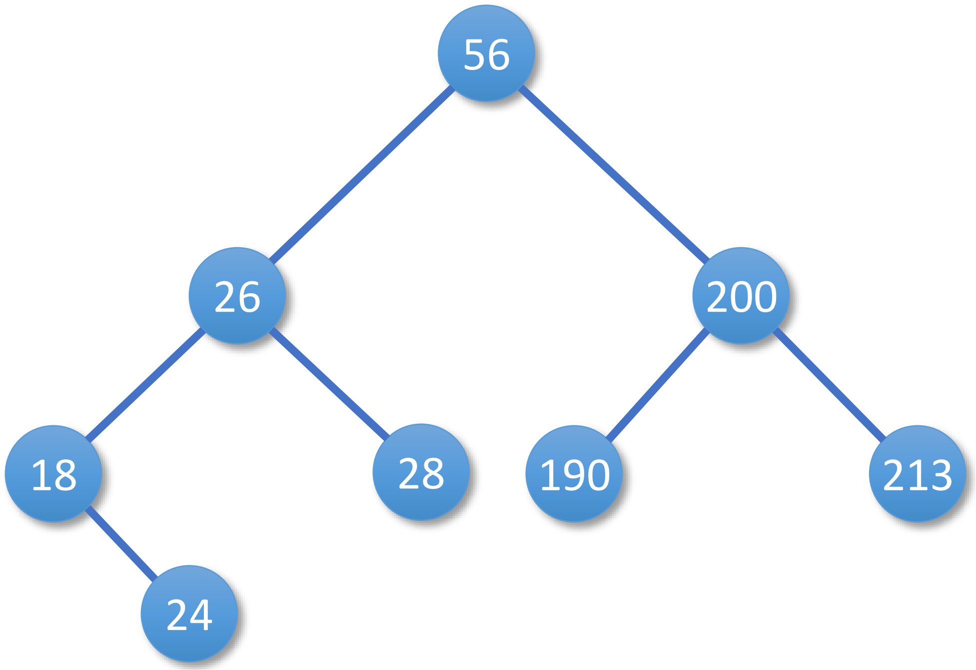
DATA STRUCTURES PART II

Sid Chi-Kin Chau

[Lecture 3]

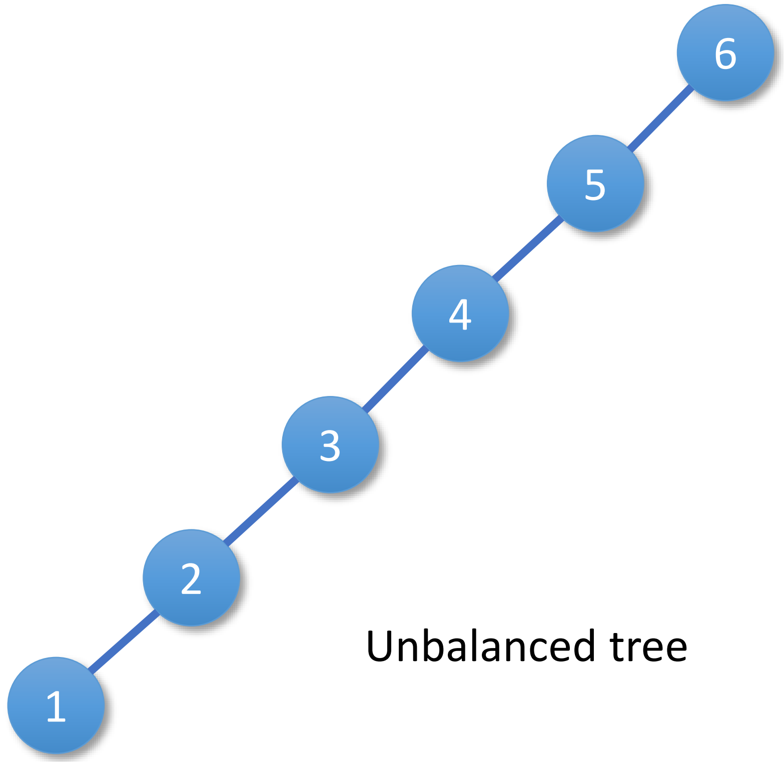
Recap from Previous Lecture

- Binary search tree
  - At most two children for each node
  - Left child node is smaller than its parent node
  - Right child node is greater than its parent node
  - Can support dynamic set operations
    - Search, Minimum, Maximum, Predecessor, Successor, Insert, and Delete

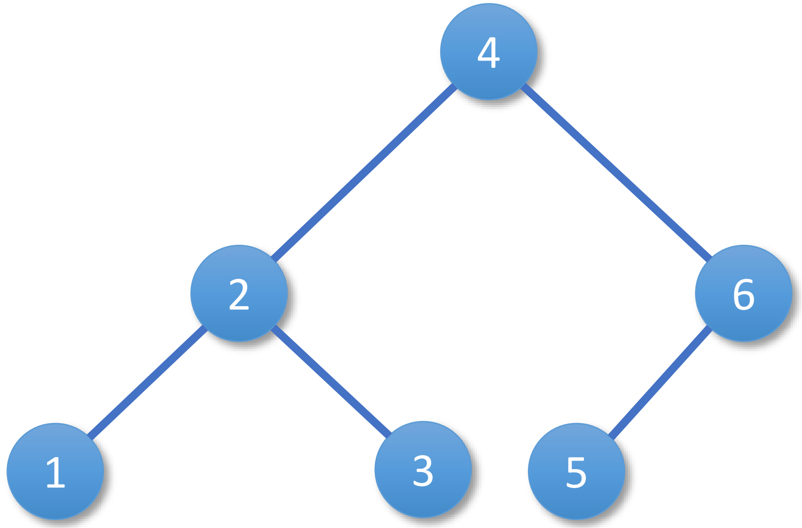


Worse-case Scenario

- Insert 6, 5, 4, 3, 2, 1 in an empty tree in order
- Depth of tree linear increases



Unbalanced tree

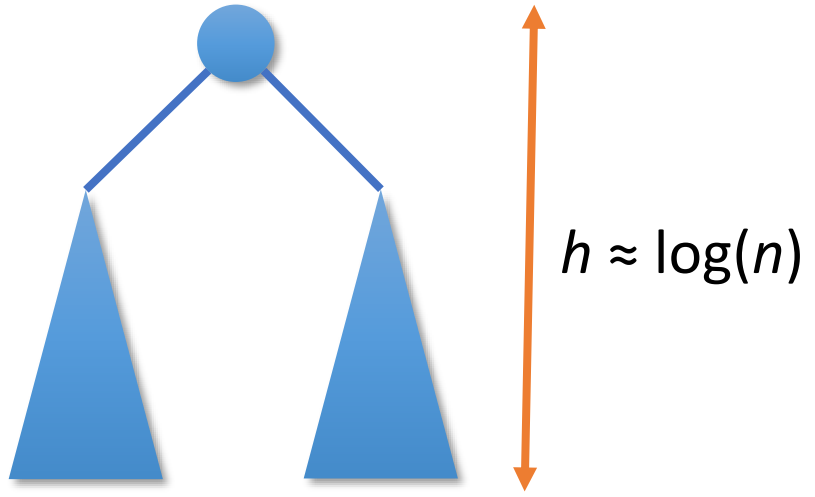


Balanced tree

Balanced Tree

- Balanced search tree
  - Belong to binary search tree
  - But with a height of O(log(n)) guaranteed for n items
  - Height h = maximum number of **edges** from the root to a leaf

- Examples
  - Red-black tree
  - AVL tree
  - B-tree



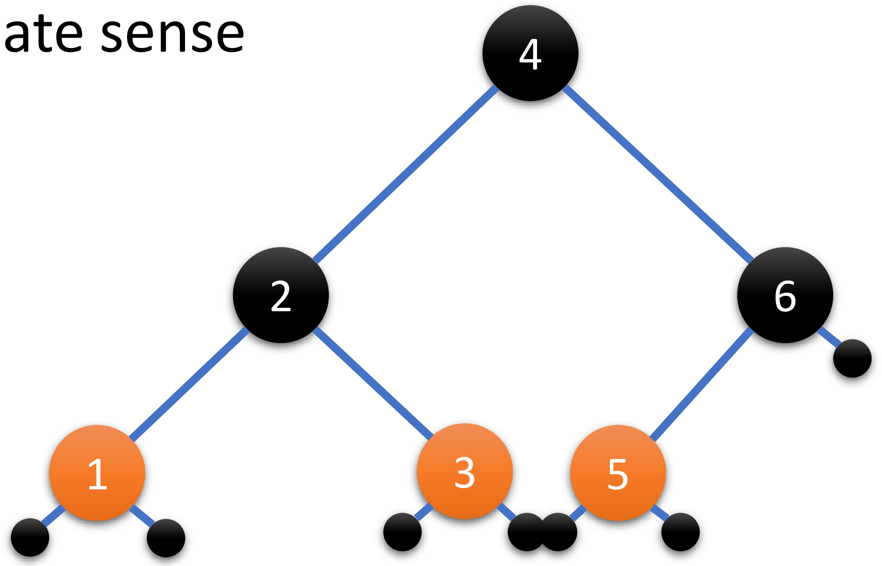
Goals of This Lecture

- Red-black tree
  - Balanced tree by coloring the nodes
- AVL tree is similar to red-black tree
- B-tree
  - Balanced tree by keeping all leaves at same level

- Something else to learn
  - Applications of different data structures

Red-Black Tree

- Close to balanced tree
- Tree structure requires an extra one-bit color field in each node: either red or **black**
  - Color is used to maintain balance in an approximate sense
- Node:
  - Key
  - Color
  - Left
  - Right
  - Parent
- Note: Leave nodes are NULL nodes (with no child)

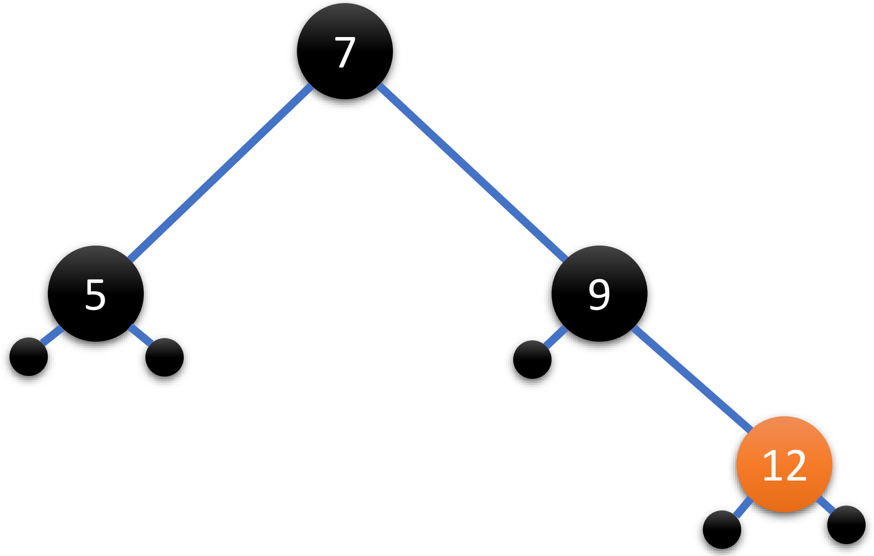


Red-Black Properties

- The red-black properties:
  1. Every node is either red or **black**
  2. (a) Root and (b) leaves (NULL node) are **black**
     - Note: this means every "real" node has 2 children
  3. If a node is red, both children are **black**
     - Note: cannot have 2 consecutive reds on a path
  4. Every path from node to descendent leaf contains the same number of **black** nodes
- "Black-height" is the number of **black** nodes on a path to a leaf
  - Red-black property #4 ensures the black-height is independent of any leaf

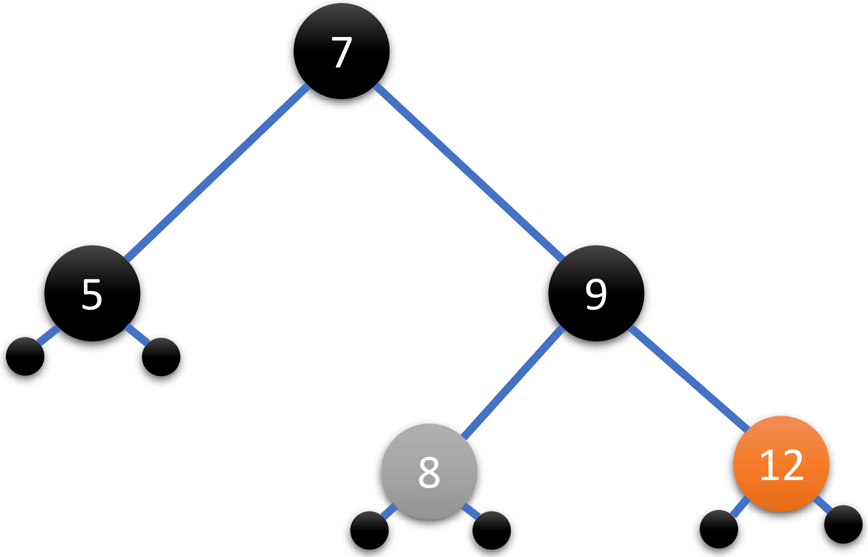
Example

- The red-black properties:
  1. Every node is either red or **black**
  2. Root and leaves (NULL node) are **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



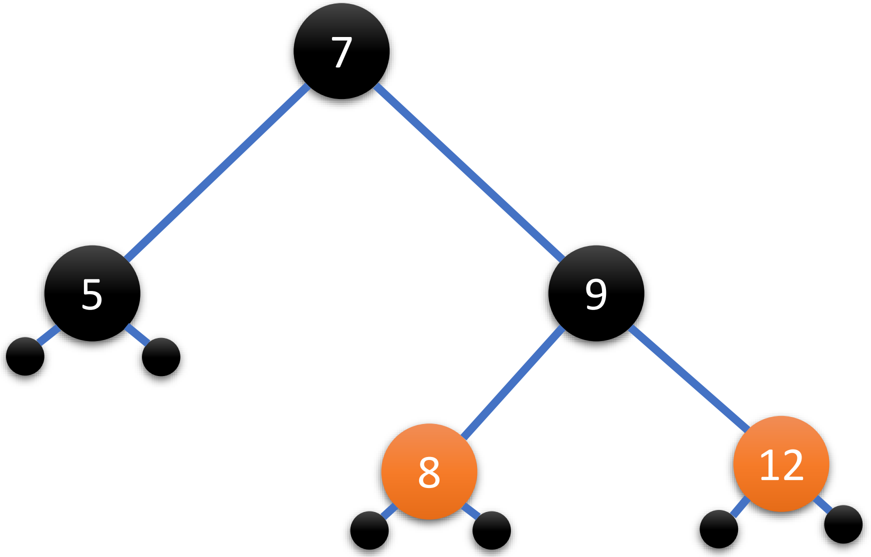
Example

- The red-black properties:
  1. Every node is either red or **black**
  2. Root and leaves (NULL node) are **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
- Add a new node with x.key = 8



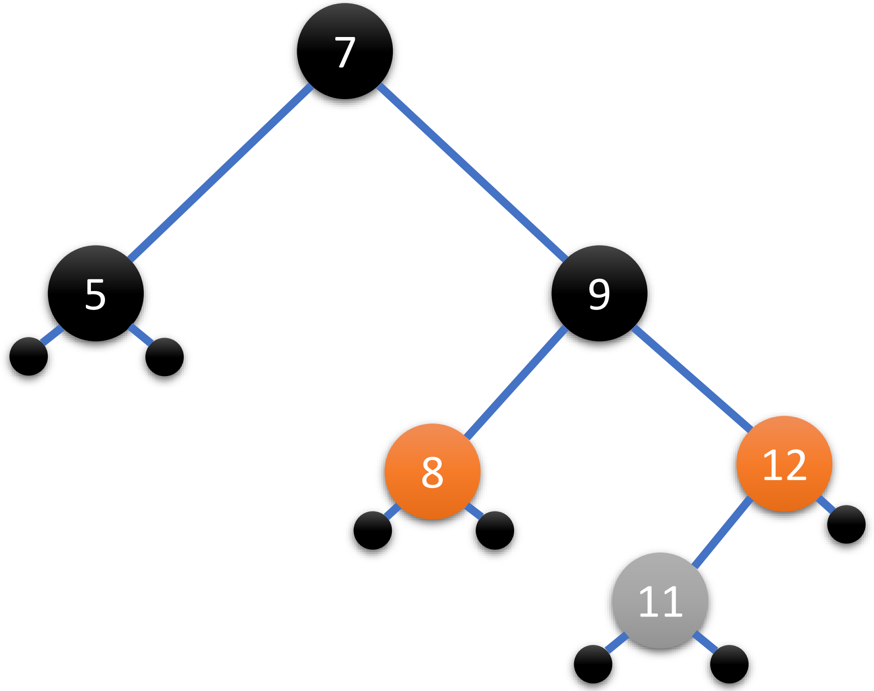
Example

- The red-black properties:
  1. Every node is either red or **black**
  2. Root and leaves (NULL node) are **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
- Add a new node with x.key = 8
  - Insert x as in binary search tree
  - Set x.color ← red, satisfying all red-black properties



Example

- The red-black properties:
  1. Every node is either red or **black**
  2. Root and leaves (NULL node) are **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
- Add a new node with y.key = 11
  - Insert y as in binary search tree
  - Set y.color $\leftarrow$???
  - Cannot satisfy all red-black properties



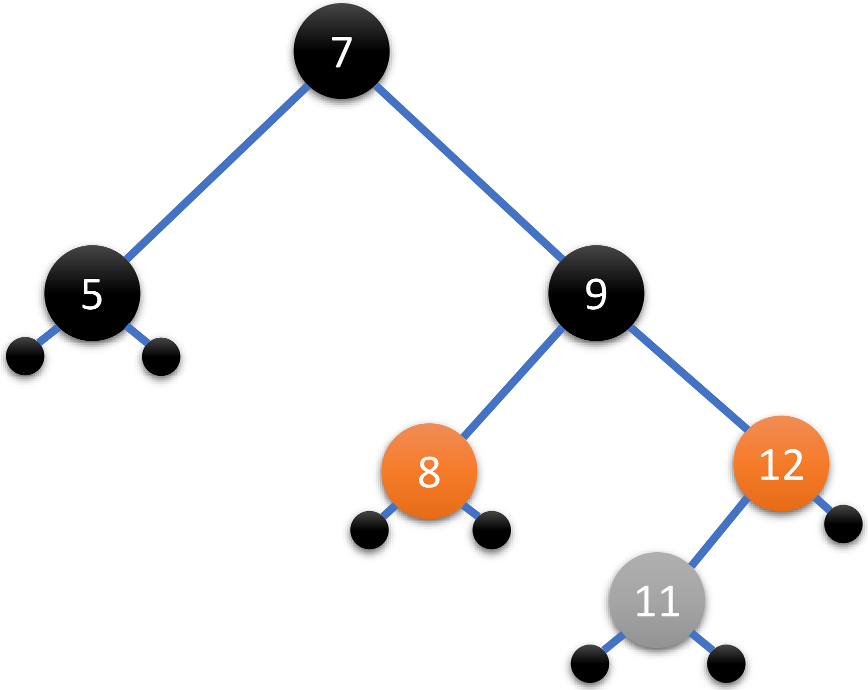
Red-Black Tree Insertion

- Insertion: the basic idea
  - Insert the new node as in binary search tree
  - Color the new node **red**
  - Only red-black property #3 may be violated
    - It will be violated if the new node's parent is **red**
  - If so, move violation up the tree until a place is found where it can be fixed

Example

- Insertion:
  - Insert the new node as in binary search tree
  - Color the new node **red**

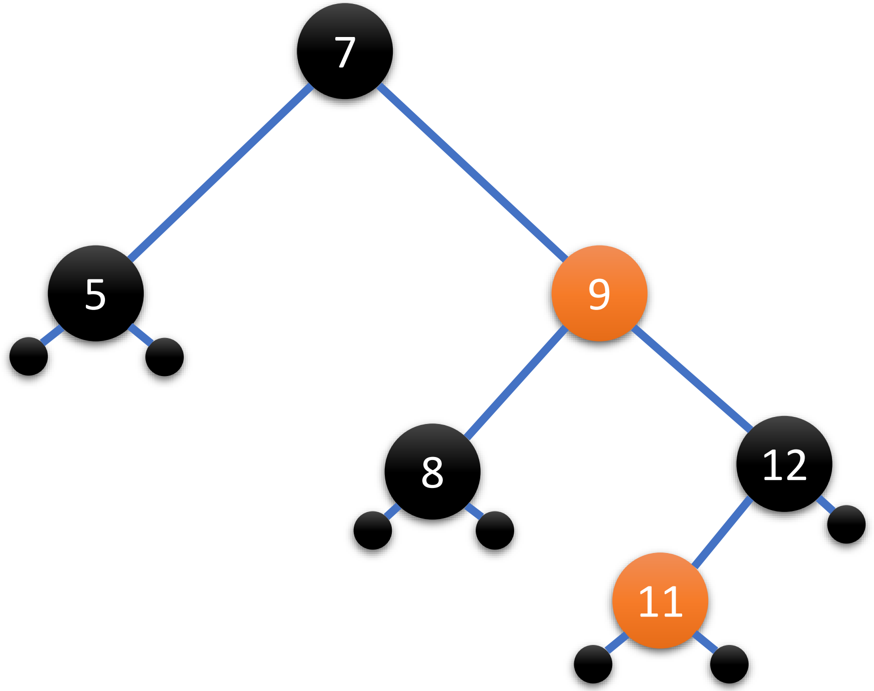
- Add a new node with y.key = 11
  - Insert y as in binary search tree
  - Set y.color ← ???



Example

- Insertion:
  - Insert the new node as in binary search tree
  - Color the new node **red**

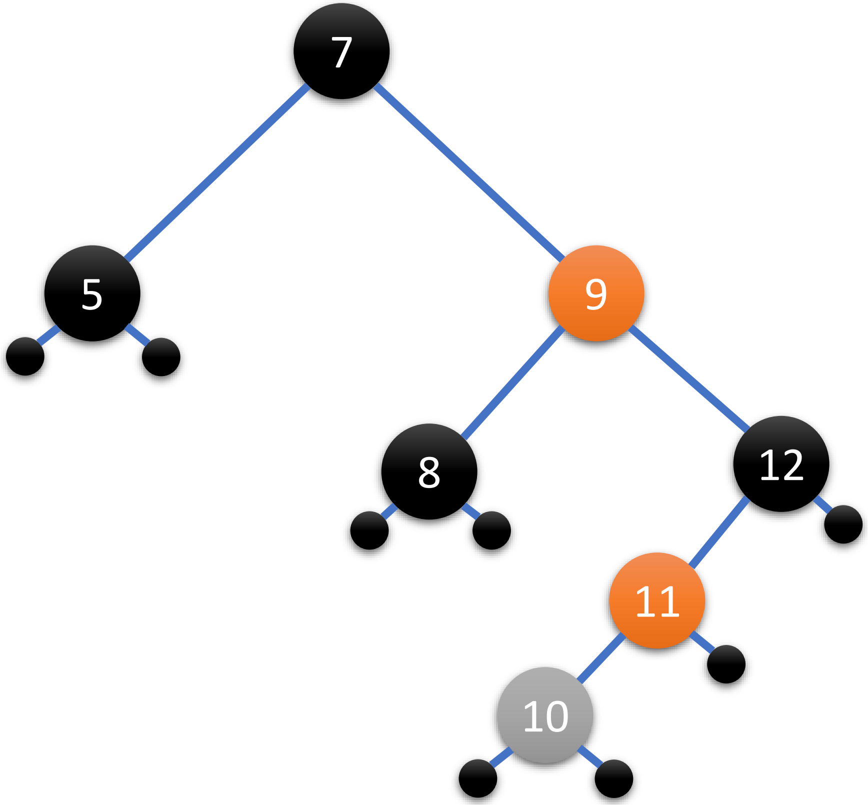
- Add a new node with y.key = 11
  - Insert y as in binary search tree
  - Set y.color ← **red**
  - Recolor the tree



Example

- Insertion:
  - Insert the new node as in binary search tree
  - Color the new node **red**

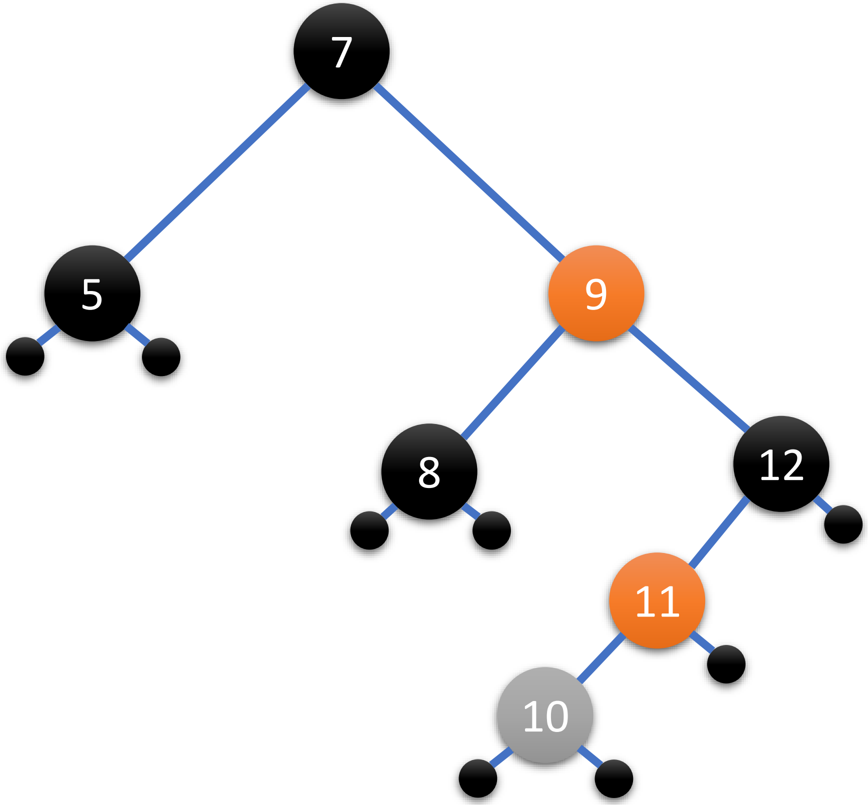
- Add a new node with z.key = 10
  - Insert z as in binary search tree
  - Set z.color ← **red**



Example

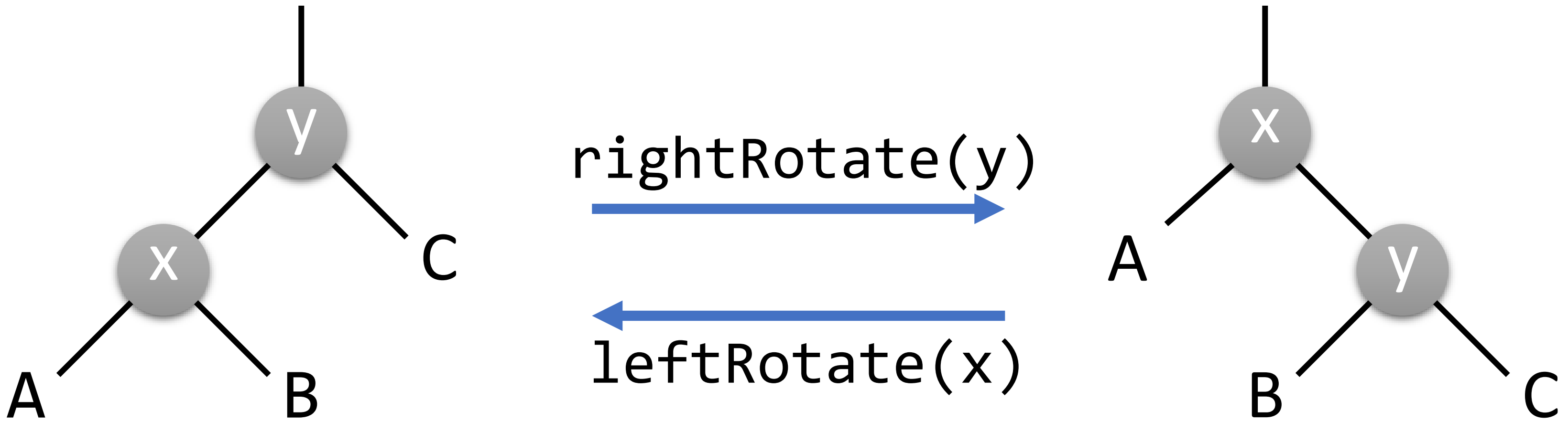
- Insertion:
  - Insert the new node as in binary search tree
  - Color the new node **red**

- Add a new node with z.key = 10
  - Insert z as in binary search tree
  - Set z.color ← ??
  - Tree is too unbalanced
  - Need to restructure
  - See how to deal with it next …



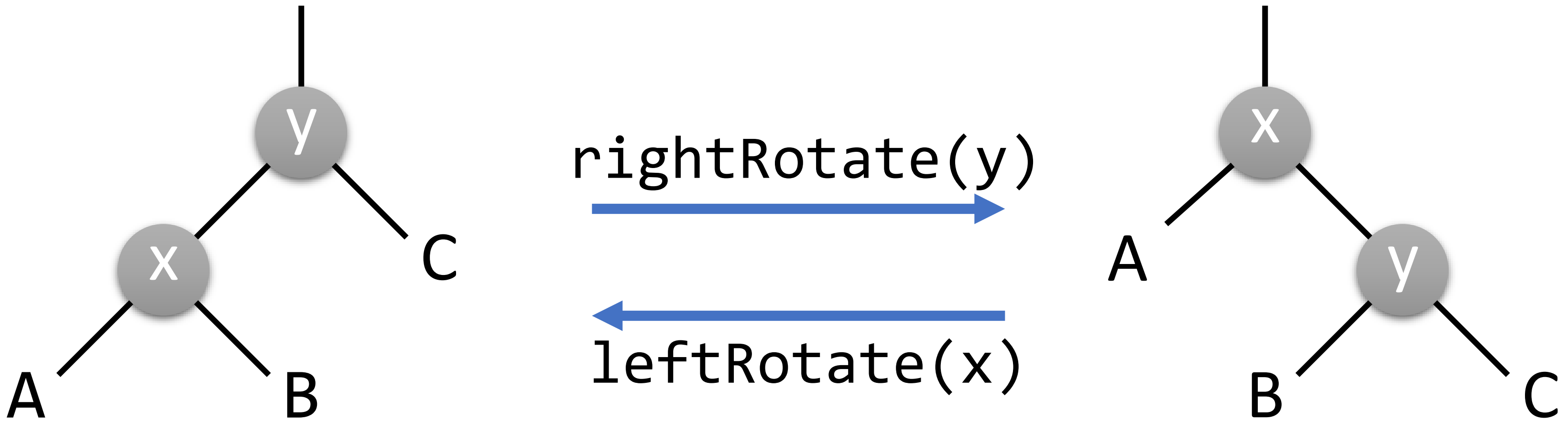
Tree Rotation

- Our basic operation for changing tree structure is called rotation
  - Rotation preserves inorder key ordering
  - How would tree rotation actually work?



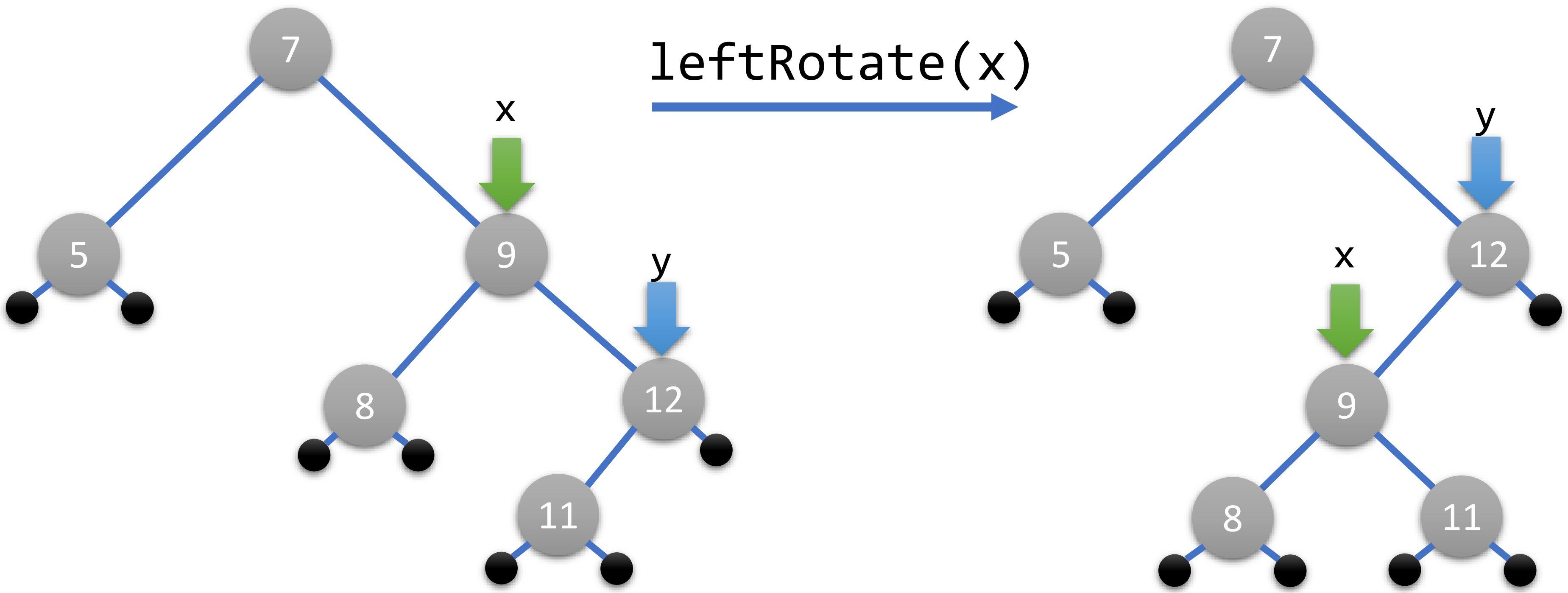
Tree Rotation

- Need a couple of pointer manipulations
  - x keeps its left child
  - y keeps its right child
  - x's right child becomes y's left child
  - x's and y's parents change



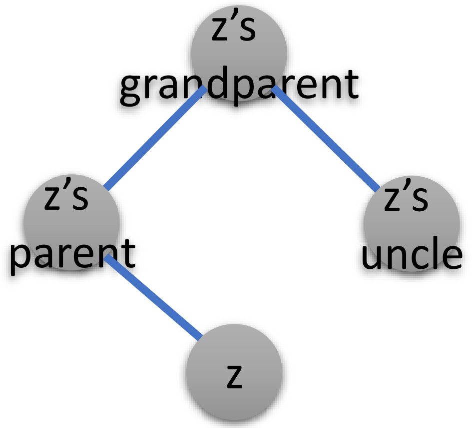
Example

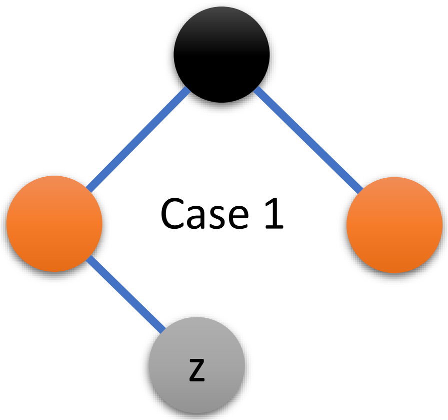
- Left Rotate at node 9

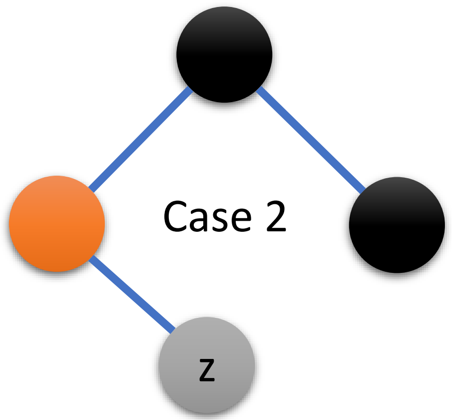


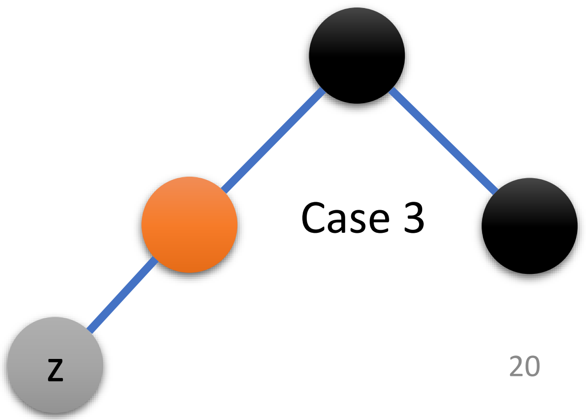
Three Possible Cases

- Suppose that the parent of the new node is the left child of its grandparent
  - "Uncle" is the other child of the grand parent
- Cases:
  1. New node z's uncle is **red**
  2. New node z is right child and its uncle is **black**
  3. New node z is left child and its uncle is **black**



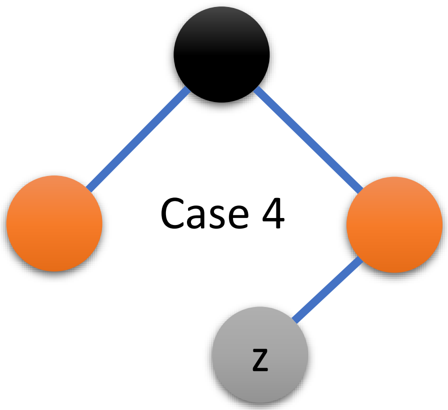


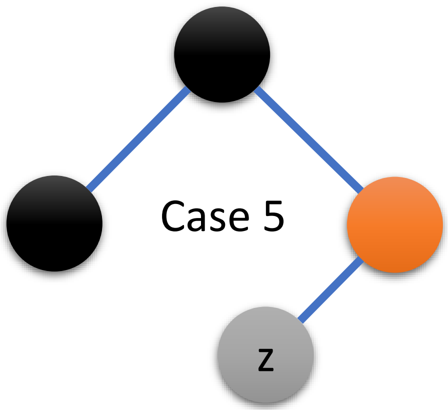


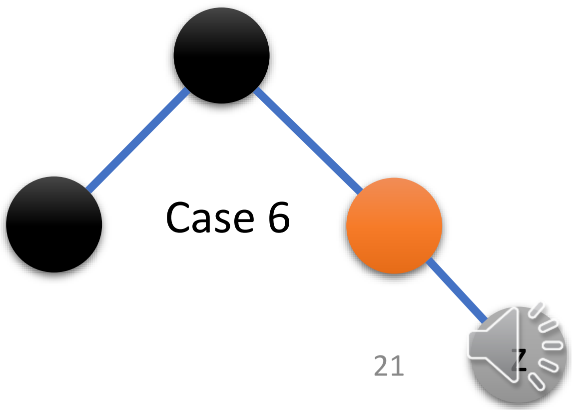


Mirrored Possible Cases

- Cases:
  1. New node z's (right) uncle is **red**
  2. New node z is right child and its uncle is **black**
  3. New node z is left child and its uncle is **black**
  4. New node z's (left) uncle is **red**
  5. New node z is left child and its uncle is **black**
  6. New node z is right child and its uncle is **black**



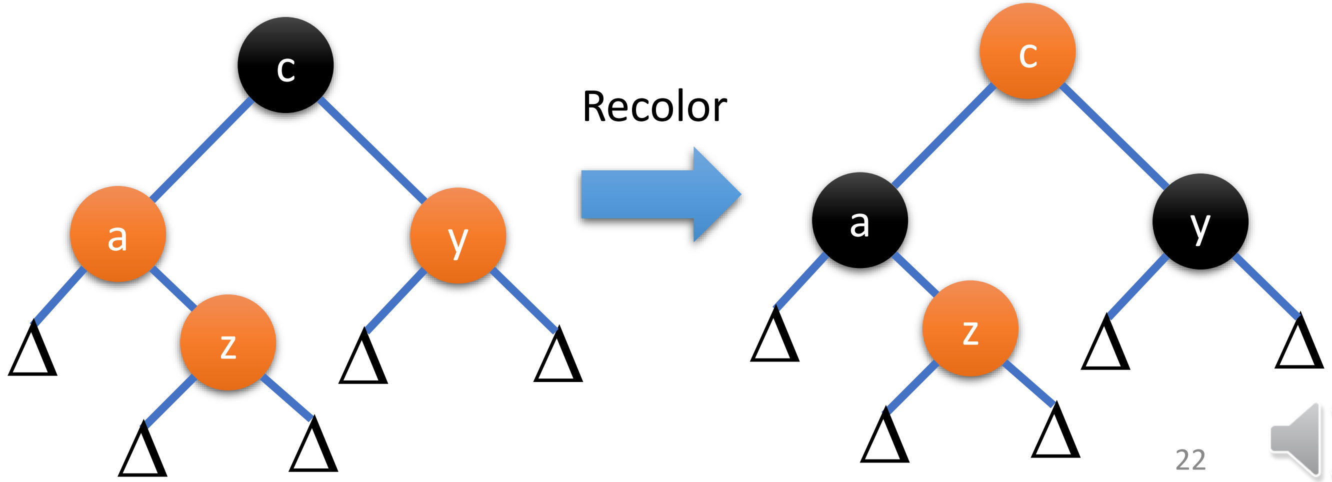




Red-Black Tree Insertion: Case 1

- Case 1: Uncle is red
  - Solution: Recoloring
    - Assume all subtree Δ's have equal black-height
    - Recolor parent, uncle y and grandparent to satisfy the property that all paths have equal black height
  - Note: If c is root, then reset the color to be black at the end of insertion

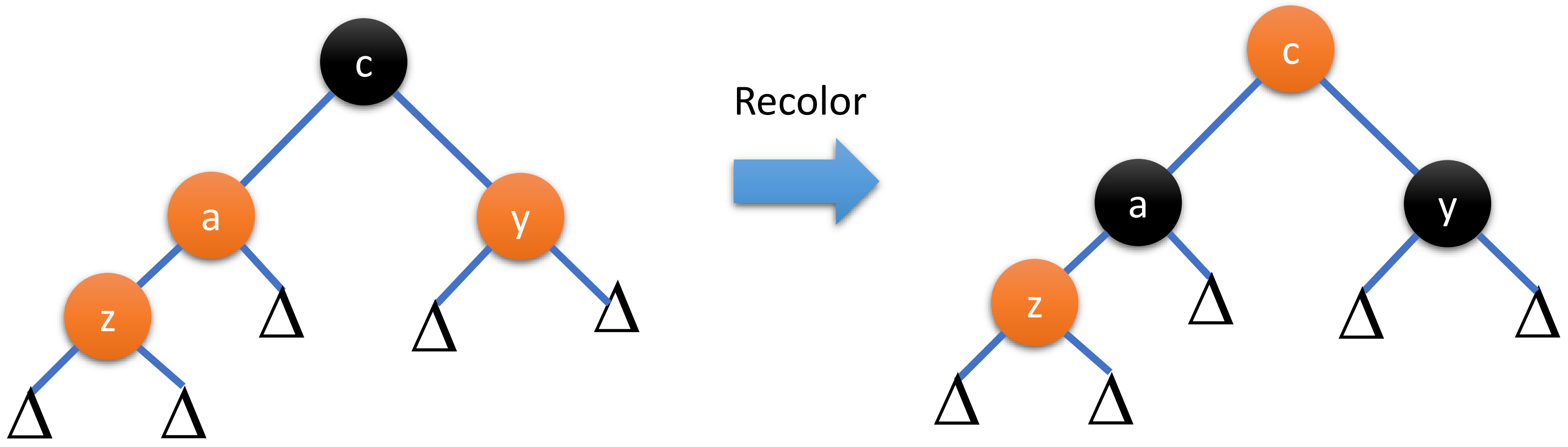
```
// Case 1
If (y.color = RED) Then
    z.parent.color ← BLACK
    y.color ← BLACK
    z.parent.parent.color ← RED
```



Red-Black Tree Insertion: Case 1

- Case 1: Uncle is red
  - Solution: Recoloring
  - New node z can be either left or right child

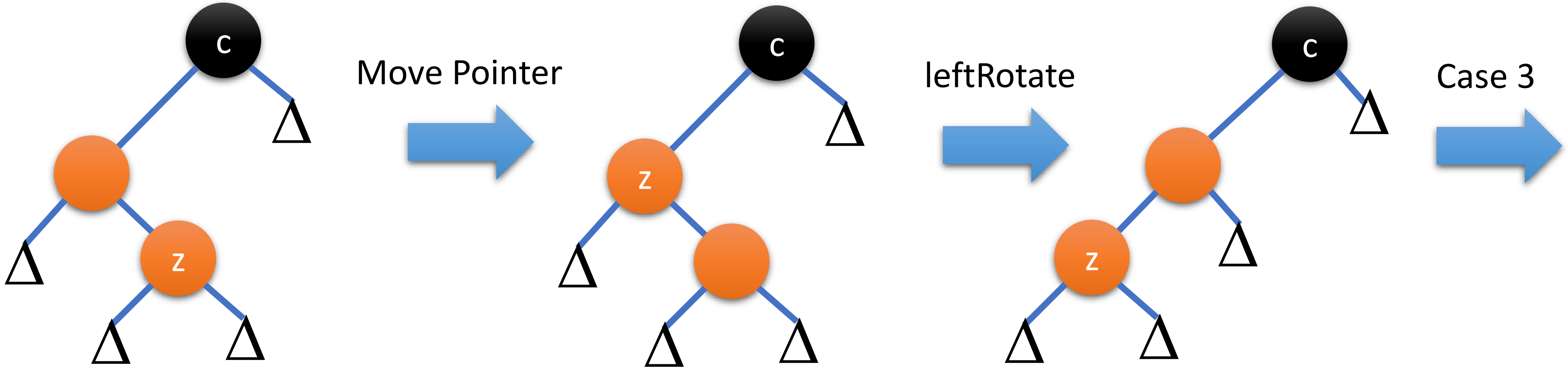
```
// Case 1
If (y.color = RED) Then
    z.parent.color ← BLACK
    y.color ← BLACK
    z.parent.parent.color ← RED
```



Red-Black Tree Insertion: Case 2

- Case 2: Uncle is black
  - New node z is right child
  - Solution: Transformation
    - Transform to case 3 via a left-rotation

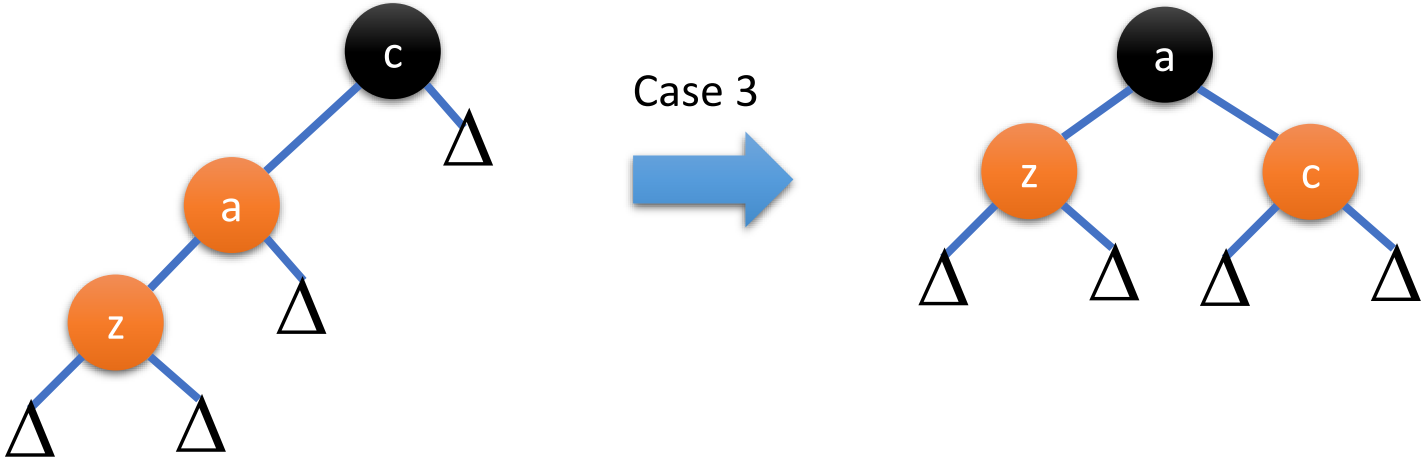
```
// Case 2
If (z = z.parent.right) Then
    z ← z.parent
    leftRotate(z)
// continue with Case 3
```



Red-Black Tree Insertion: Case 3

- Case 3: Uncle is black
  - New node z is left child
  - Solution: Rotation
    - Recolor and right rotate

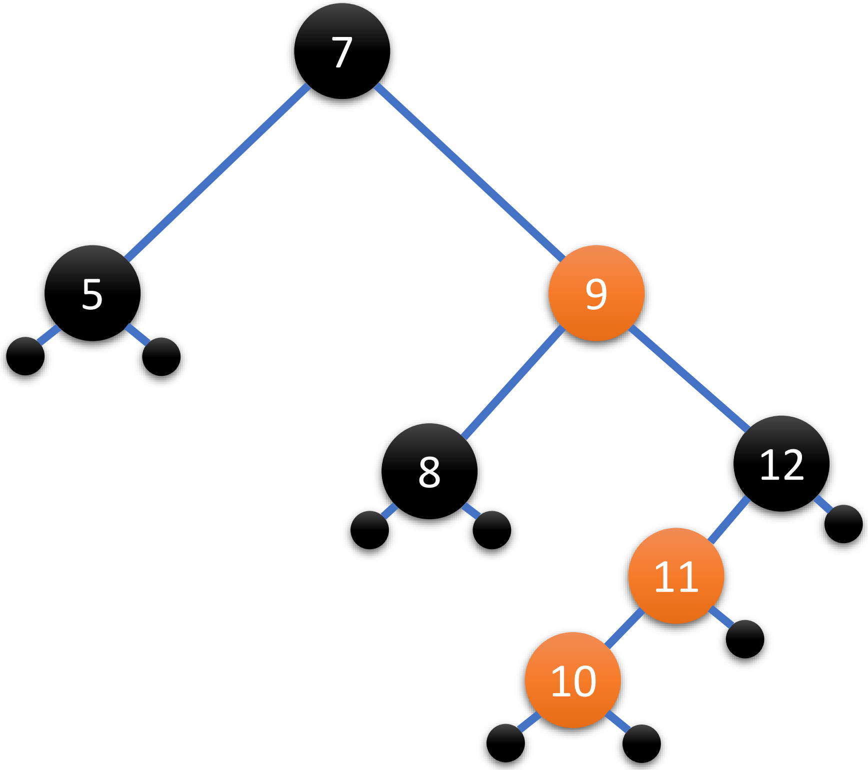
```
// continue with Case 3
z.parent.color ← BLACK
z.parent.parent.color ← RED
rightRotate(z.parent.parent)
```



Example

- Let us return to the example

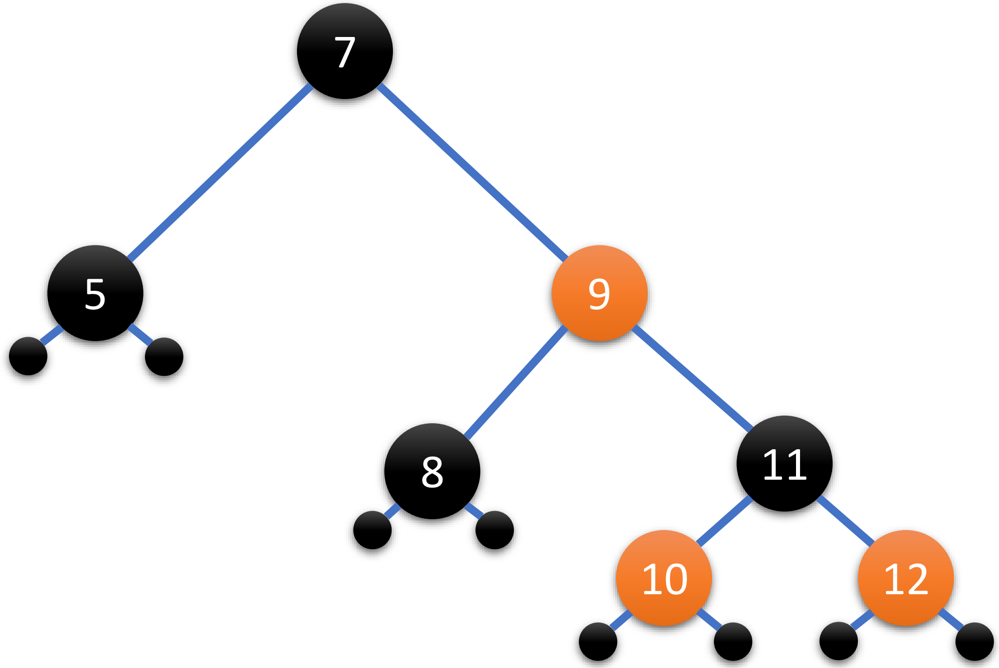
- Add a new node with z.key = 10
  - Which Case is it?
  - It is Case 3
    - Recolor and rotate right



Example

- Let us return to the example

- Add a new node with z.key = 10
  - Which Case is it?
  - It is Case 3
    - Recolor and rotate right



Red-Black Tree Insertion: Cases 4-6

- Cases:
  1. New node z's (right) uncle is **red**
  2. New node z is right child and its uncle is **black**
  3. New node z is left child and its uncle is **black**
  4. New node z's (left) uncle is **red**
  5. New node z is left child and its uncle is **black**
  6. New node z is right child and its uncle is **black**
- Cases 1-3 hold if z's parent is a left child of its grandparent
- If z's parent is a right child of its grandparent, Cases 4-6 are symmetric to Cases 1-3
  - Swap left for right, and vice versa

Red-Black Tree Insertion

```
RB-Insert[T,z]

// Call BST-Insert
BST-Insert[T,z]

z.left ← null; z.right ← null
z.color ← RED

// Call RB-InsertFixup to
// recolor and restructure tree
RB-InsertFixup[T,z]
```

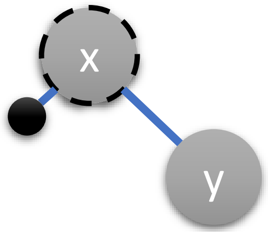
```
RB-InsertFixup[T,z]

While z.parent.color = RED Do
   If z.parent = z.parent.parent.left Then
      y ← z.parent.parent.right
      // Case 1
      If (y.color = RED) Then
         z.parent.color ← BLACK
         y.color ← BLACK
         z.parent.parent.color ← RED
         // Continue While Loop
         z ← z.parent.parent
      Else
         If (z = z.parent.right) Then
            // Case 2
            z ← z.parent
            leftRotate(z)
         // continue with case 3
         z.parent.color ← BLACK
         z.parent.parent.color ← RED
         rightRotate(z.parent.parent)
   Else
   // Cases 4-6 with "right" & "left" exchanged
T.root.color ← BLACK
```

Exercise

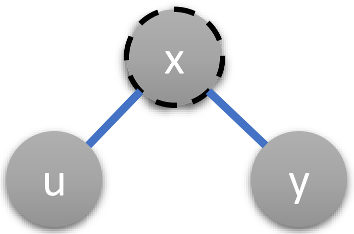
- Consider the following operations to red-black tree
  - Insert 12, 43, 34, 11, 44, 1
- What is the tree height of the final tree?
- How many **red** nodes are in the final tree?

Red-Black Tree Deletion

- x is the to-be-deleted node (disregarding the colors):
- Cases:
  1. x has only one child
     - Just remove x and replace x by its child
     - If the red-black property #3 is now broken, recolor y in black keeping the black-height, since x was definitely **black** (as its parent is red)



  2. x has two children

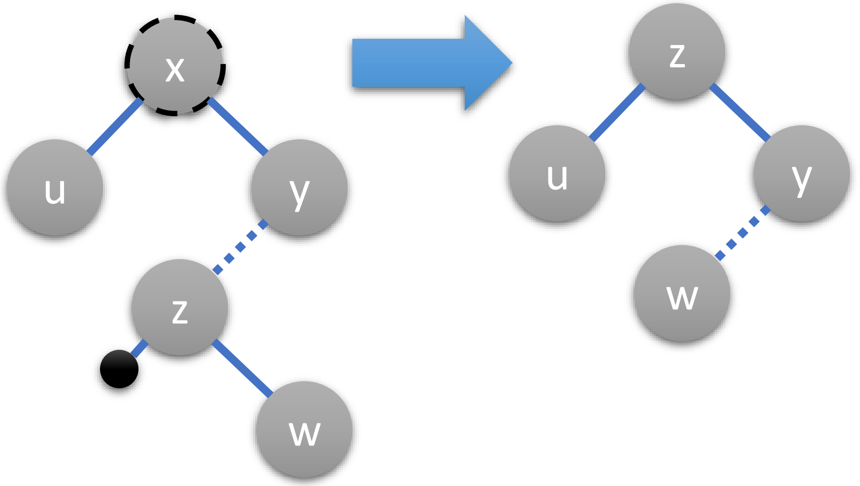


Red-Black Tree Deletion

NOT in exam!

- Cases:

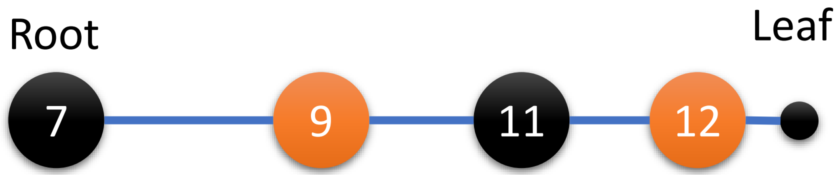
2. x has two children, x's successor is z
   - Replace the successor z by left child w
   - Remove x and replace x by its successor z
   - But if x is **red** and z is **black**, then there will be an extra black node in z's new left child



- See Chapter 14 of "Introduction to Algorithms" (by Cormen, Leiserson and Rivest)

Black-height

- The red-black properties:
  1. Every node is either **red** or **black**
  2. Root and leaves (NULL node) are **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

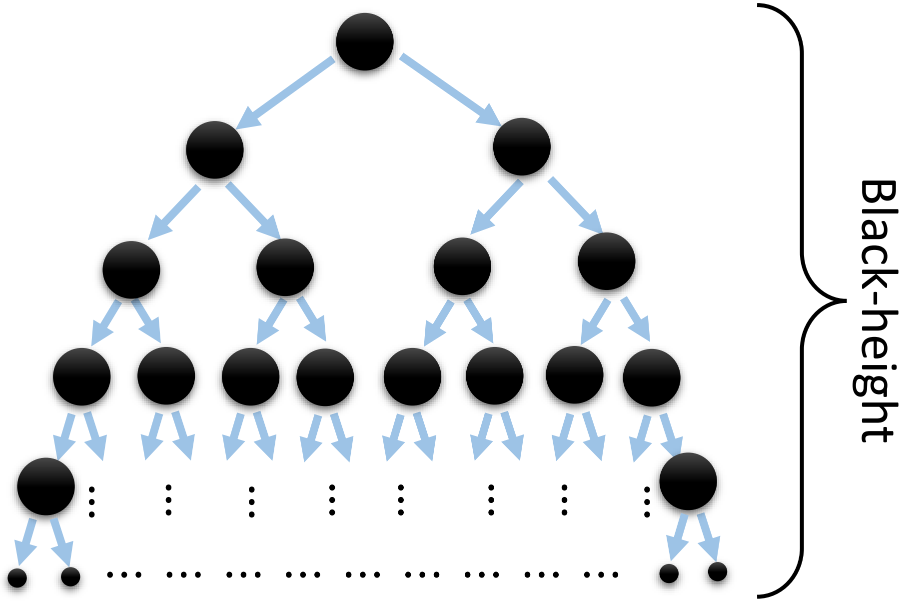


- Let black-height (BH(x)) be the number of black nodes from x to a leaf
  - BH(x) is independent of any leaf node
- Note: No adjacent nodes on any path on red-black tree can be both **red**
  - Otherwise, it violates red-black property #3

Quick Fact

- Note that Black-height $\leq$ tree height
- Let the "internal" nodes be non-leaf nodes
- Consider the extreme case:
  - All nodes are **black** (ignoring all red nodes)
  - The tree is perfectly balanced
    - Check: satisfy all red-black properties
  - Total number of **black** internal nodes is

    $$2^0+2^1+2^2+ \ldots + 2^{BH(root)-1} = 2^{BH(root)} - 1$$
  - Hence,
    - Total number of internal nodes $\geq 2^{BH(root)} - 1$
    - Since adding red internal nodes back will not affect the black-height



Red-Black Tree: Worst-case Running Time

- What is the minimum black-height of the root with height h?
  - Answer: $BH(root) \geq h/2$
    - Because the number of **black** nodes from the root to a leaf is at least $h/2$
    - Otherwise, two **red** nodes will be adjacent to each other

Theorem: A red-black tree with n internal nodes has height $h \leq 2 \log(n + 1)$

Proof: Note that $n \geq 2^{BH(root)} - 1$

Since $BH(root) \geq h/2$, we obtain

$$n \geq 2^{BH(root)} - 1 \geq 2^{h/2} - 1$$

$$\Rightarrow \quad n \geq 2^{h/2} - 1$$

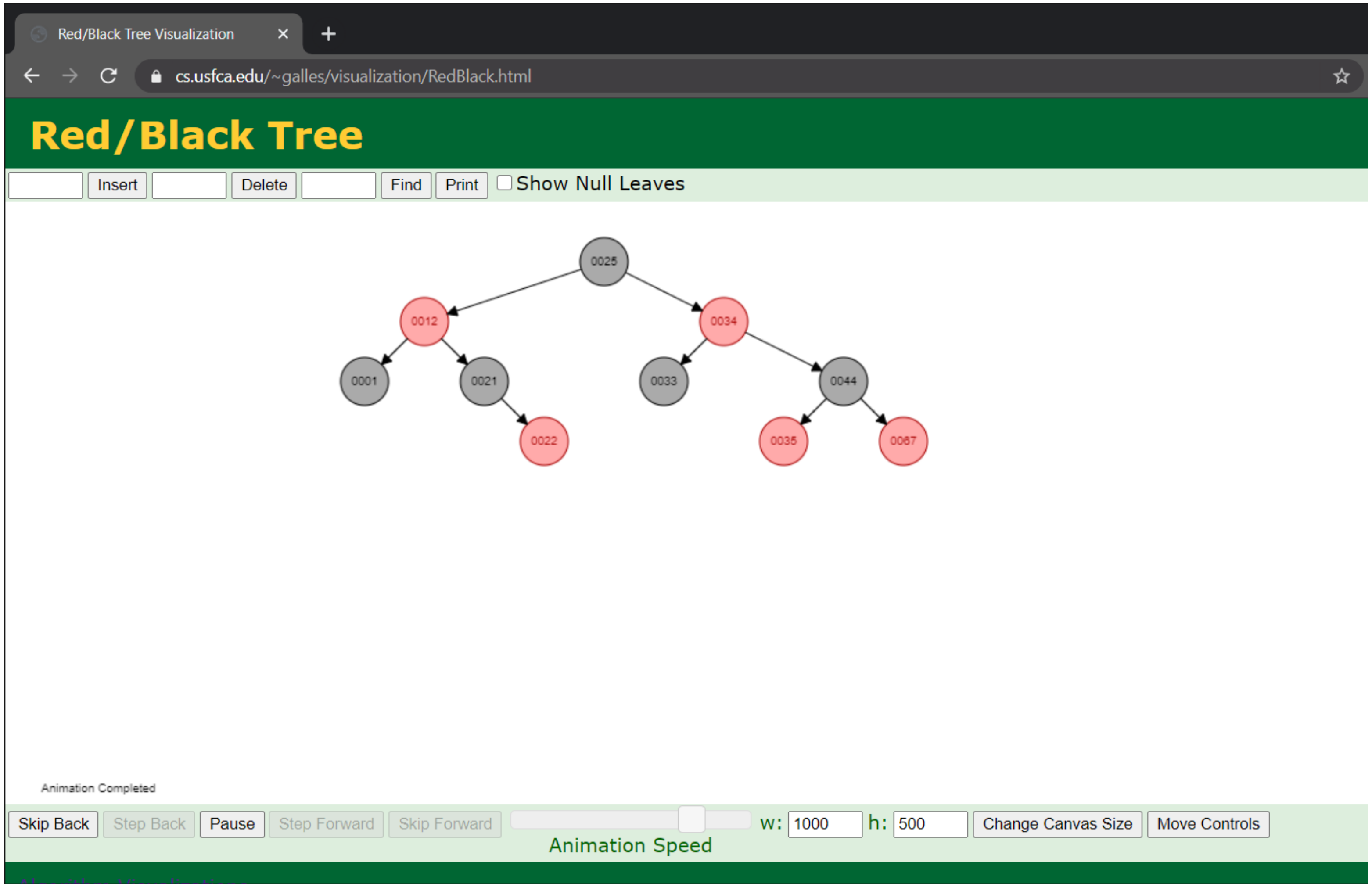
Then, $\log(n + 1) \geq h/2 \;\Rightarrow\; h \leq 2 \log(n + 1)$

Therefore, $h = O(\log(n))$

Red-Black Tree: Worst-case Running Time

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

Demo



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

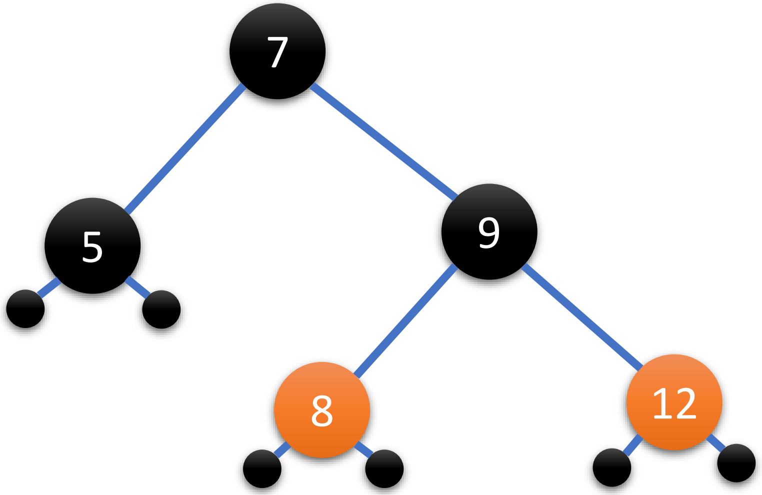
Implement a Red-Black Tree

```java
public class RBTree <T extends Comparable<T>> {

    private void insert(Node<T> x) { ... }

    public void rotateLeft(Node<T> x) { ... }

    public void rotateRight(Node<T> x) { ... }

    public Node<T> search(T key) {
}
```

```java
public class Node<T> {

    Colour colour;   // Node colour
    T key;           // Node value
    Node<T> parent; // Parent node
    Node<T> left, right;   // Child nodes
}
```

Summary

- Balanced search tree
  - Red-black tree
    - Red-black properties
    - Insertion
    - Black-height



Reference

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

- Reference:
  - Chapter 14 in Introduction to Algorithms (by Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest)