Red-Black Trees Introduction

BST

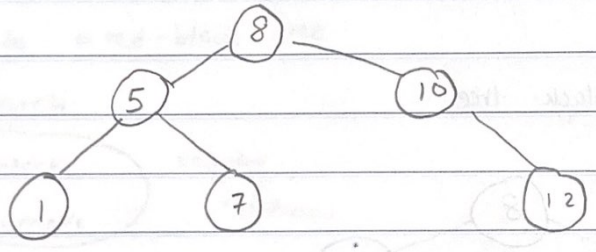
↳ Ordered, or sorted, binary trees

↳ Nodes can have 2 subtrees

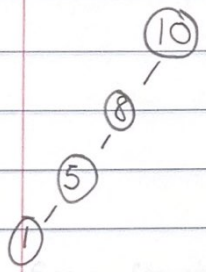
↳ items to the left of given node are smaller

↳ items to the right of given node are bigger

ex of BST:



Now lets say our BST is like:



its nothing more than a list, and to find the node (1) you would have to traverse through the whole BST

The answer to this is Balanced Search Trees

guaranteed height of $O(\log N)$ for N times

red-black tree is a particular type of balanced search tree

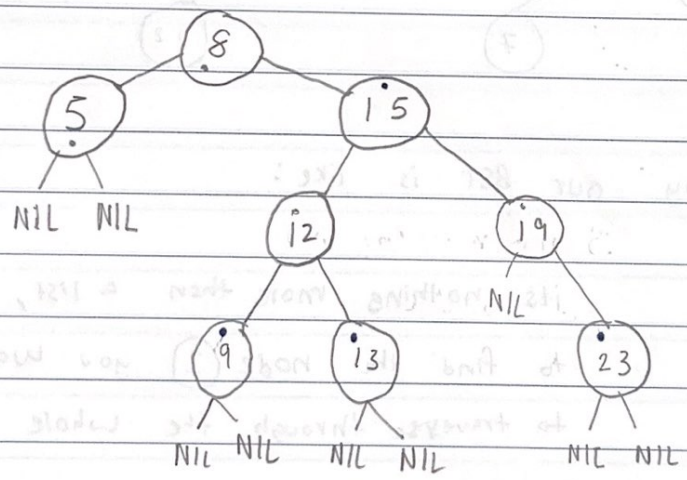
properites of red-black tree

1. A node is either red or black
2. The root and leaves (NIL) are black
3. If a node is red, then its children are ALL black
4. All paths from a node to its NIL descdants contain the same # of black nodes

ex of red-black tree

z = red

z = black



black-height → # of Nodes from Root to the NIL node (we don't count the root Node)

Extra notes:

① Nodes require one storage bit to keep track of color

② The longest path (root to furthest NIL) is no more than twice the length of the shortest path (root to nearest NIL)

- Shortest path: all black nodes
- Longest path: alternating red & black

Operations to a red-black tree

↳ Search

↳ Insert
↳ Remove

require rotations

Time Complexity:

Search → O(log N)
Insert → O(log N)
Remove → O(log N)

Space Complexity: O(N)

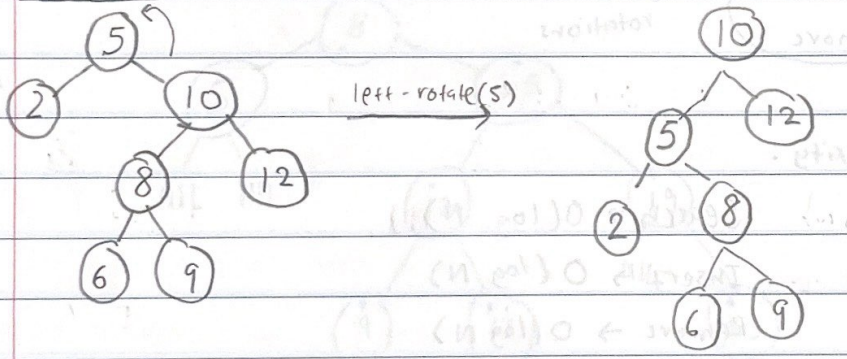
Red-black trees Rotations

Rotation: 1) alters the structure of a tree by rearranging subtrees

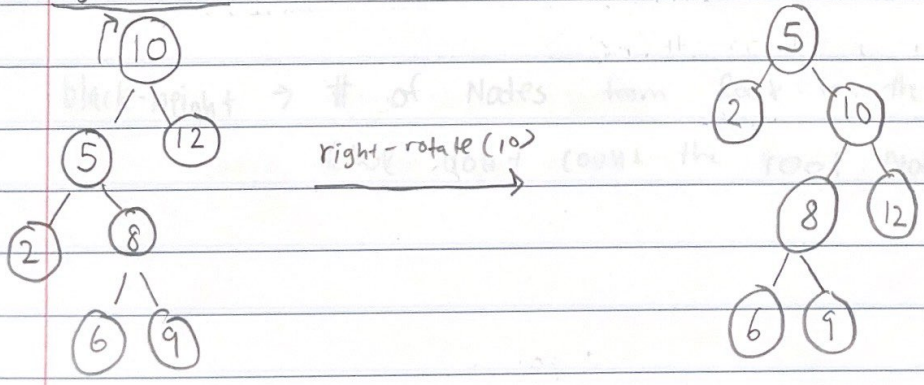
2) goal is to decrease the height of the tree
   - red-black trees: maximum height of $O(\log N)$
   - larger subtrees up, smaller subtrees down

3) Does not affect the order of elements (smaller elements to left, larger to the right)

left-rotate



right-rotate



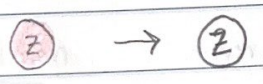
Time complexity: O(1) for rotations

Red-Black Trees: Insertions (strategies)

Strategy:

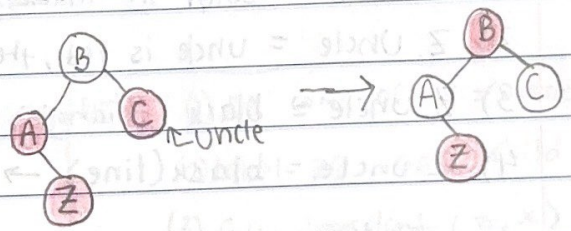
1) Insert z and color it red
2) Recolor and rotate nodes to fix violation
   - ↳ 0. z = root
   1. z.uncle = red
   2. z.uncle = black (triangles)
   3. z.uncle = black (line)

Case 0: z = root



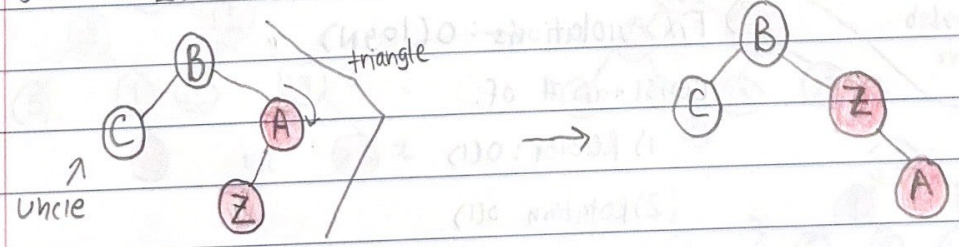
Solution: Color black

Case 1: z.uncle = red



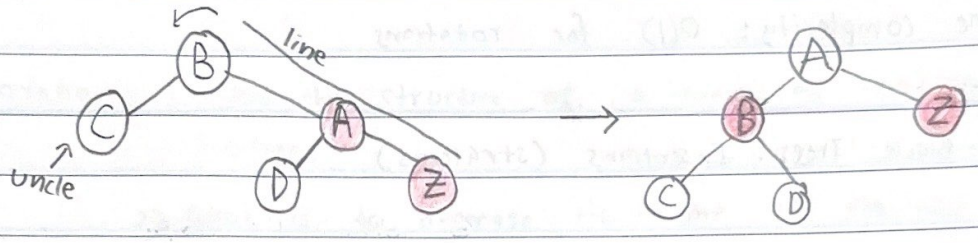
Solution: recolor z's parent, grandparent, and uncle!

Case 2: z.uncle = black (triangle)



Solution: rotate z.parent

Case 3: Z.uncle = black (line)



Solution: rotate Z.grandparent, then recolor

Strategy:

1) insert Z and color it red
2) Recolor and rotate nodes to fix violation

4 Scenarios guide:

1) Z.root = Color it black
2) Z.uncle = uncle is red, then recolor
3) Z.uncle = black (triangle) → rotate Z.parent
4) Z.uncle = black(line) → rotate Z.grandparent & recolor

Time Complexity:

1) Insert: O(log N)
2) Color red: O(1)
3) Fix violations: O(log N)

constant # of:

1) Recolor: O(1)
2) Rotation: O(1)

total: O(log N)

Red-black Trees - Deletions

Sections:

1) transplant
   - helps us move subtrees within the tree
2) delete
   - deletes the node
3) delete_fixup
   - fixes any red-black violations

Transplant

- helps us move subtrees within the red-black tree

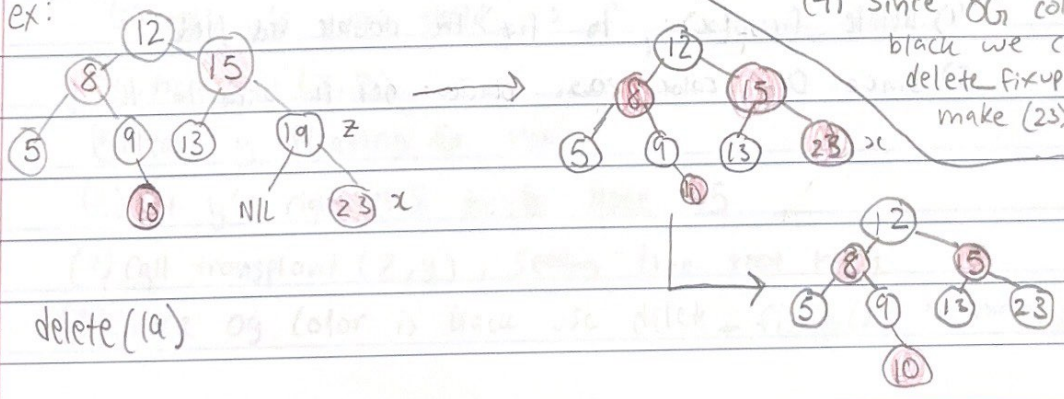
basically allowing us to change who parent/grandparent is

Deletion

1) left child is NIL
2) right child is NIL
3) neither is NIL

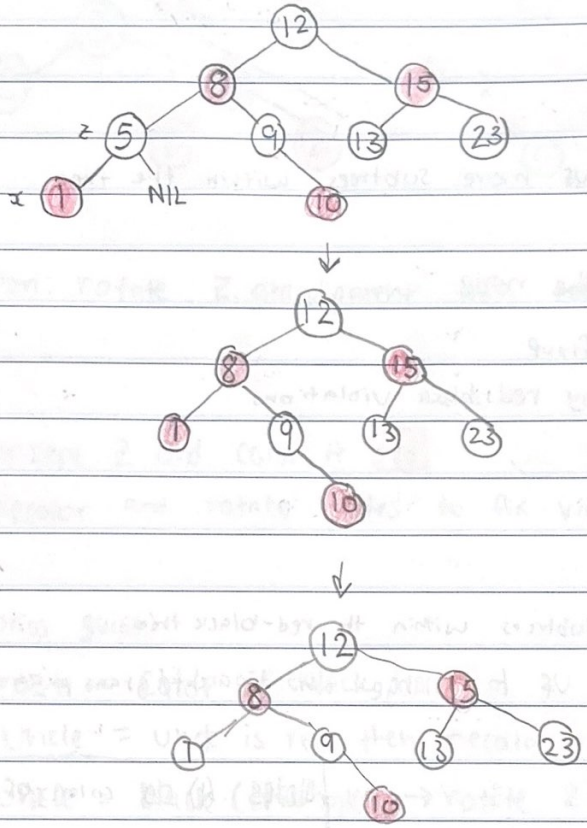
Notes: (1) og color of z = black
(2) label z's, right child x
(3) call transplant(z, x) to get rid of 19 and switch it with 23
(4) since OG color was black we call delete_fixup(x) and make (23) to black

Case #1 ex:



delete(19)

Case #2: right child is NIL



Notes: 1) z's og color is black

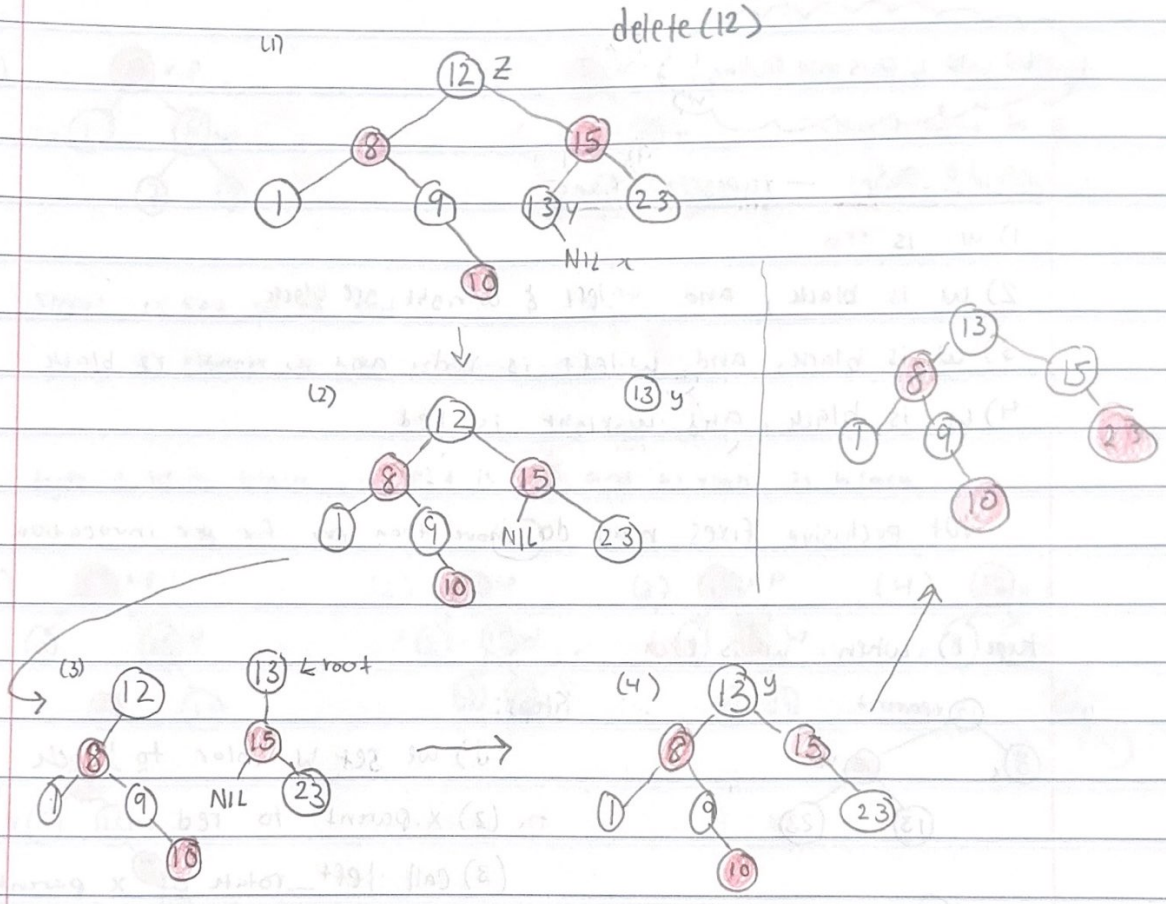
2) set left child (NON NIL one to x)

3) transplant (x, z) [NO more 5]

4) delete_fixup(x), to fix the double red Node

5) Since OG color was black get it back to that!

Case #3: neither child is NIL



Notes: (1) Find the minimum in z's right child subtree (here it's 13)

(2) label it y, and mark down OG color

(3) label y's right child as x

(4) transplant (y, z)

(5) leave y floating for now

(6) set y's right child to be Node 15

(7) Call transplant (z, y), setting tree root to y

(8) Since og color is black use delete_fixup(x) to make is black

Red-Black trees — Delete_fixups

w is x's sibling!

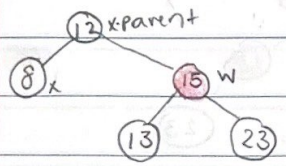
delete_fixups — types of fixes

1) w is red
2) w is black, and w.left & w.right are black
3) w is black, and w.left is red and w.right is black
4) w is black, and w.right is red

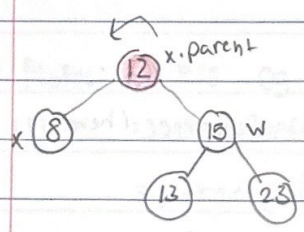
NOt exclusive fixes, may do more than one fix per invocation

type 1) when w is red

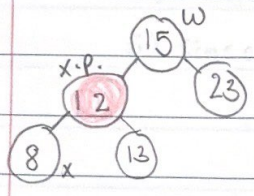
(1)



(2)



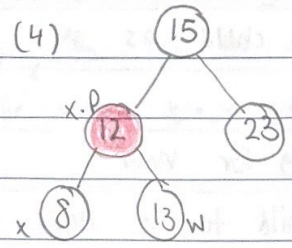
(3)



Steps:

(1) we set w color to black
(2) x.parent to red
(3) call left_rotate of x parent
(4) set w back to x siblins

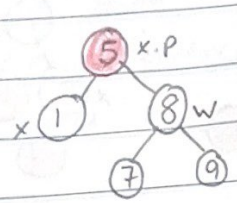
(4)



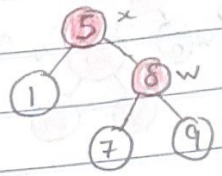
(These are Not full Red-Black Tree, ALL are Sub-trees)

type 2: w, w.left, w.right are black

(1)



(2)



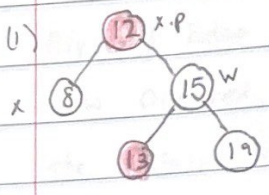
* Not valid Red-Black (would need to do type #1 again)

Steps: 1) Set w to red

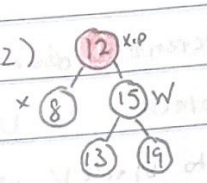
2) Change x to its parent

type 3: W is black, w.left is red, and w.right is black

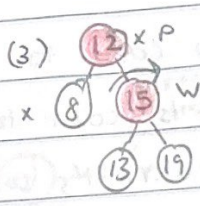
(1)



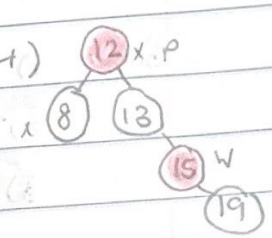
(2)



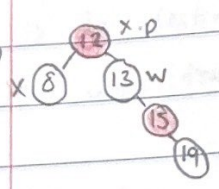
(3)



(4)



(5)



Steps: 1) Set W.left color to black

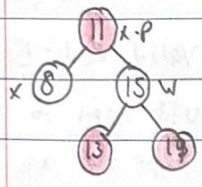
2) Set w.color to red

3) rotate w right

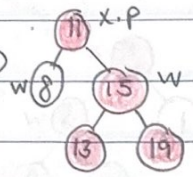
4) Set w to x sibling

type 4: w is black and w.right is red

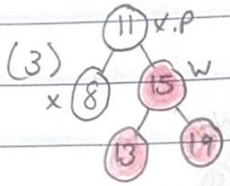
(1)



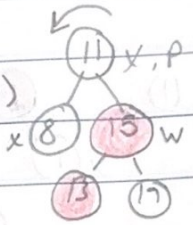
(2)



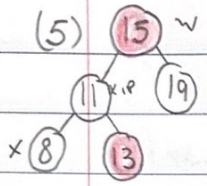
(3)



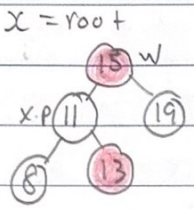
(4)



(5)



(6)



Steps: 1) set w color to x.parent's color

2) set x.parents color to black

3) set w.right's color to black

4) left rotate on x's parent.

5) set x equal to tree's root

Time Complexity

$O(\log N)$