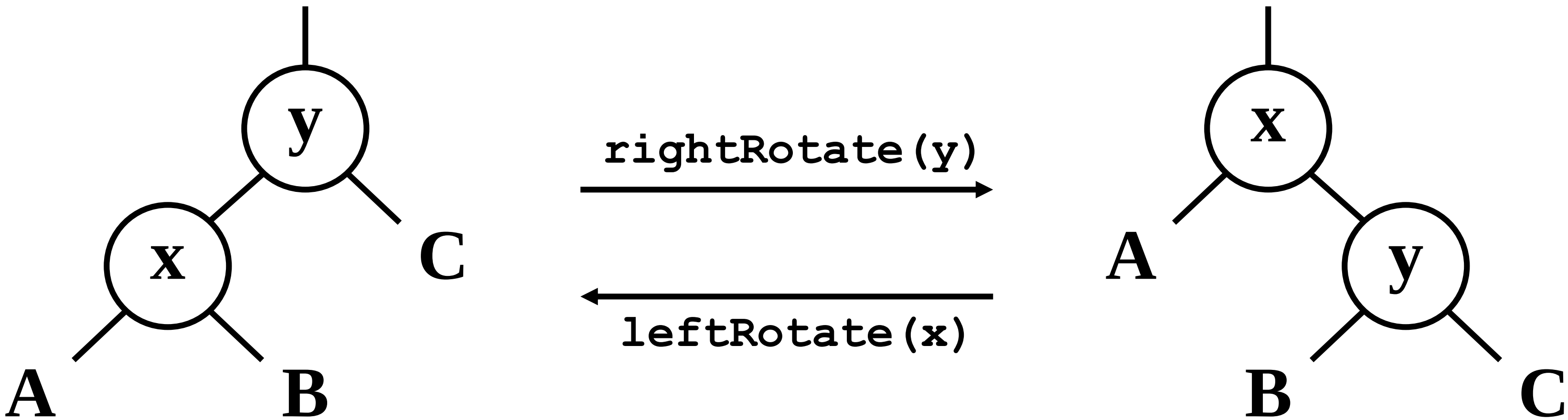
Red-Black Trees: The Problem With Insertion

- Insert 10
  - *Where does it go?*
  - *What color?*
    - A: no color! Tree is too imbalanced
    - Must change tree structure to allow recoloring
  - Goal: restructure tree in O(lg *n*) time

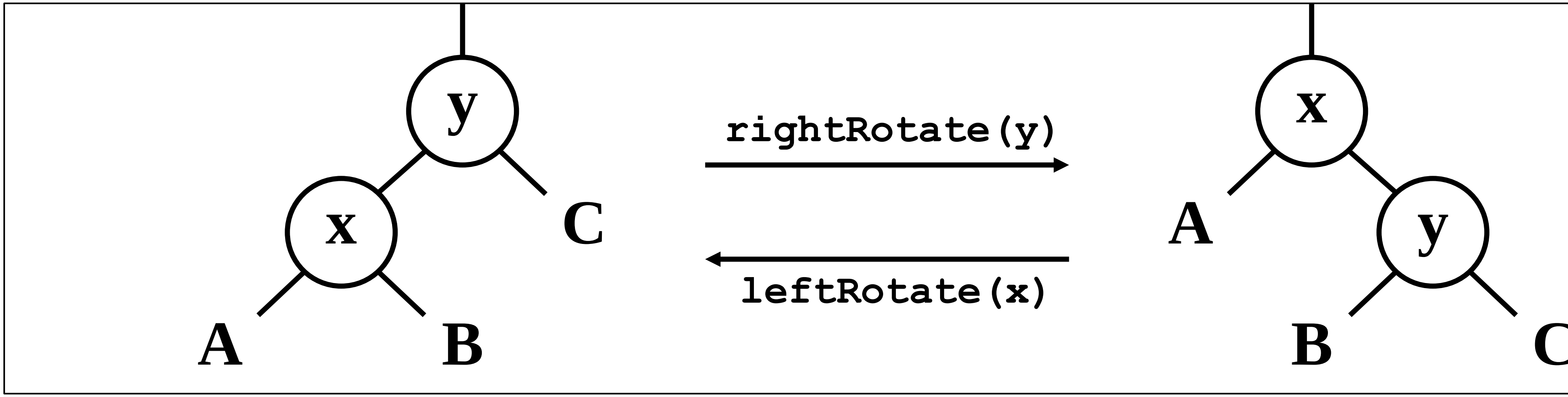


RB Trees: Rotation

- Our basic operation for changing tree structure is called *rotation*:

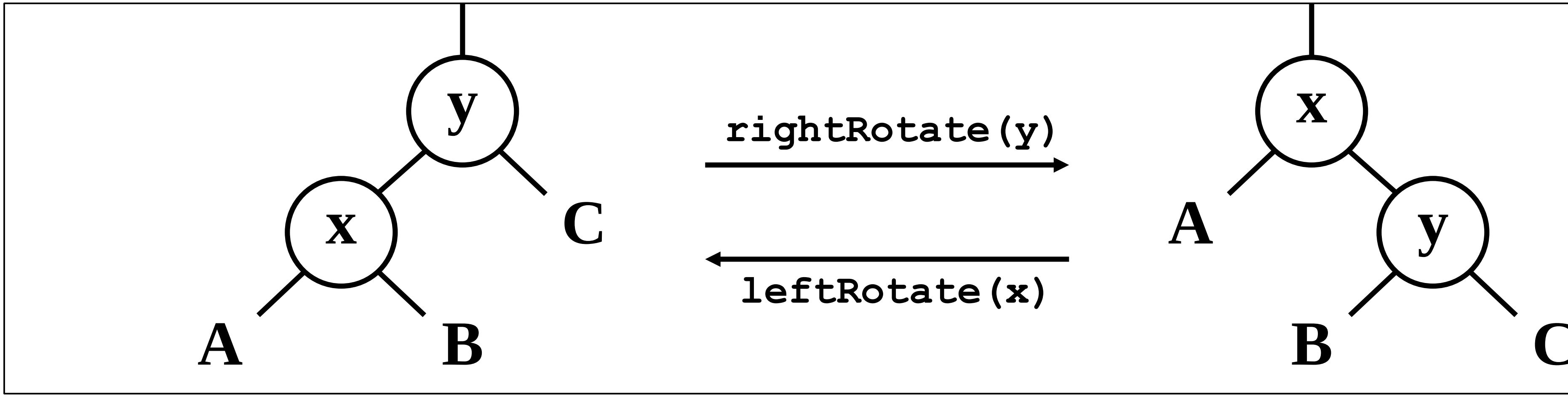


- *Does rotation preserve inorder key ordering?*



Left-Rotate(T,x)

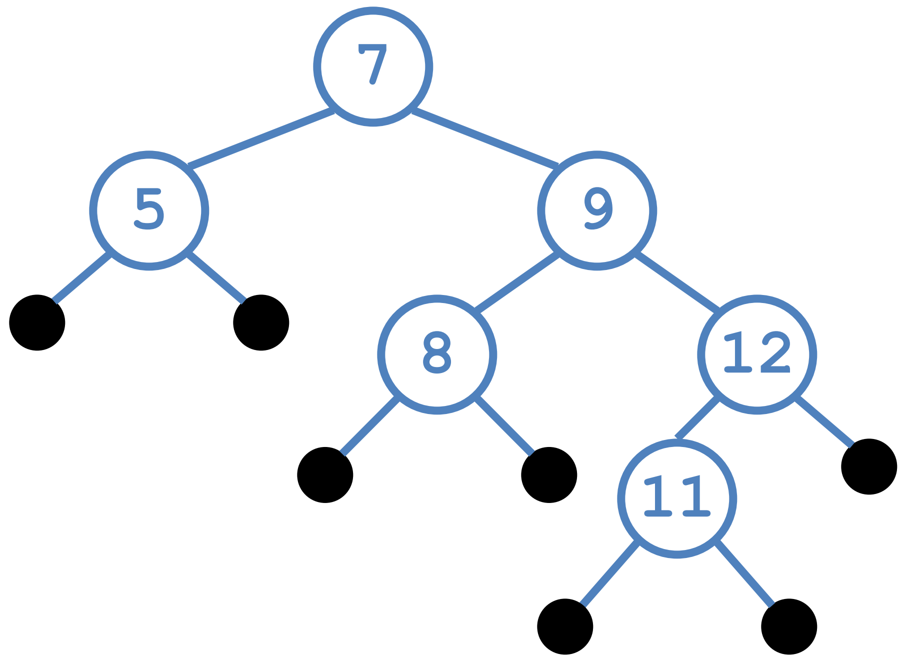
```
y = right(x)                    ; assume right(x) <> NIL
right(x) = left(y)              ; move y's child over
if left(y) <> NIL
then parent(left(y)) = x
parent(y) = parent(x)           ; move y up to x's position
if parent(x) = NIL
then root(T) = y
else if x = left(parent(x))
    then left(parent(x)) = y
    else right(parent(x)) = y
left(y) = x                     ; move x down
parent(x) = y
```




```
Right-Rotate(T,y)
    x = left(y)          ; assume left(y) <> NIL
    left(y) = right(x)
    if right(x) <> NIL
    then parent(right(x)) = y
    parent(x) = parent(y)
    if parent(y) = NIL
    then root(T) = x
    else if y = left(parent(y))
        then left(parent(y)) = x
        else right(parent(y)) = x
    right(x) = y
    parent(y) = x
```

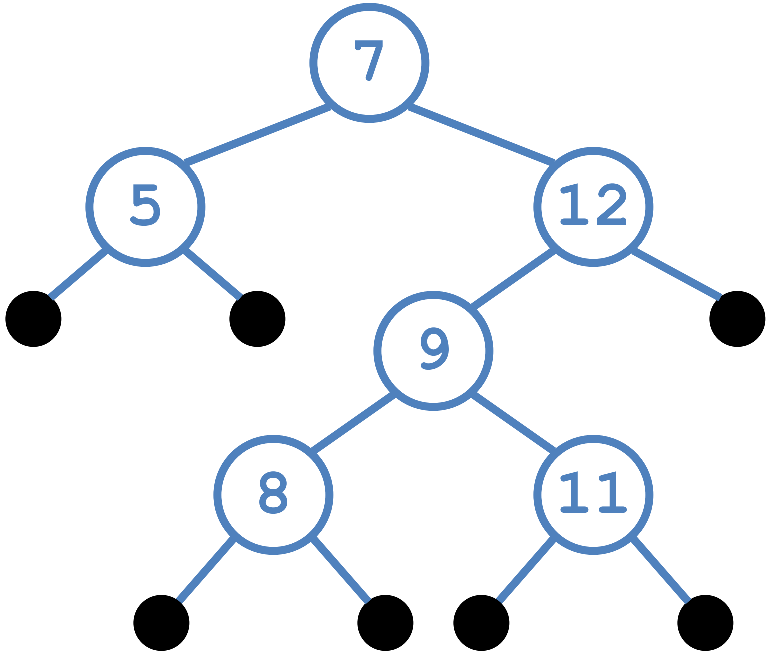
Rotation Example

- Rotate left about 9:



Rotation Example

- Rotate left about 9:



Red-Black Trees: Insertion

- Insertion: the basic idea
  - Insert *x* into tree, color *x* red
  - Which of the red-black properties might be violated?
    - Root is always black
    - Red node cannot have a red child

  - Fix the violated properties.

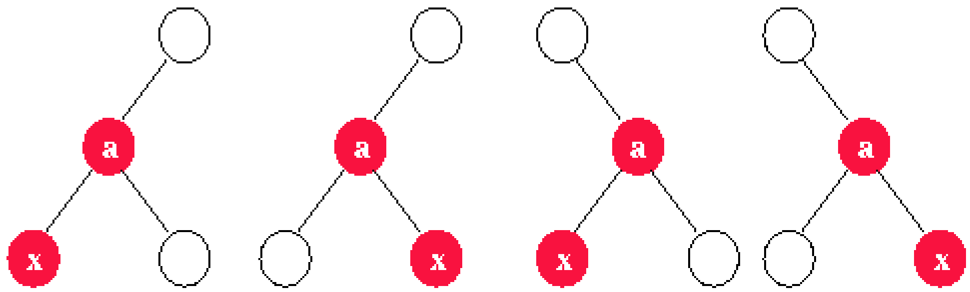
Red-Black Trees: Insertion

Insertion

1. Insert node into tree using BST Insert(T,x) and color node Red
2. Fix violated RBT properties
    1. Root is always black
    2. Red node cannot have a red child
3. Color root Black

Red-Black Trees: Insertion

- If parent node `a' was Black, then no changes are necessary.
- If not, then there are following cases to consider for each of the orientations below.



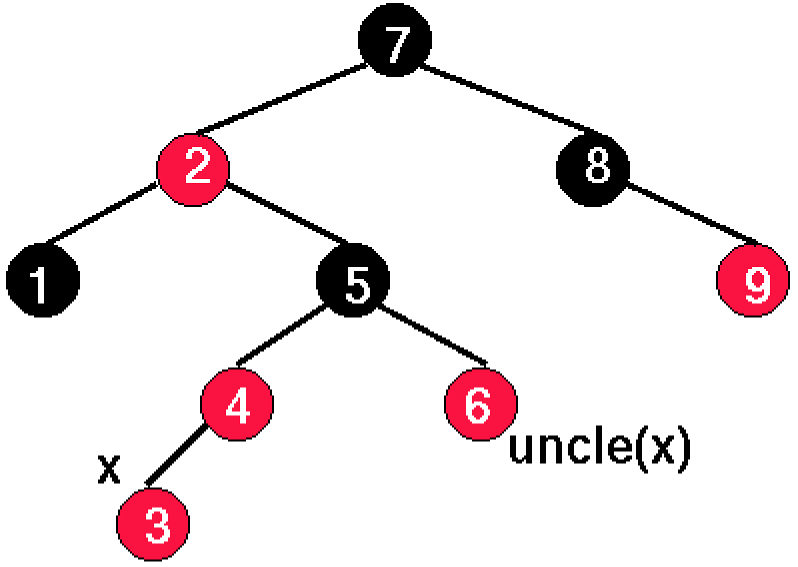
- Move up the tree until there are no violations or we are at the root.
- In the following discussion we will assume the parent is a left child (if the parent is a right child perform the same steps swapping ``right'' and ``left'')

Red-Black Trees: Insertion

RB-Insert(T,x)

Case I: x's uncle is Red

- Change x's grandparent to Red
- Change x's uncle and parent to Black
- Change x to x's grandparent



How to get uncle (x)

if parent(x) = left(parent(parent(x)))
  then uncle(x) = right(parent(parent(x)))
  else uncle(x) = left(parent(parent(x)))

Red-Black Trees: Insertion

RB-Insert(T,x)

Case II: x's uncle is Black, x is the right child of its parent

- Change x to x's parent
- Rotate x's parent (now x) left to make Case III
- Case II is now Case III

Case III: x's uncle is Black, x is the left child of its parent

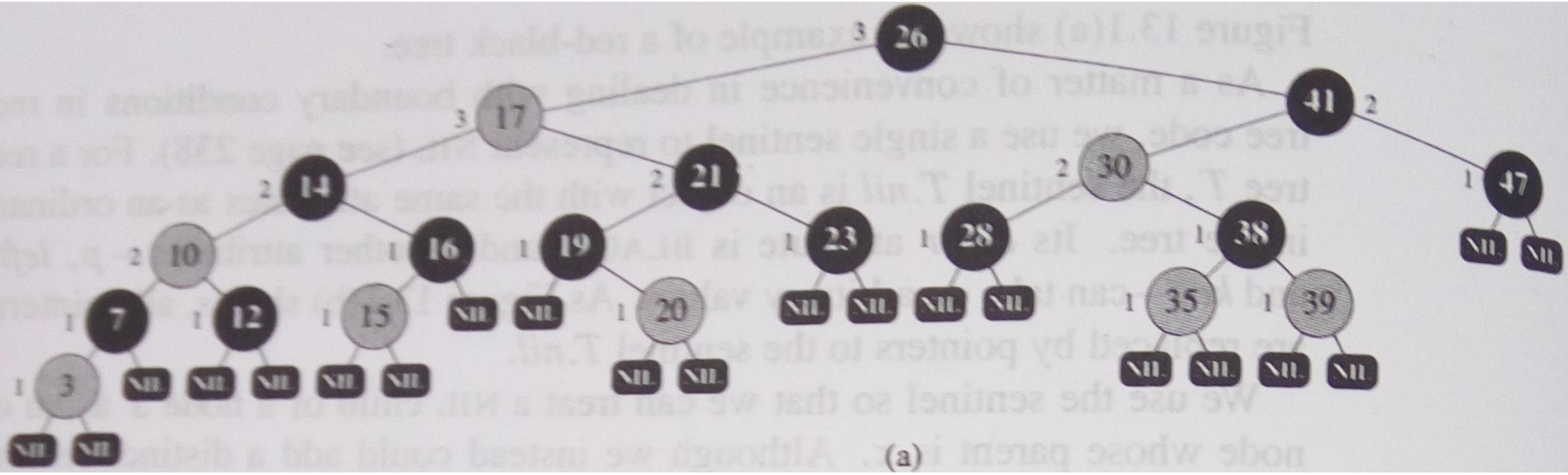
- Set x's parent to Black
- Set x's grandparent to Red
- Rotate x's grandparent right

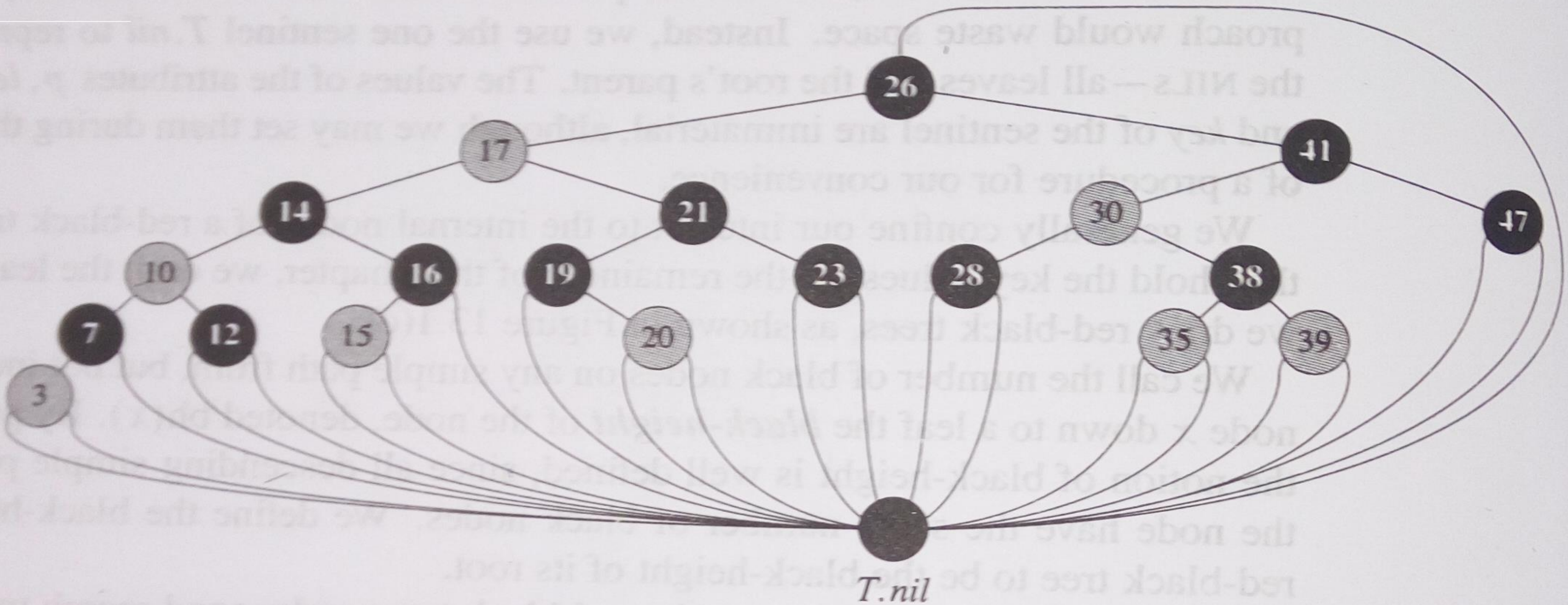
Red-Black Trees: Insertion

Example:

Insert the following keys in a Red Black Tree in order:

3 2 5 6 9 4 7 8


26
17
41
14
21
30
47
10
16
19
23
28
38
7
12
15
20
35
39
3
NIL
3
3
2
2
2
2
1
1
1
1
1
1
1
1
1
1
1
1
1
(a)


26
17
41
14
21
30
47
10
16
19
23
28
38
7
12
15
20
35
39
3
T.nil
(b)

Theorem 1 – In a red-black tree, any subtree rooted at x contains atleast $(\mathbf{2^{bh(x)} - 1})$ internal nodes, where bh(x) is the black height of node x.

Proof: by induction on height of x.

In a red-black tree, at least half the nodes on any path from the root to a NULL (i.e. leaf) must be Black.

Proof – If there is a Red node on the path, there must be a corresponding Black node.

Algebraically this theorem means

$$bh(\ x\ ) \geq h/2$$

where x is the root node.

In a red-black tree, no path from any node, X, to a NULL (i.e. leaf) is more than twice as long as any other path from X to any other NULL (i.e. leaf).

Proof: By definition, every path from a node to any NULL contains the same number of Black nodes. By Theorem 2, atleast ½ the nodes on any such path are Black. Therefore, there can be no more than twice as many nodes on any path from X to a NULL as on any other path. Therefore the length of every path is no more than twice as long as any other path.

Theorem 4 – A red-black tree with ***n*** internal nodes has height **$h \le 2 \lg(n + 1)$**.

Proof: Let h be the height of the red-black tree with root x. By Theorem 2,

$$bh(x) \ge h/2$$

From Theorem 1, $n \ge 2^{bh(x)} - 1$

Therefore $n \ge 2^{h/2} - 1$

$$n + 1 \ge 2^{h/2}$$

$$\lg(n + 1) \ge h/2$$

$$2\lg(n + 1) \ge h$$

RB Trees: Worst-Case Time

- So we've proved that a red-black tree has O(lg *n*) height
- Corollary: These operations take O(lg *n*) time:
  - Minimum(), Maximum()
  - Successor(), Predecessor()
  - Search()
- Insert() and Delete():
  - Will also take O(lg *n*) time
  - But will need special care since they modify tree

Red-Black Trees: Deletion

- As insert had three cases, delete has four different cases.
- Do it yourself.

innovate achieve lead

BITS Pilani

Pilani|Dubai|Goa|Hyderabad

Thank You