Data Structure & Algorithms

Lec 13: Binary Search Trees
Red-Black Trees

Fall 2017 - University of Windsor
Dr. Sherif Saad

Learning Objectives

1. Introduce advanced and efficient data structures
2. Learn the implementation techniques of fundamental Red-Black Tree operations
3. Learn to analyze and compare the time and space complexity of different data structures

Learning Outcomes

By the end of this class you should be able to:

- Distinguish between an AVL and Red-Black Tree.
- Explain the characteristics of Red-Black Tree.
- Explain the basic operations and algorithms related to the Red-Black Tree
- Describe possible applications and usages of the Red-Black Tree.

Outlines

1. Motivation & Background
2. Red-Black Tree Definition & Operations
3. Red-Black Tree Rotation
4. Red-Black Tree Insertion
5. Red-Black Tree Deletion

Motivation

Create a dynamic tree structure that support the basic dynamic-set operations in O (log n)

History

Proposed by Robert Sedgewick in late 1972

Robert Sedgewick derived the new data structure (Red-Black Trees) from the symmetric binary B-tree.

B-tree is a generalization of a binary search tree in that a node can have more than two children, was invented in 1971

Where does the term “Red-Black Tree” come from?

Red-Black Tree vs. AVL Tree

- Both are height-balanced binary tree data structures
- AVL is more balanced than the Red-Black Tree
- Searching an AVL tree is ***O (1.44 * log(n+2))***
- Searching a Red-black tree is ***O (2 * log(n+1))***
- The Red-Black tree has fixed number of operations to balance the tree
- AVL Tree is prefered for look-up intensive applications
- Red-Black Tree is prefered when insertion and deletion operations are frequent

Red-Black Tree: definition

What is a Red-Black Tree?

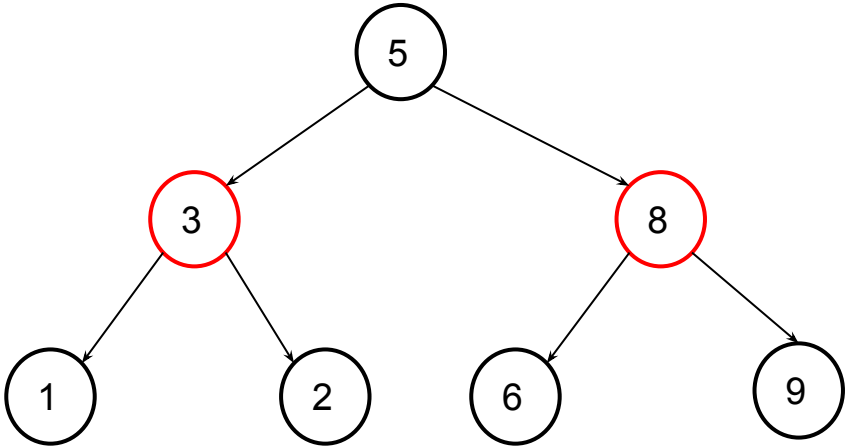
- Is a self-balanced binary search tree (approximately balanced)
- Each node has one extra bit of storage to store its color [Red or Black]
- No simple path from the root to a leaf is more twice as long as any other
- The height is O(log n), where n is the number of nodes in the tree.
- Each node of the tree now contains the attributes color, key, left, right, and parent.

Red-Black Tree: Properties (rules)

1. The root is black
2. Every red node has a black parent
3. For each node, all simple paths from the node to descendant leaves contain the same number of black nodes.
4. Every node is either red or black.
5. Every leaf (null) is black

Red-Black Tree: Example

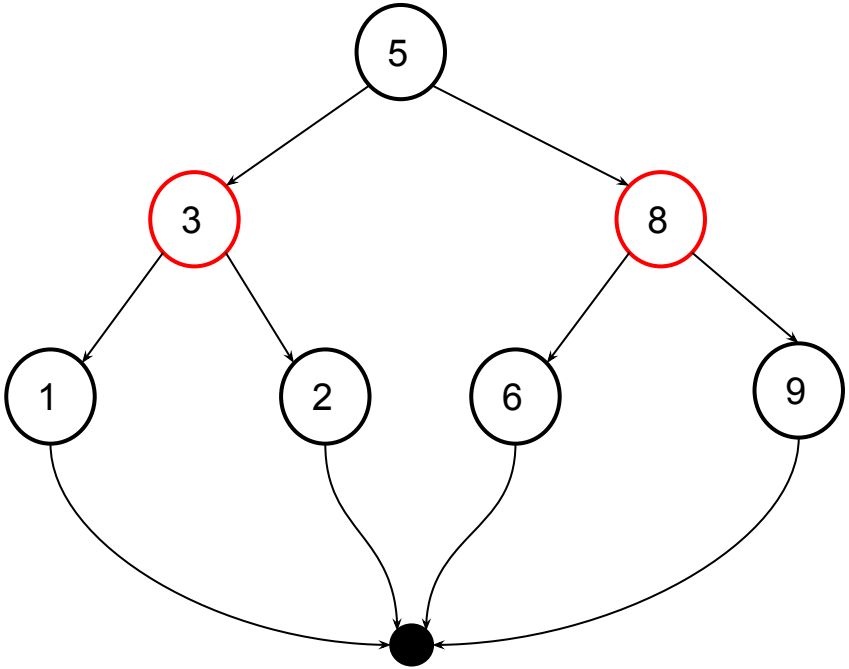
1. The root is black
2. Every red node has a black parent
3. For each node, all simple paths from the node to descendant leaves contain the same number of black nodes.
4. Every node is either red or black.
5. Every leaf (null) is black



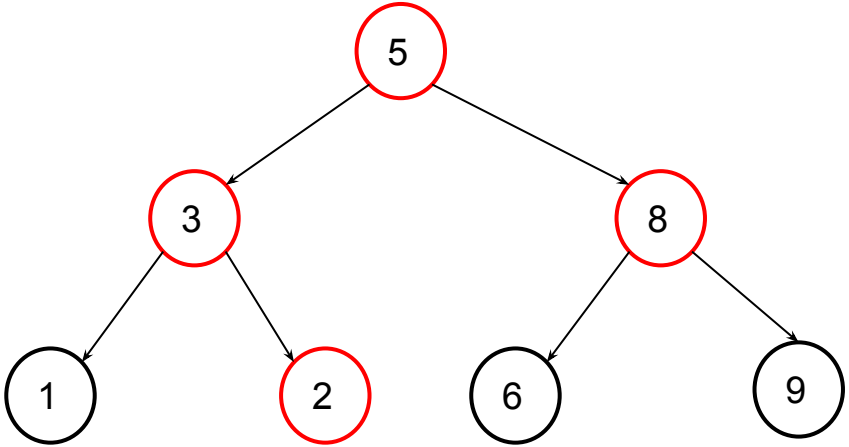
Red-Black Tree: Option

We can create special node (null) with black color to represent all the leafs (null) in the tree.

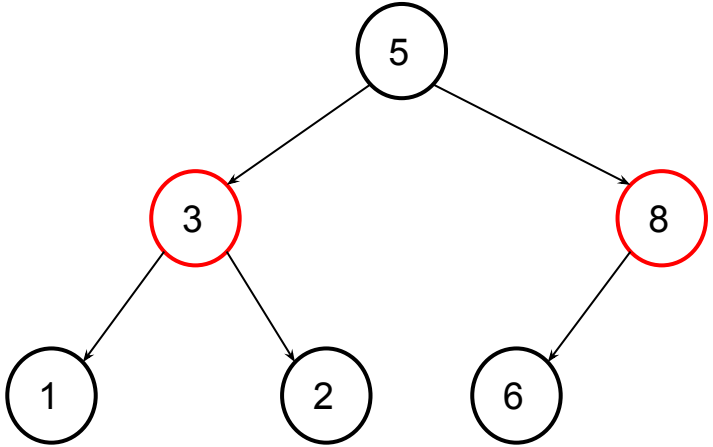
We use the sentinel so that we can treat a NIL child of a node x as an ordinary node whose parent is x.



Checkpoint: Find the red-black tree if any exist?



A



B

Red-Black Trees: Basic Operations

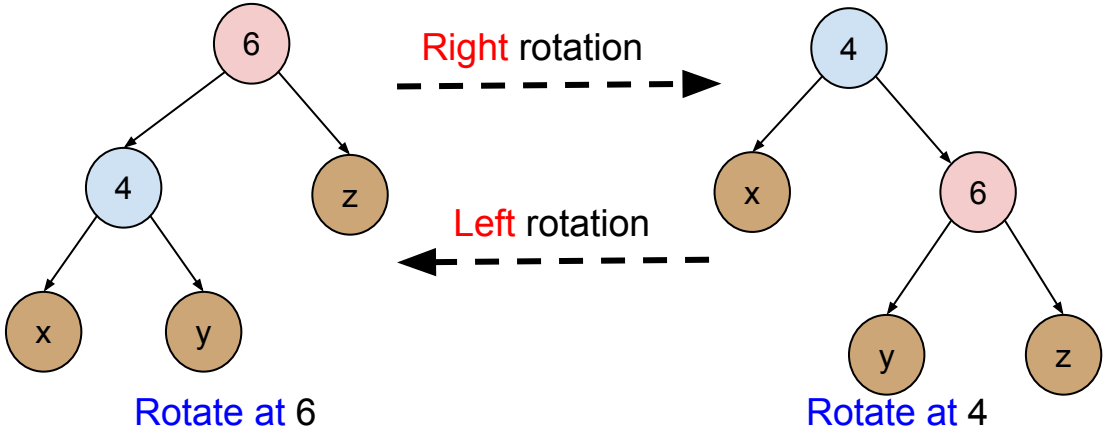
Here is a list of the basic operations we can perform over a red-black tree:

1. Search (find a key)
2. Find Minimum
3. Find Maximum
4. Find Predecessor
5. Find Successor
6. Insert a new key
7. Delete an existing key

Red-Black Tree: Maintenance

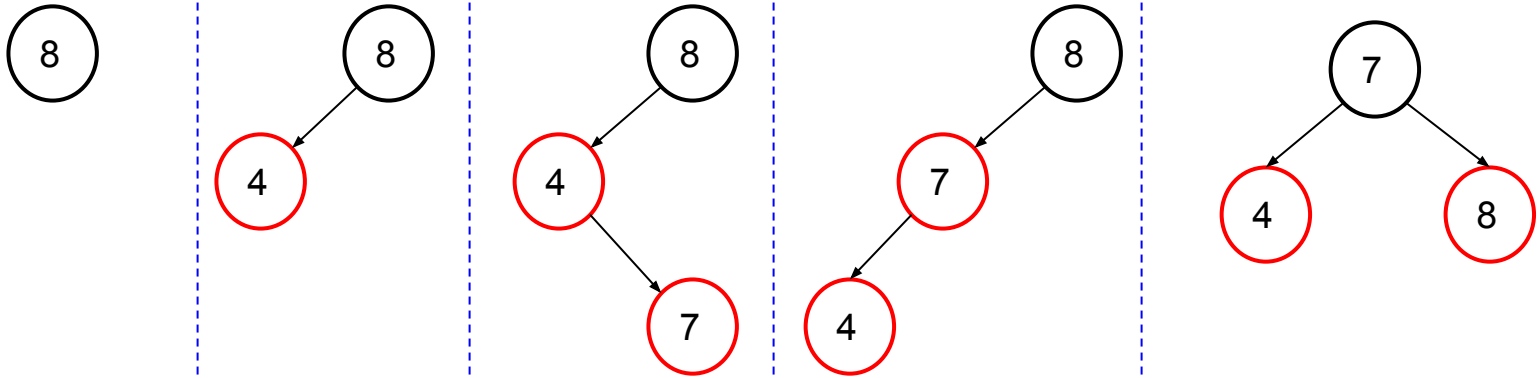
- The insert and the delete operations over the red-black tree modify the tree structure, which may violate the red-black tree rules (properties).
- To maintain the red-black tree properties we use two simple operations recoloring and rotation.

Red-Black Tree: Rotation



Red-Black Tree: Rotation Example

Let us assume we want to build a red-black tree of the set {8, 4, 7} of keys



[a] Set root

[b] insert red child

[c] violate rule # 2

[d] left rotate

[e] right rotate

Red-Black Tree: Rotation Summary

- Left Rotation on x, makes x the left child of y, and the left subtree of y into the right subtree of x.
- Right Rotation on y, makes y the right child of x, and the right subtree of x into the left subtree of y.
- Right-Rotate is symmetric to Left -Rotate: exchange left and right everywhere (Try to write it yourself)
- The rotation cost is O(1) for both Left-Rotate and Right-Rotate, since a constant number of pointers are modified.

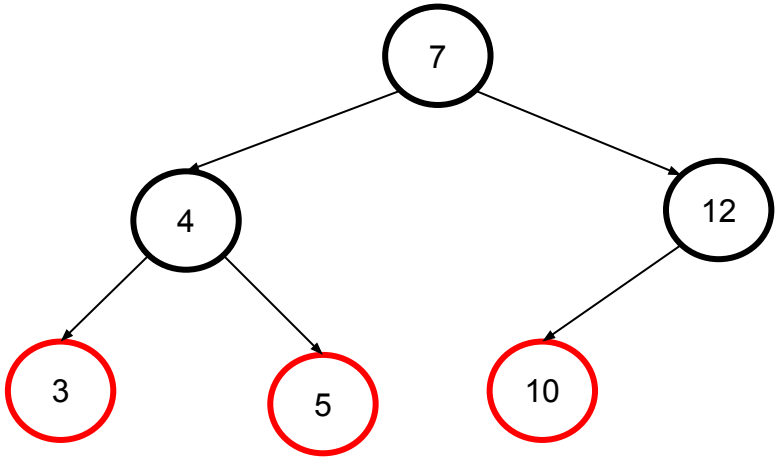
Red-Black Insertion

Red-Black Tree: Insertion

- Maintain the red-black tree properties after inserting a new key (node)
- What color should we use for the new inserted node Red or Black?
- The insertion operation is the same as the insertion operation for the regular BST (binary search tree)
- Check for red-black tree property violation and if any apply recoloring and rotation to fix-up the violation

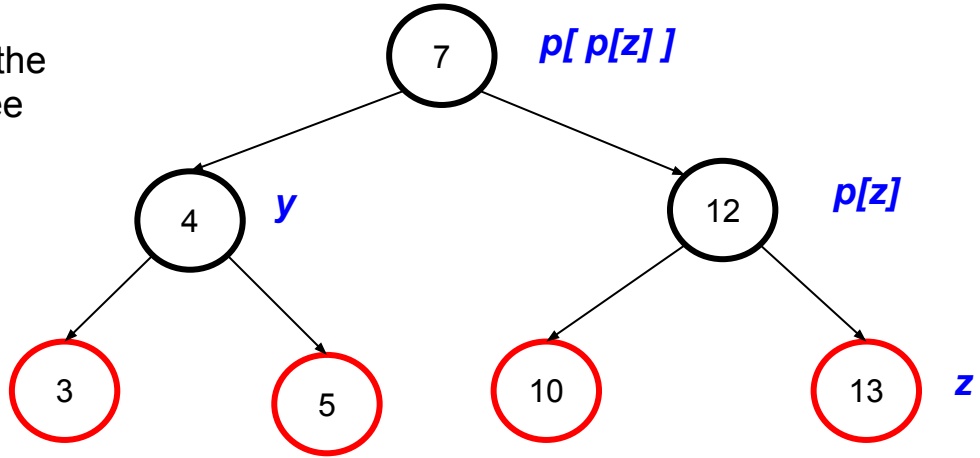
Red-Black Tree: Insertion

Insert 13



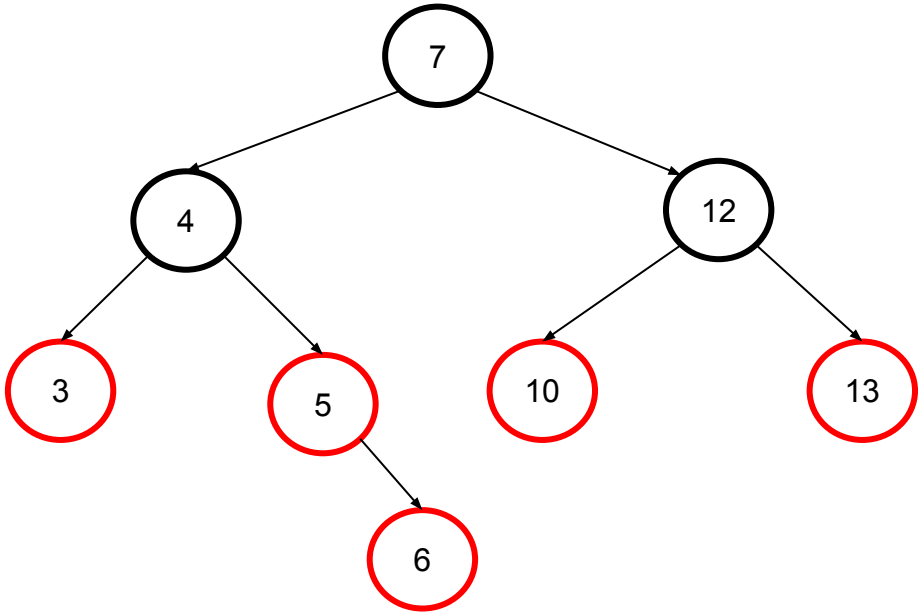
Red-Black Tree: Insertion

13 is inserted and the result is a red-black tree



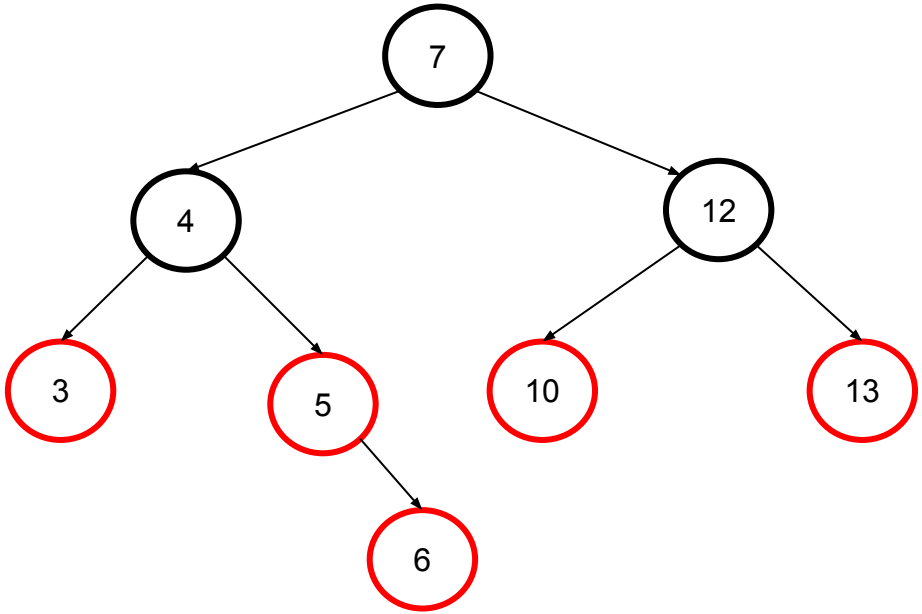
Red-Black Tree: Insertion

Insert 6, the result is not red-black tree



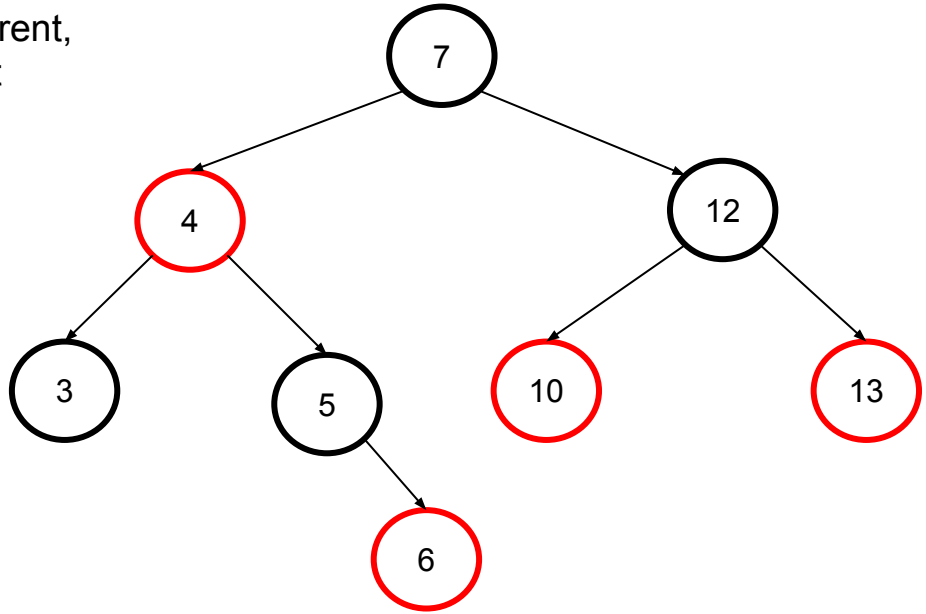
Red-Black Tree: Insertion

Flip 6 parent, uncle and grandparent



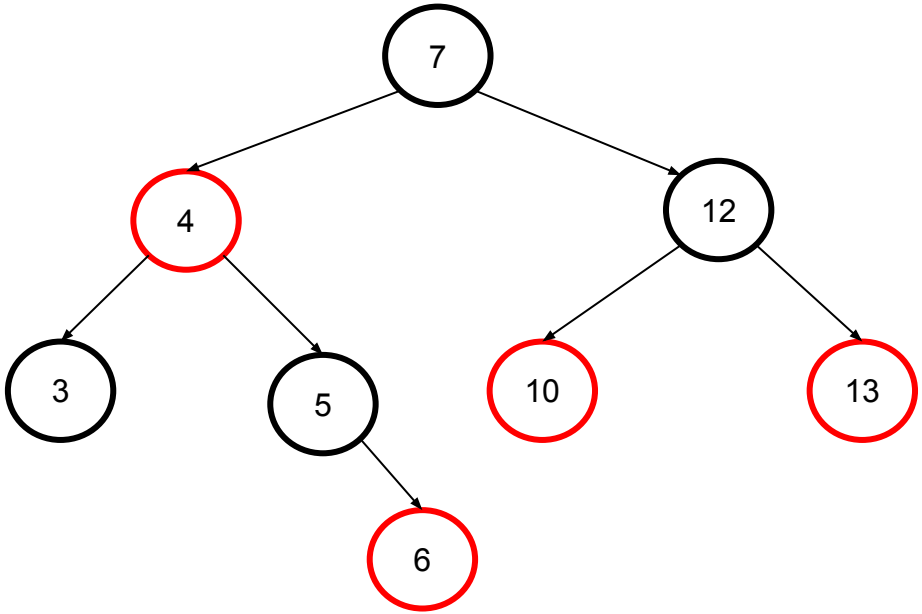
Red-Black Tree: Insertion

Flip the colors of 6 parent, uncle and grandparent



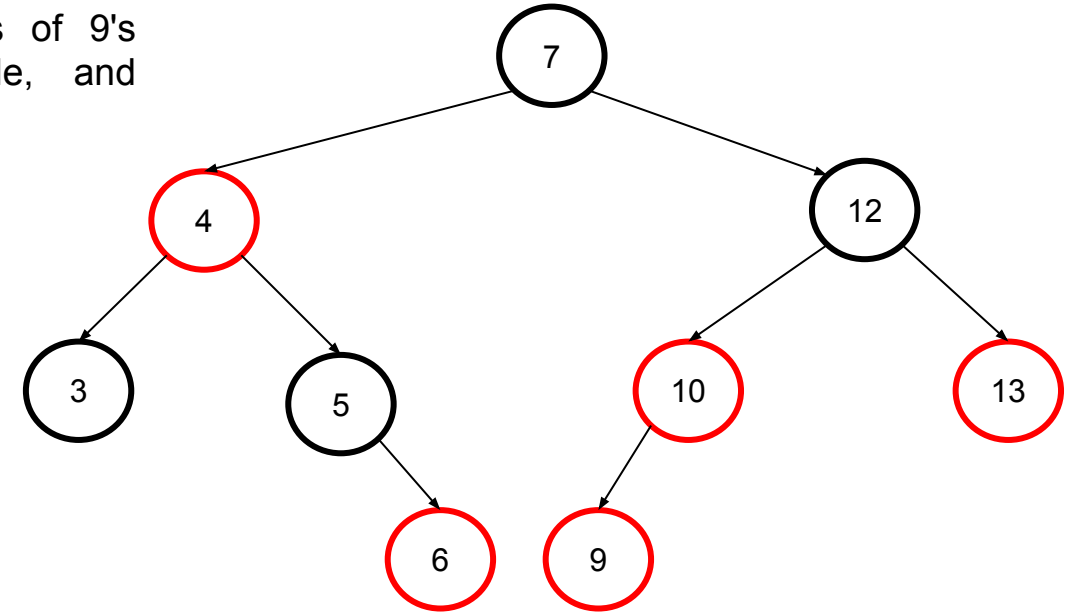
Red-Black Tree: Insertion

Insert 9



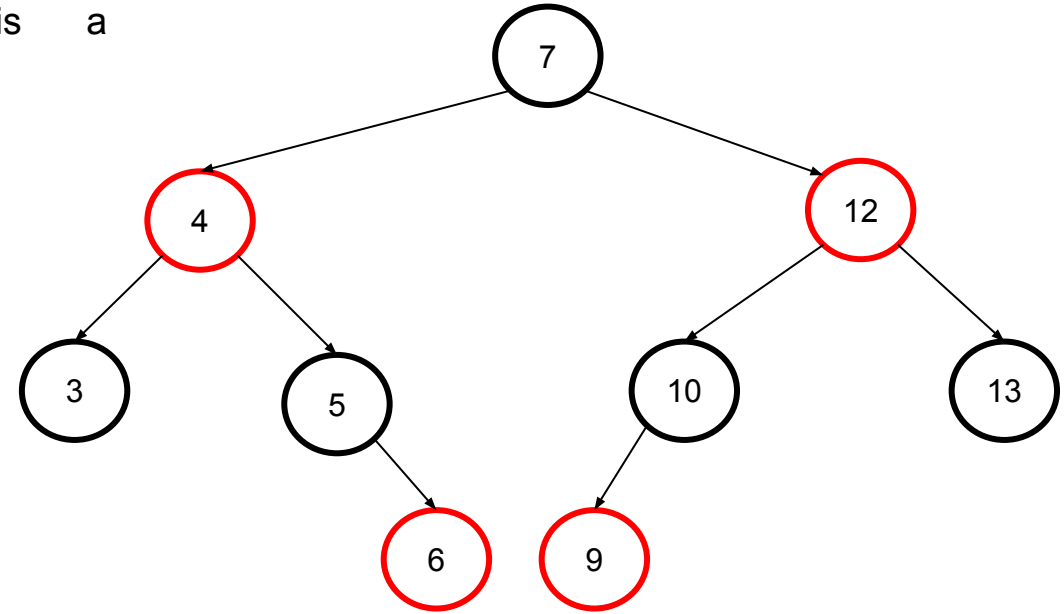
Red-Black Tree: Insertion

Flip the colors of 9's parent , uncle, and grandparent.



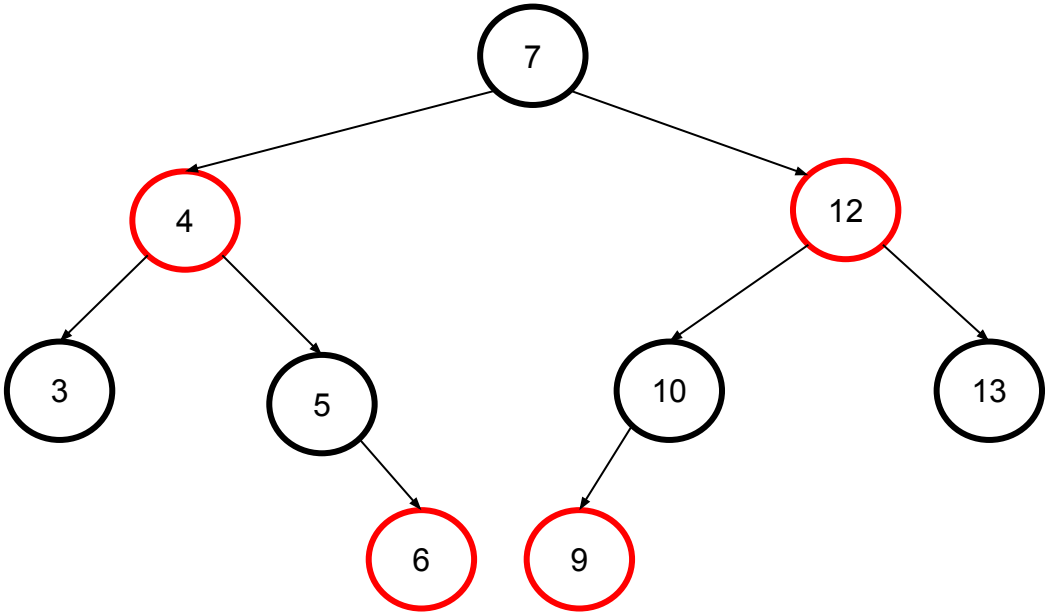
Red-Black Tree: Insertion

The result is a red-black tree



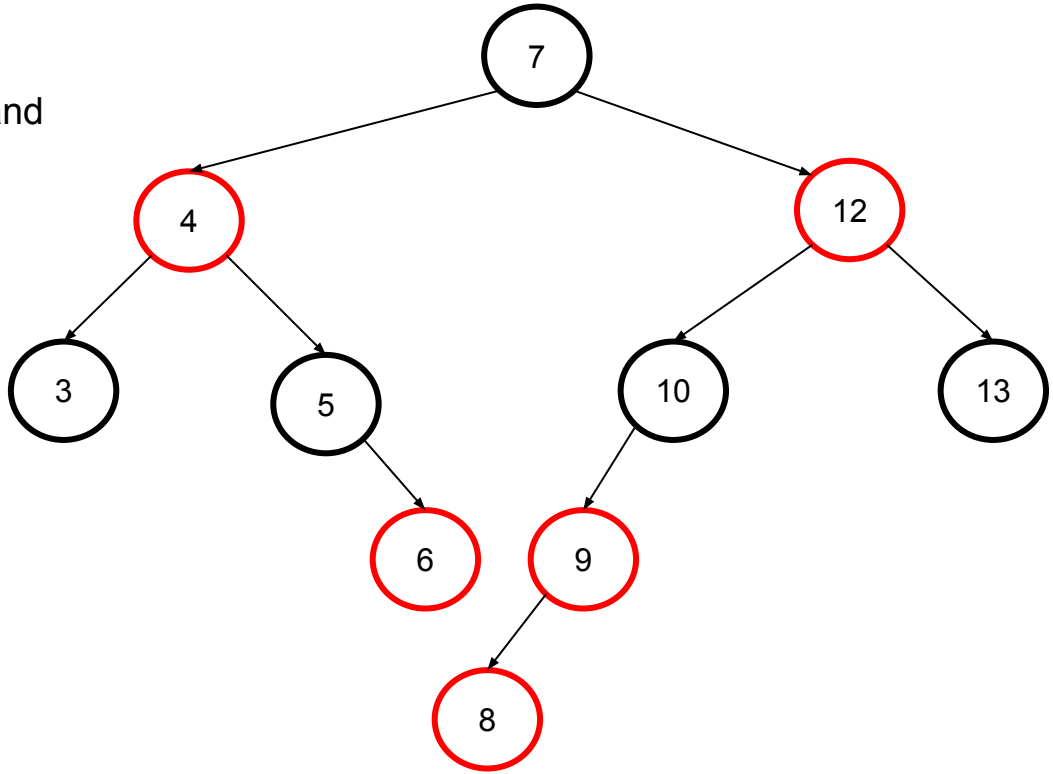
Red-Black Tree: Insertion

Insert 8



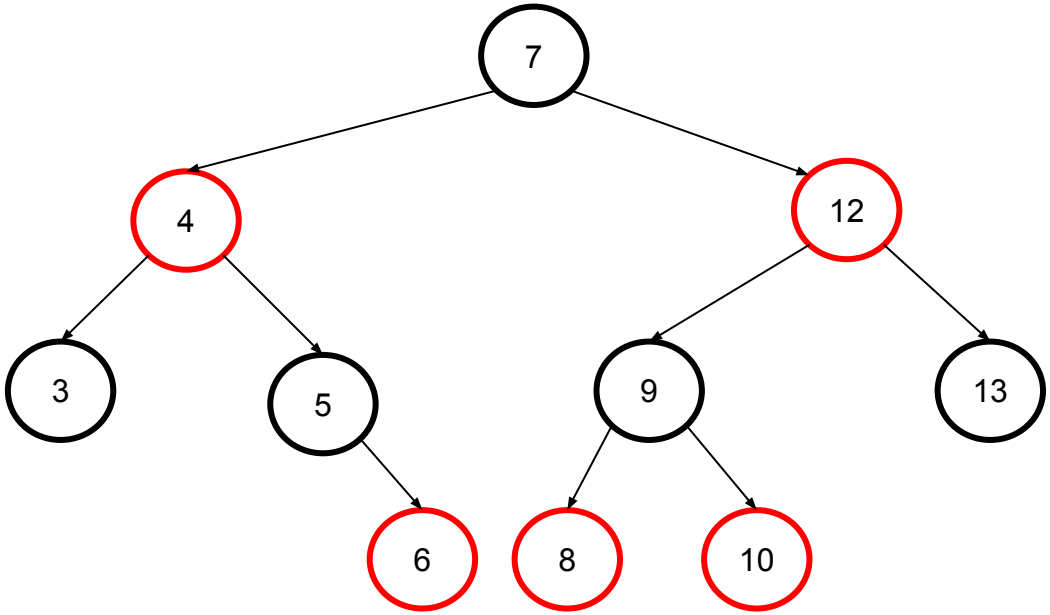
Red-Black Tree: Insertion

Insert 8, red clid and red parent



Red-Black Tree: Insertion

Insert 8



Red-Black Tree: Insertion

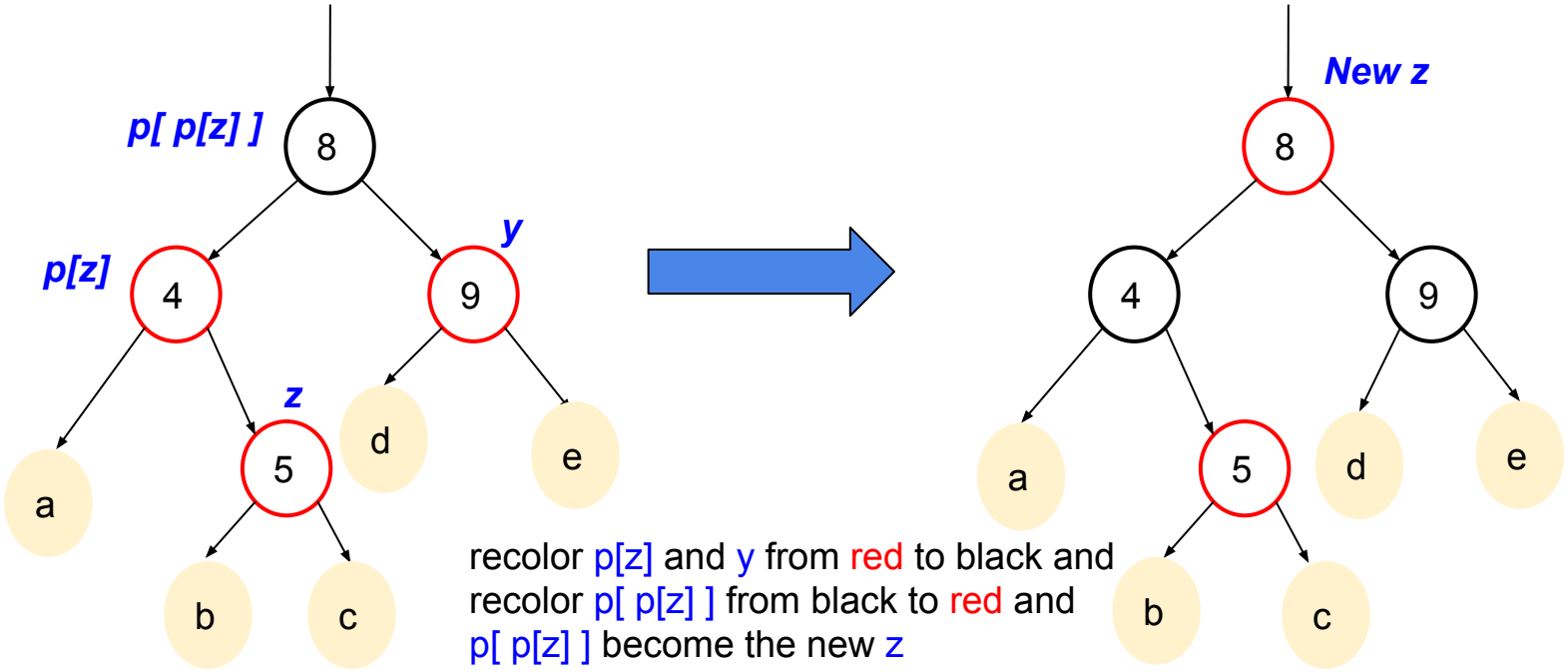
After the insertion we may have one pair of consecutive reds nodes.

To fix-up this issue we rotate it up the tree and away.

There are three cases we have to handle. Well there are 6 case , but 3 of which are symmetric to the other.

Red-Black Tree: Insertion Case 1

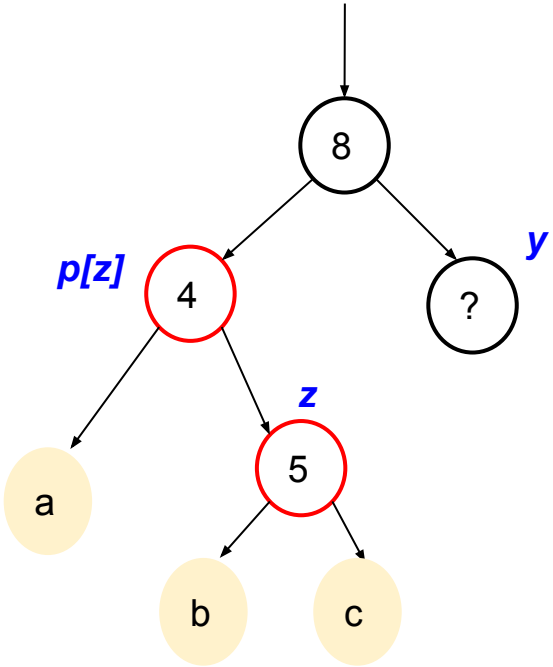
Case 1: Uncle y is red



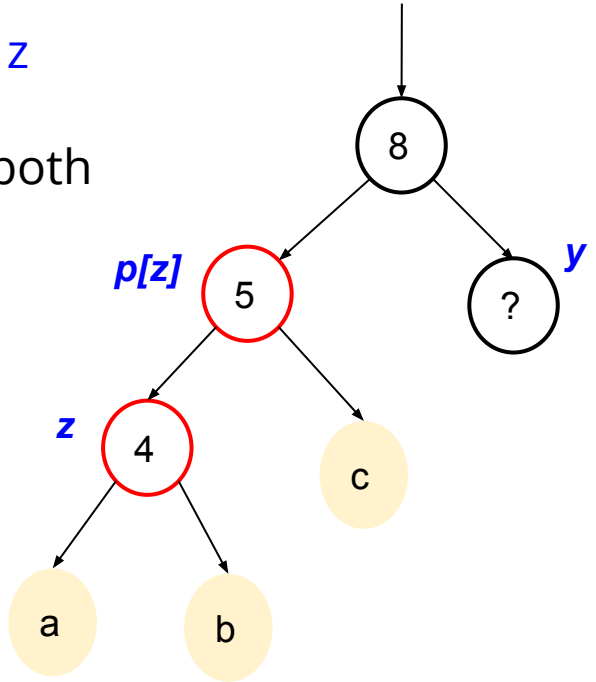
recolor p[z] and y from red to black and
recolor p[p[z]] from black to red and
p[p[z]] become the new z

Red-Black Tree: Insertion Case 2

Case 2: Uncle y is black and z is a right child

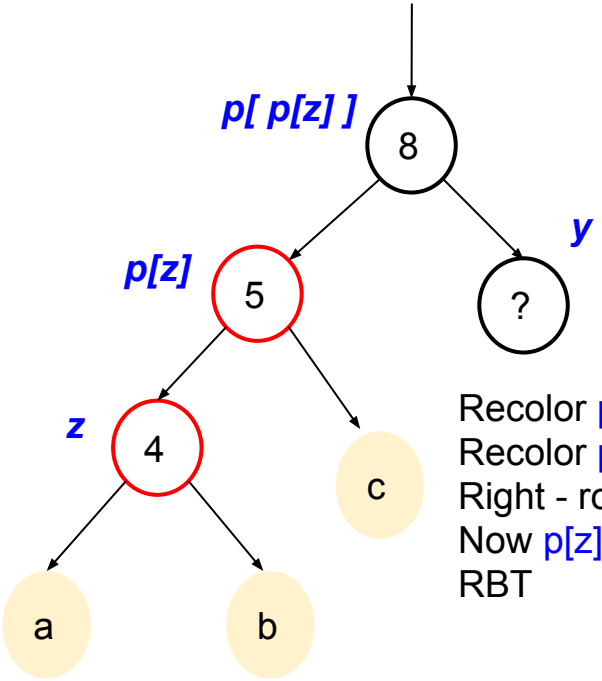


Left-rotate on p[z] where z and p[z] switch roles.
Now z is a left child and both z and its parent are red

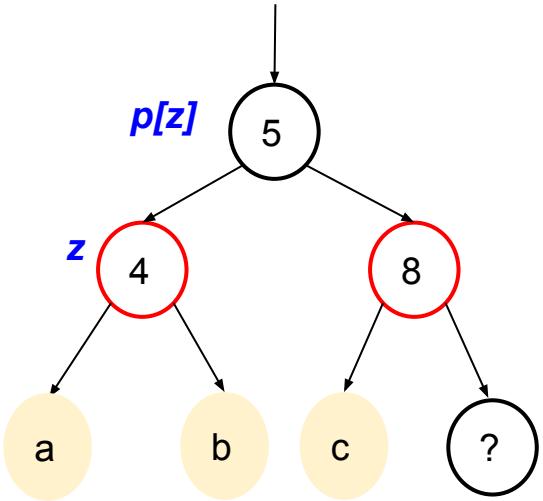


Red-Black Tree: Insertion Case 3

Case 3: Uncle y is black and z is a left child



Recolor p[z] from red to black
Recolor p[p[z]] from black to red
Right - rotate on p[p[z]]
Now p[z] is black the tree is a valid RBT



Insertion- Fixup: Algorithm Analysis

- At the beginning of each loop, z is red.
- At most there is one red-black violation
  - The root is red because z is the root (rule 1)
  - The parent of z is red and z is red (rule2)
- The algorithm will terminate when p[z] is black, then rule 2 is fine
- The last line ensures that rule 1 is maintained (root is black)
- All the 3 cases we discussed p[z] was a left child, what happen when is is a right child.
- In fact there are 6 cases not 3, but the other 3 cases are symmetric.

Red-Black Tree: Insertion- Fixup Algorithm

```
1.  algorithm InsertFixup
2.    while color[p[z]] = RED
3.      if p[z] = left[p[p[z]]]
4.          then y ← right[p[p[z]]]
5.               if color[y] = RED
6.                  then color[p[z]] ← BLACK   // Case 1
7.                       color[y] ← BLACK      // Case 1
8.                       color[p[p[z]]] ← RED  // Case 1
9.                       z ← p[p[z]]           // Case 1
10.              else if z = right[p[z]]       // color[y] != RED
11.                 then z ← p[z]              // Case 2
12.                      LEFT-ROTATE(T, z)     // Case 2
13.                 color[p[z]] ← BLACK        // Case 3
14.                 color[p[p[z]]] ← RED       // Case 3
15.                 RIGHT-ROTATE(T, p[p[z]])  // Case 3
16.     else (if p[z] = right[p[p[z]]])
17.          // same as the if part with “right” and “left” exchanged)
18.   end-while
19.   color[root[T ]] ← BLACK
```

Insertion- Fixup: Algorithm Analysis

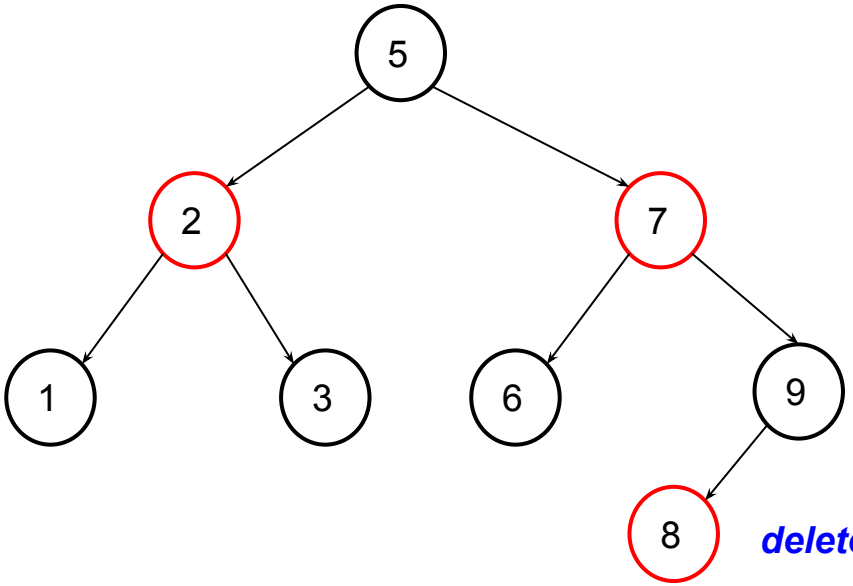
- The insertion procedure has O(log n) cost, since the height of the tree is log n
- Each iteration takes O(1) time and each iteration moves z up 2 levels, except the last one.
- There are at most 2 rotations overall
- Insertion in a red-black tree takes O(lg n) time.

Red-Black Deletion

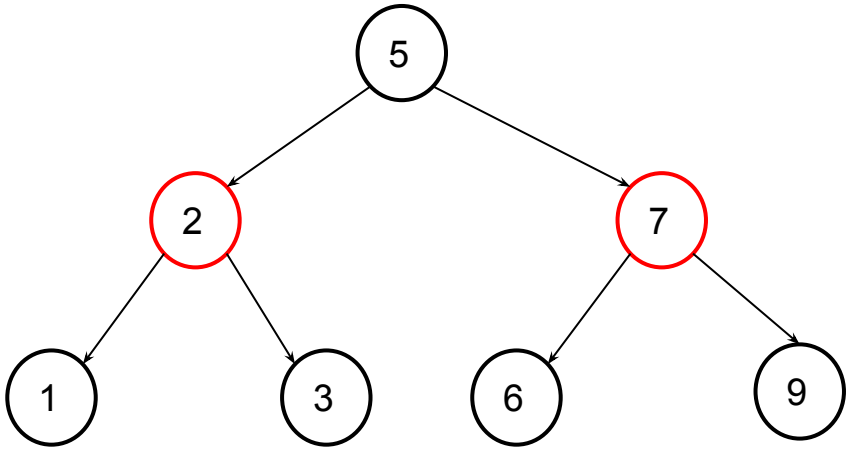
Red-Black Tree: Delete

- When deleting any node, we should make sure that we maintain all the red-black tree properties
- The properties that may be violated depends on the color of the deleted node.
- The same as insertion we go through a regular BST deletion, then check and fix if any violation occurred.
- What do you think more problematic deleting a red node or black node?

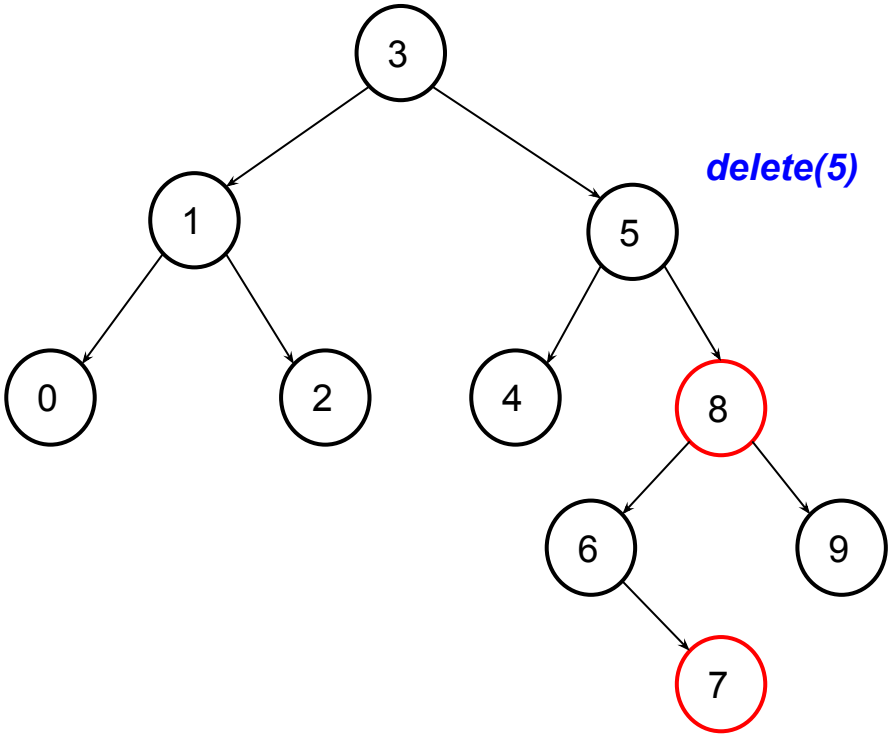
Red-Black Tree: Delete Example (1)



delete(8)

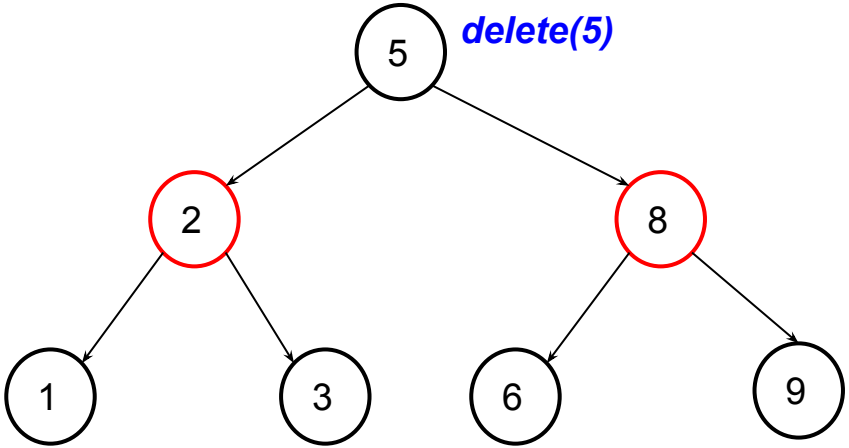


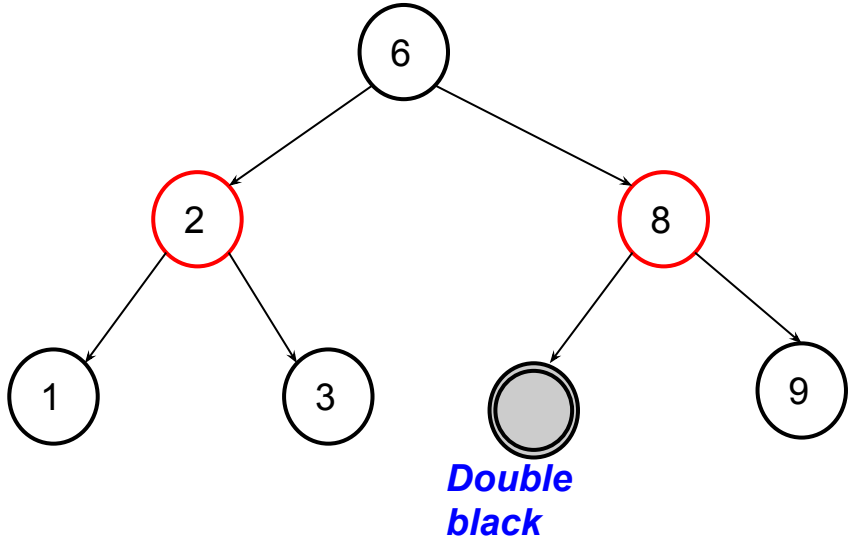
Red-Black Tree: Delete Example (2)



1. To delete node 5, we convert it to a node with zero or 1 not null node
2. If red node just delete it
3. If black node with red child then delete the node and replace it with its child and recolor

Red-Black Tree: Delete Example (3)





Red-Black Tree: Properties violation on Delete

If x is black, we could have violations of red-black properties:

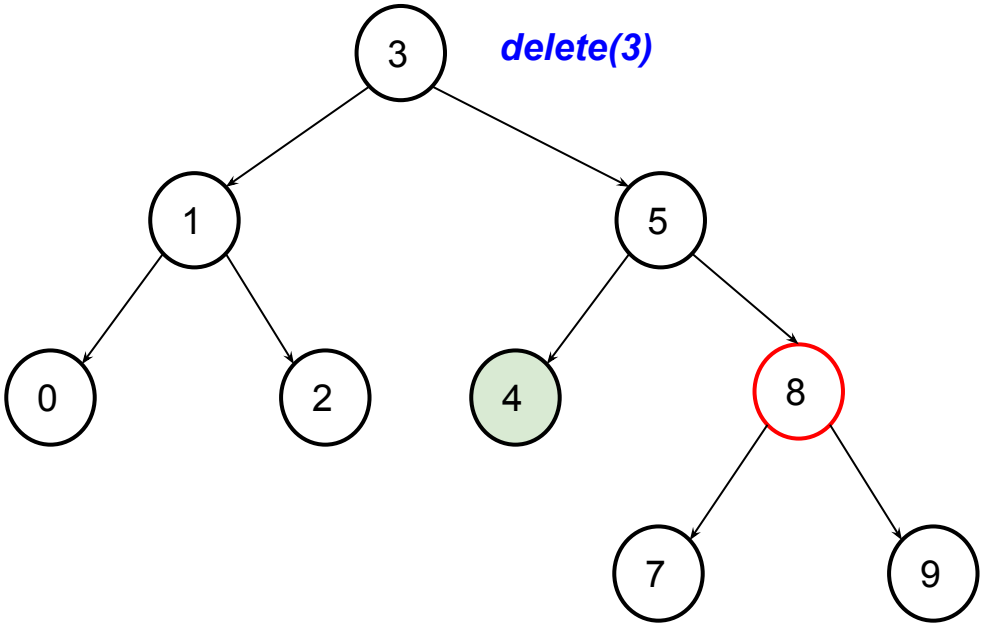
1. Root is always black
2. Every red node has a black parent
3. For each node, all simple paths from the node to descendant leaves contain the same number of black nodes.

Red-Black Tree: Delete node In nutshell

1. Find the node to be deleted applying regular binary search tree.
2. If node to be deleted has 2 non-null children, then replace it with its inorder successor, then delete the inorder successor node.
3. If node to be deleted is red then just delete it and terminate
4. If node to be deleted is black but has one red child, replace it with that child and recolor the child from red to black.
5. Otherwise, there are four cases we have to handle using delete-fix up

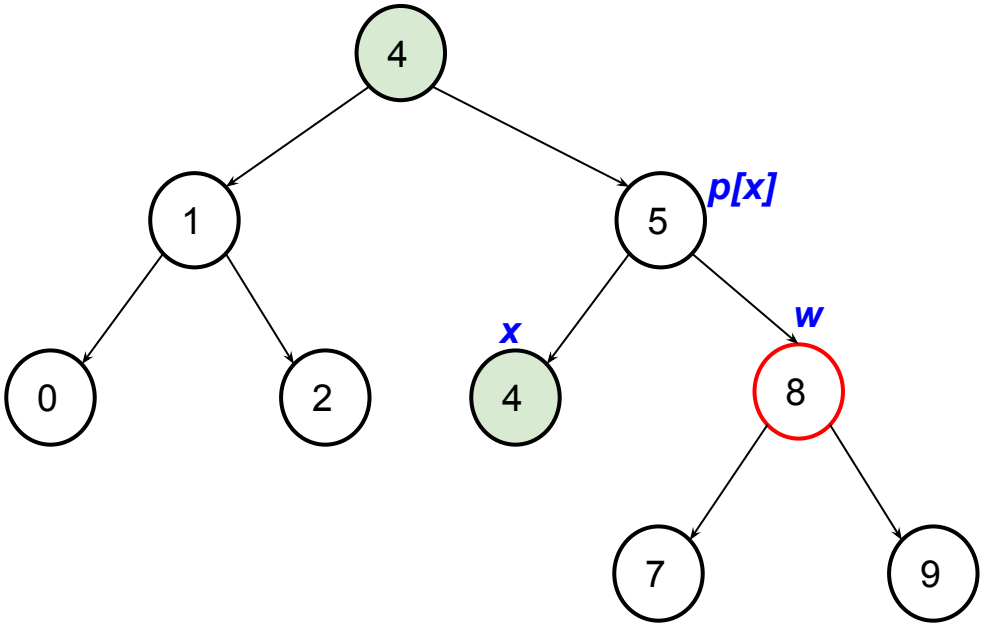
Red-Black Tree: Delete Case 1

Case 1: x's sibling w is red



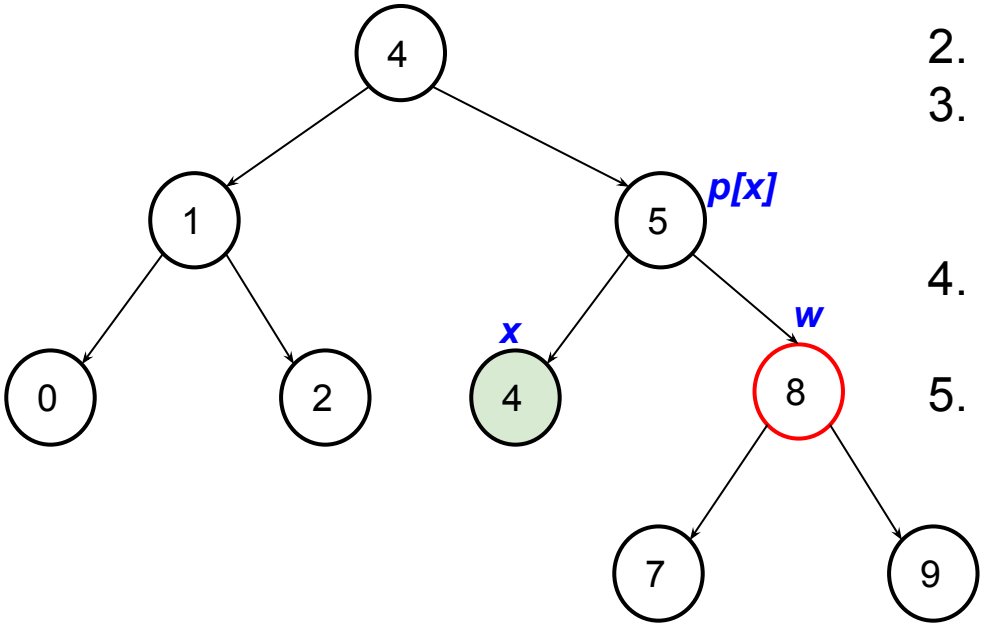
Red-Black Tree: Delete Case 1

Case 1: x's sibling w is red



Red-Black Tree: Delete Case 1

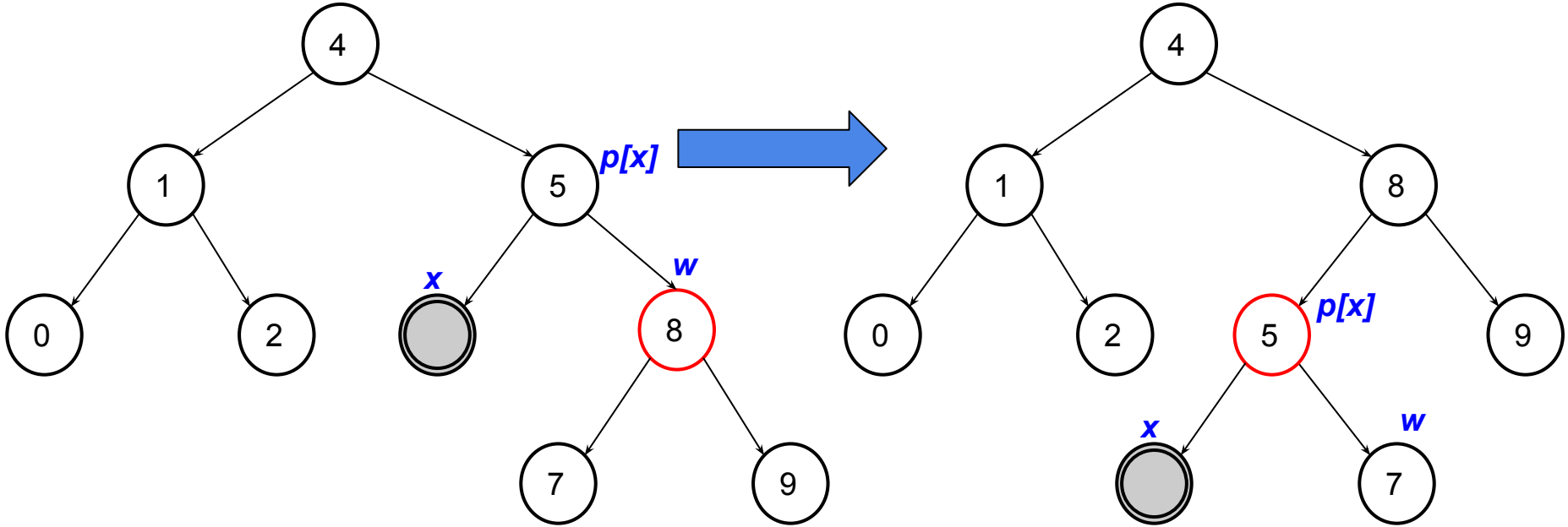
Case 1: x's sibling w is red



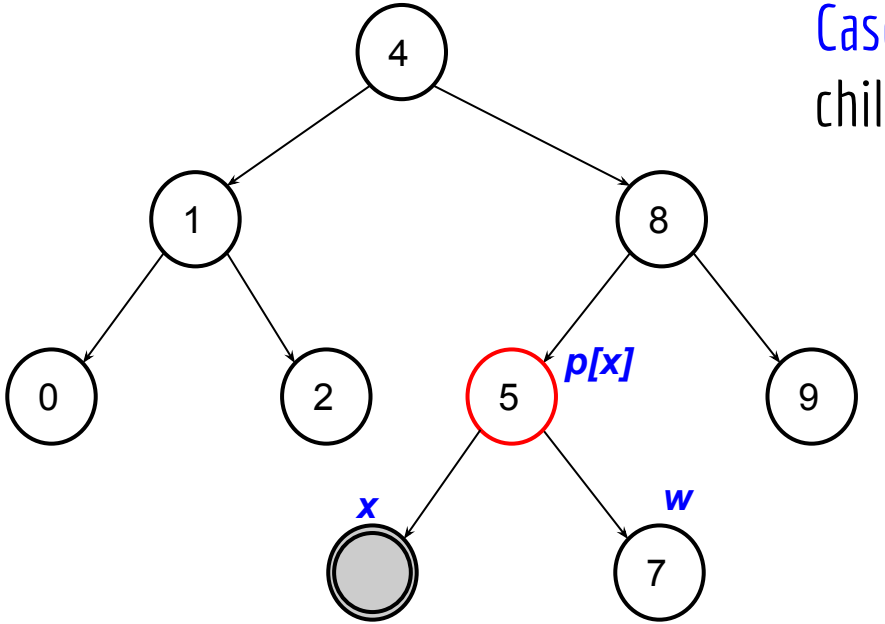
1. The sibling w is red, then it must have black parent
2. Switch the color of w and p[x]
3. Then perform a left-rotation on p[x] without violating any of the red-black properties.
4. New sibling of x was a child of w before rotation must be black.
5. Go immediately to case 2, 3,or 4

Red-Black Tree: Delete Case 1

Case 1: x's sibling w is red



Red-Black Tree: Delete Case 2

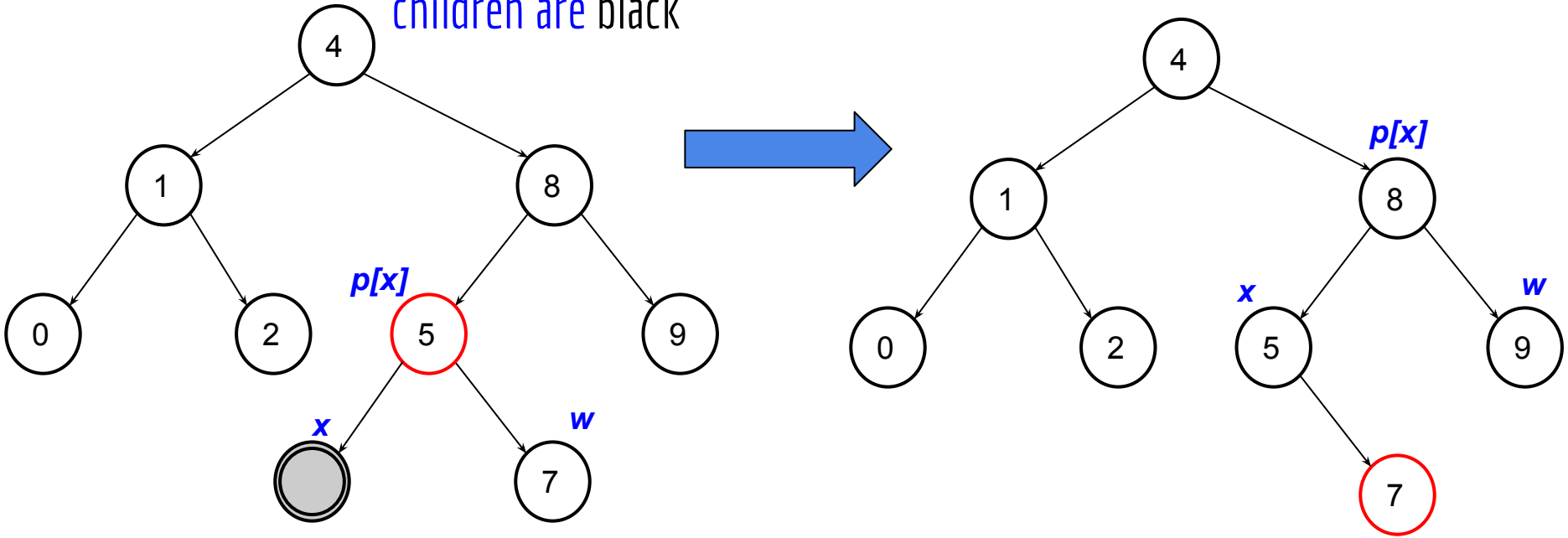


Case 2: x's sibling w is black, and both of w's children are black

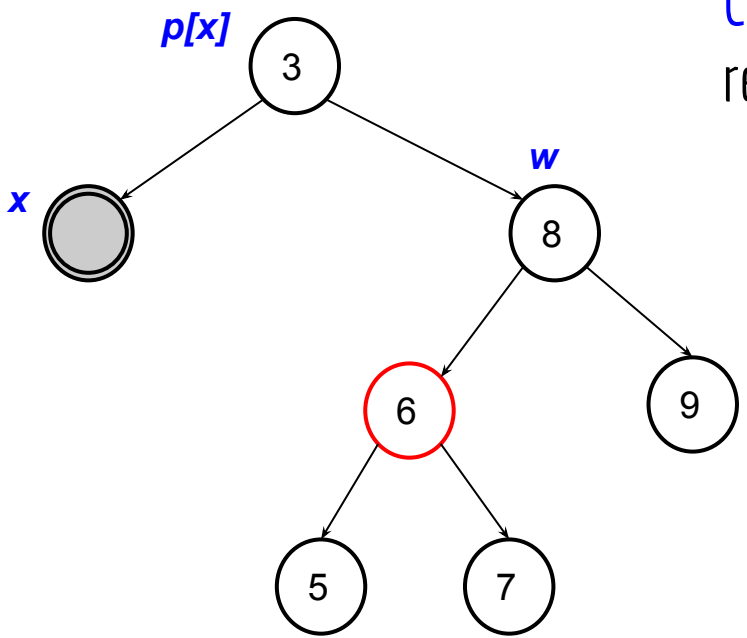
1. Take 1 black off x (singly black) and off w
2. In this case change w color from black to red
3. set p[x] color to black
4. Let p[x] the new x

Red-Black Tree: Delete Case 2

Case 2: x's sibling w is black, and both of w's children are black



Red-Black Tree: Delete Case 3

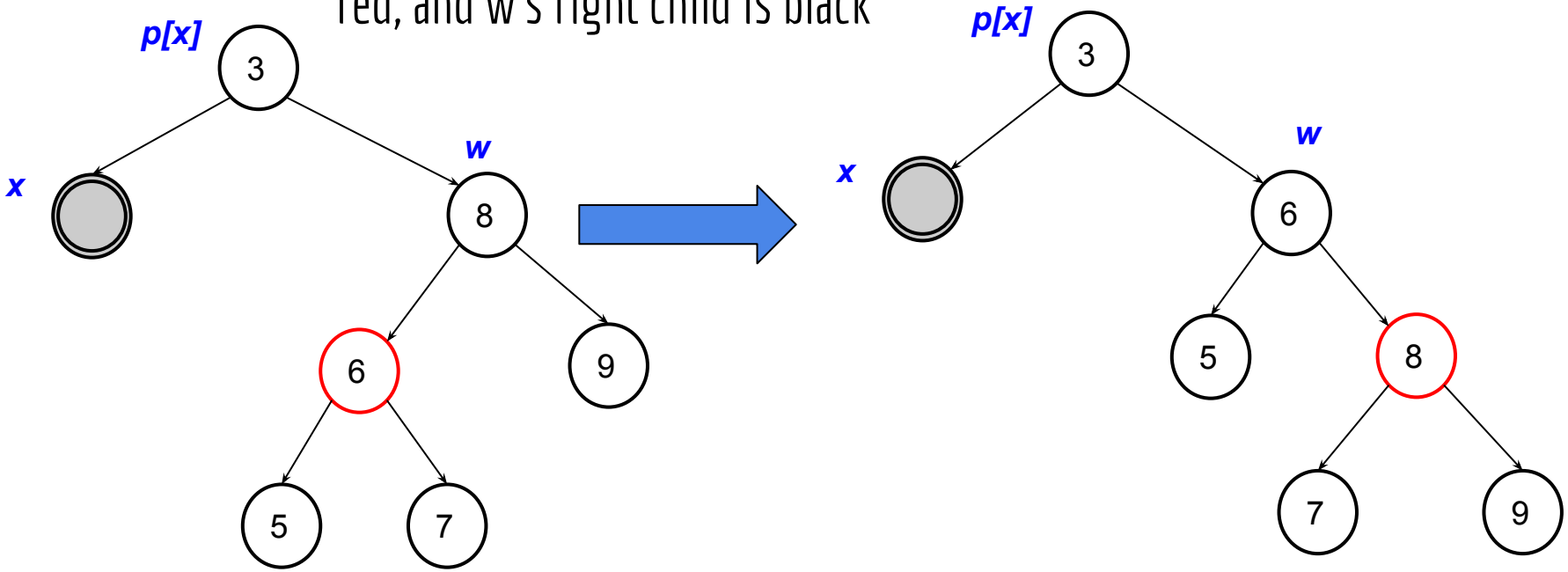


Case 3: x's sibling w is black, w's left child is red, and w's right child is black

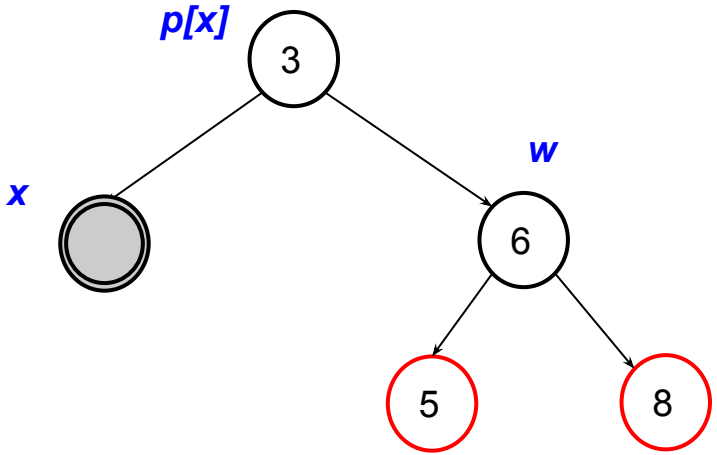
1. Make w red and w's left child black.
2. Then right rotate on w.
3. New sibling w of x is black with a red right child then go to case 4.

Red-Black Tree: Delete Case 3

Case 3: x's sibling w is black, w's left child is red, and w's right child is black



Red-Black Tree: Delete Case 4

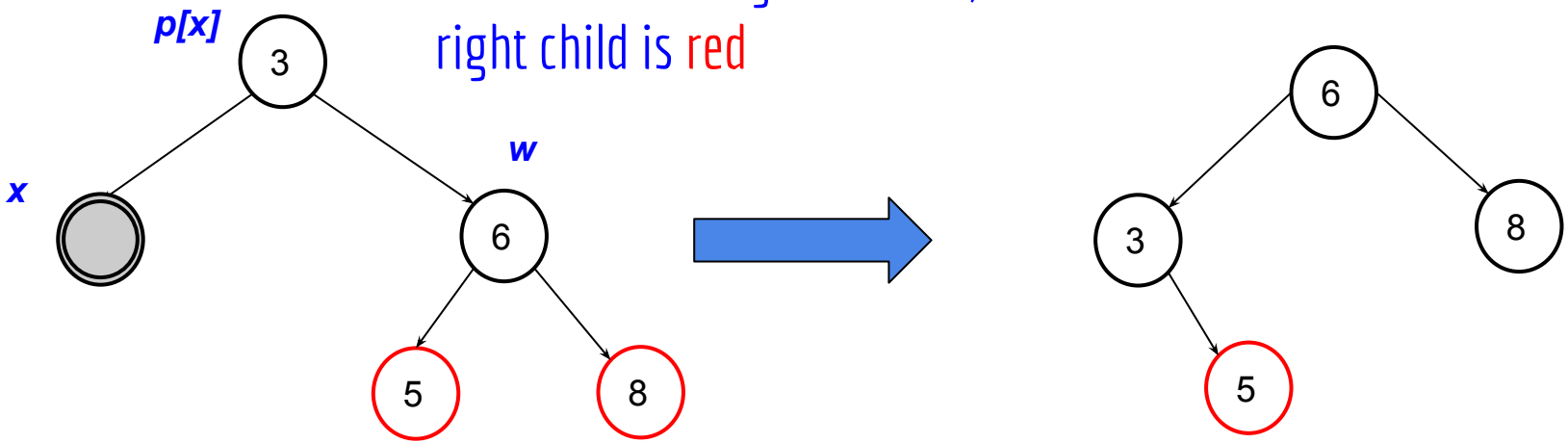


Case 4: x's sibling w is black, and w's right child is red

1. Recolor w with w's left child color
2. Take 1 black off x (singly black)
3. Set the color of right[w] to black
4. Left-rotate on p[x]

Red-Black Tree: Delete Case 4

Case 4: x's sibling w is black, and w's right child is red



Red-Black Tree: Delete Fixup Algorithm

```
1. algorithm DeleteFixup(T,x)
2.    while x !=root[T ] and color[x] = BLACK
3.        if x = left[p[x]]
4.         then w ← right[p[x]]
5.                 if color[w] = RED
6.                    then color[w] ← BLACK          // Case 1
7.                         color[p[x]] ← RED         // Case 1
8.                         LEFT-ROTATE(T, p[x])      // Case 1
9.                         w ← right[p[x]]           // Case 1
```

Red-Black Tree: Delete Fixup Algorithm

```
1.  algorithm DeleteFixup
2.    // continue where x is still left p[x]
3.    if color[left[w]] = BLACK and color[right[w]] = BLACK
4.              then color[w] ← RED                          // Case 2
5.                   x ← p[x]                                // Case 2
6.              else if color[right[w]] = BLACK
7.                        then color[left[w]] ← BLACK        // Case 3
8.                             color[w] ← RED                // Case 3
9.                             RIGHT-ROTATE(T,w)             // Case 3
10.                            w ← right[p[x]]               // Case 3
11.                  color[w] ← color[p[x]]                  // Case 4
12.                  color[p[x]] ← BLACK                     // Case 4
13.                  color[right[w]] ← BLACK                 // Case 4
14.                  LEFT-ROTATE(T, p[x])                    // Case 4
15.                  x ← root[T ]                            // Case 4
16.       else (same as then clause with “right” and “left” exchanged)
17.   color[x] ← BLACK
```

Delete in Red-Black Tree

Case 1 is transformed to case 2, 3 or 4 by changing the colors

Case 2, the extra black represented by the pointer x is moved up the tree by coloring node w red and setting x to point to node p[x]. If we enter case 2 through case 1, the while loop terminates because the new node x is red-and-black, and therefore the value c of its color attribute is RED.

Case 3 is transformed to case 4 by exchanging the colors of nodes w and its left child and performing a right rotation on w.

Delete in Red-Black Tree

In Case 4, the extra black represented by x can be removed by changing some colors and performing a left rotation (without violating the red-black properties), and the loop terminates.

Deletion Fixup: Algorithm Analysis

- Within the while loop x always points to a non root doubly black node and w is x's sibling.
- 8 cases in all, 4 of which are symmetric to the other.
- O(log n) time to get through RB-Delete up to the call of RB-Delete-Fixup.
- Within RB-Delete-Fixup:
  - Case 2 is the only case in which more iterations occur.
- Each of cases 1, 3, and 4 has 1 rotation 3 rotations in all.
- The deletion takes O(log n) time

Self-Check Exercise

Q1] - Insert into an initially empty red-black tree, the following keys: [4, 7, 12, 15, 3, 5, 14, 18, 16, 17].

Use drawing to illustrate your answer. Make sure to count the number of rotation and recoloring operations.

Now insert the same list of keys into an empty AVL Tree.

You can use the following link to check your answer.

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