Faculty of Engineering & Technology
Electrical & Computer Engineering Department

COMP2321

Research Report No.1

Red-Black Trees

Prepared by: Tareq Shannak

ID Number: 1181404

Instructor: Dr. Radi Jarrar

Section Number: 3

Date: 16/5/2020

Red-Black Tree's Concept

Red-Black tree is a binary search tree which has a balance property to reduce the time complexity of reaching the last node in the tree. So we can ensure that the depth of the tree is O(log n). Each node of this tree has an extra bit that determine the node's color (Red or Black), and these bits are used to know if the tree is balanced or not yet.

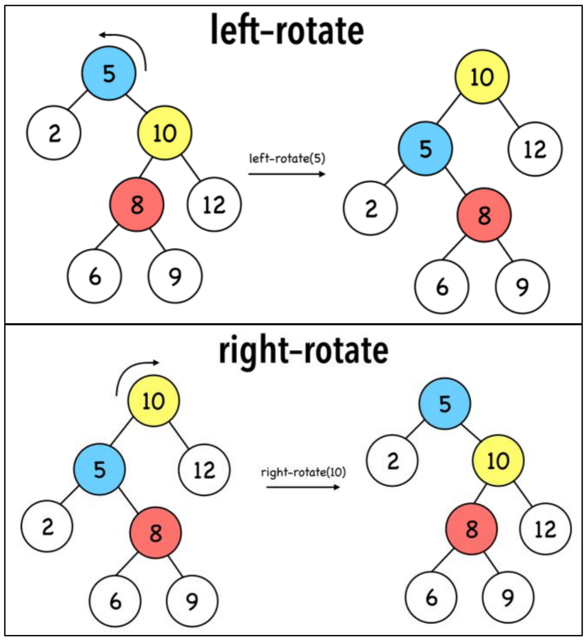
Red-Black Tree's Properties

- A node is either red or black, and the root and leaves (NIL) are black.
- There are no two adjacent red nodes. So the parent and children of a red node are black.
- All paths from a node to its NIL descendants contain the same number of black nodes.

When it is best to use it

Red-Black tree is more efficient in the re-balancing stage; because the rotation in this tree is an O(1) operation, hence the tree offers faster insertion and deletion compared to AVL tree at the cost of slightly lower lookup, so if your program have to do many insertions or deletions with less searching, then Red-Black trees should be preferred.

Rotations on Red-Black Trees



- Left Rotate: As shown in the picture, the parent node will become the left child of its right child, and the left child (and its followers if exists) of the right child will become the right child of the ex-parent.

- Right Rotate: the parent will become the right child of its left child taking the right child on its left child (if exists).

Red-Black Tree's Operations

Searching

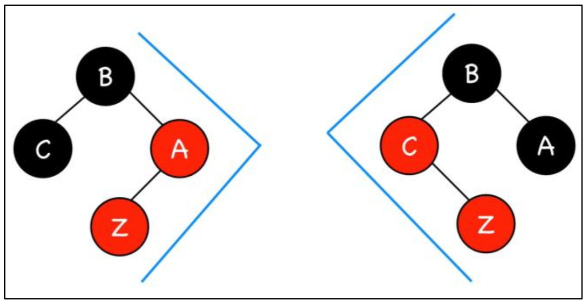
Finding an insertion point in the bottom of the tree or a node to delete is similar to the Finding operation in the simple Binary Search Tree (BST), hence its time complexity is O(log n).

Insertion

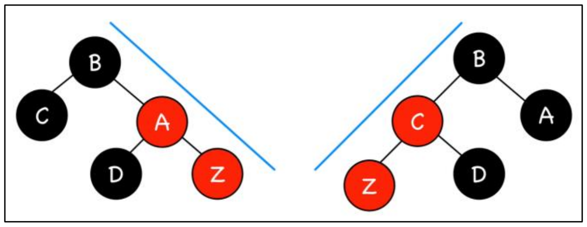
First, a node will be created with the new data (key) and its color is red; because the leaves (NIL) are black and the red node's children are always black, this node will be put in the correct place according to its key. After that, we need to fix violation by recoloring and rotating the nodes.

When a node is inserted it will produce one of these shapes:

Triangle shape: When Z's parent is the left child of Z's grandparent and Z is parent's right child, or vice versa as shown in the image on the right.

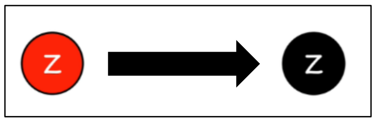


Line shape: When Z node is the left child of its grandparent's left child, or Z is the right child of its grandparent's right child.



There are four cases when a node Z is inserted:

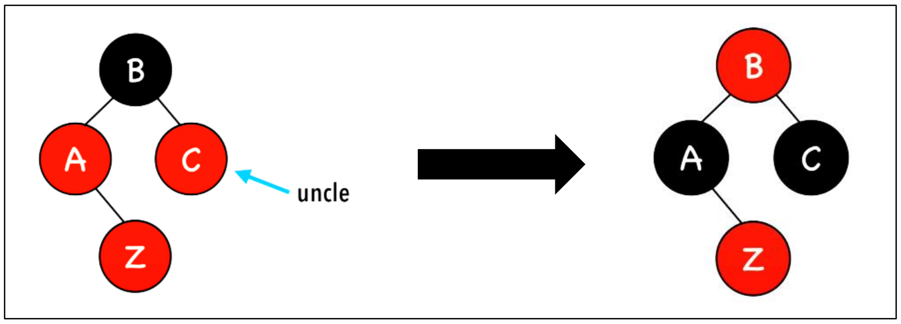
1- The node is the root



Just make its color black because it is the root.

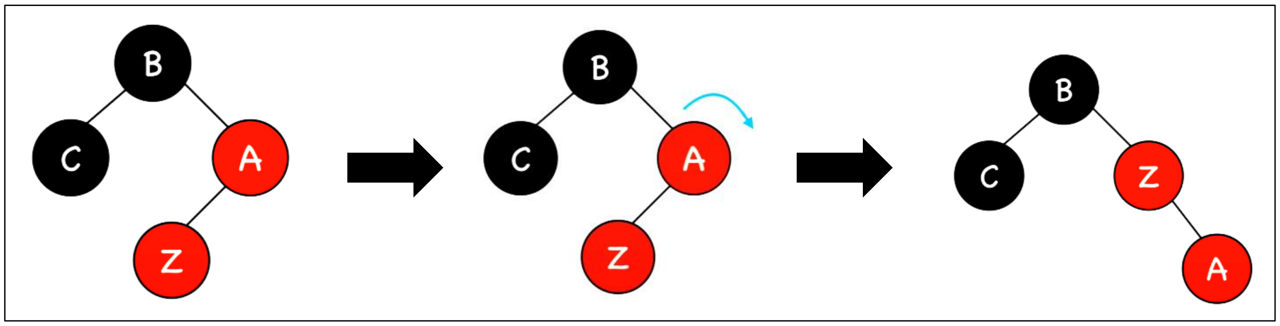
2- The node's uncle is red

The node's uncle means the second child of node's grandparent. As shown in the image below, the solution is recoloring all nodes except the new node. Pay attention that B isn't the root, it's just a sub-tree.



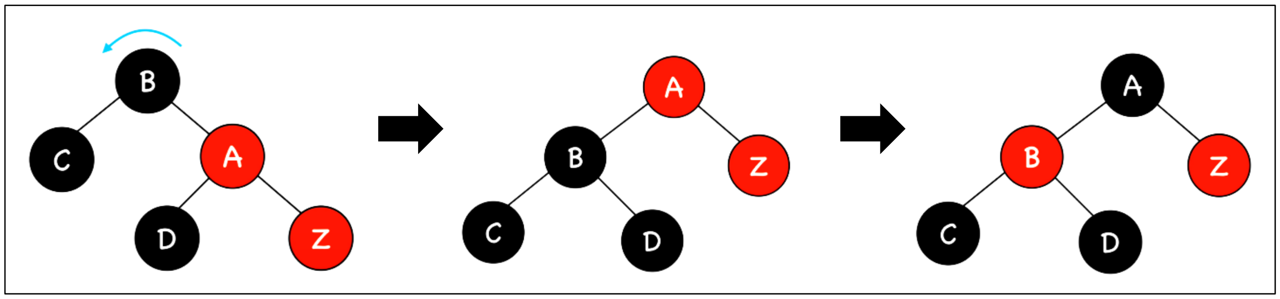
3- The node's uncle is black and the shape is triangle.

Z's parent will rotate to the opposite direction of Z, so Z will become the parent of A as shown in the image below. The balance operation hasn't finished yet, hence this tree will go to the next case.



4- The node's uncle is black and the shape is line.

First, the grandparent will rotate on the opposite direction of Z and its parent, after that the tree will be recolored correctly.



Deletion

In insertion, we check color of node's uncle to decide which case to execute, but in deletion we check color of node's sibling to decide the case. First, we delete the node to be deleted from the tree as any simple Binary Search Tree (BST). To ensure that the heights of black nodes in every possible path are equal, we need to fix the tree. When a black node become deleted and replaced by another black node it called double black node. Let V be the node to be deleted and U is the child that replaces V, we can handle cases as below. Pay attention that may V is a leaf node, U is NULL and its color is black and s is V's sibling.

1- Either U or V is red

Just make sure that color of U is black.

2- Both U and V are black

In this case, U will become a double black node after delete operation. If U is root, make it a single black node, otherwise look to these cases below.

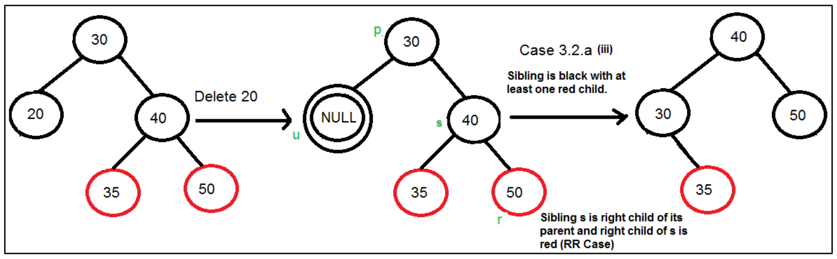
A. S is black and its both children are black.

Recoloring the nodes and make S's parent a double black if it's black, if it is red make the parent single black.

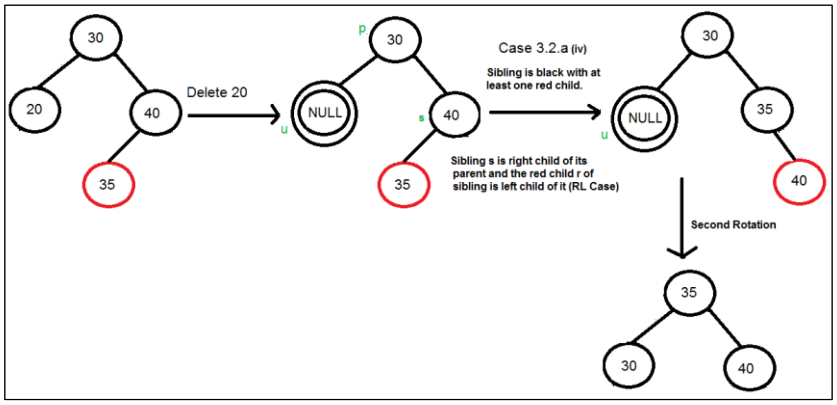
B. S is black and at least one of S's children is red

Let R be the red child of S, this goes to four subcases to decide the rotations:

i. Right-Right Case: S is a right child and R is a right child or S's children are both red when S is a right child.


30
20
40
35
50
Delete 20
u
NULL
p
30
s
40
35
50
r
Case 3.2.a (iii)
Sibling is black with at least one red child.
40
30
50
35
Sibling s is right child of its parent and right child of s is red (RR Case)

ii. Right-Left Case: S is a right child and R is a left child.


30
20
40
35
Delete 20
u
NULL
p
30
s
40
35
Case 3.2.a (iv)
Sibling is black with at least one red child.
Sibling s is right child of its parent and the red child r of sibling is left child of it (RL Case)
u
NULL
30
35
40
Second Rotation
35
30
40

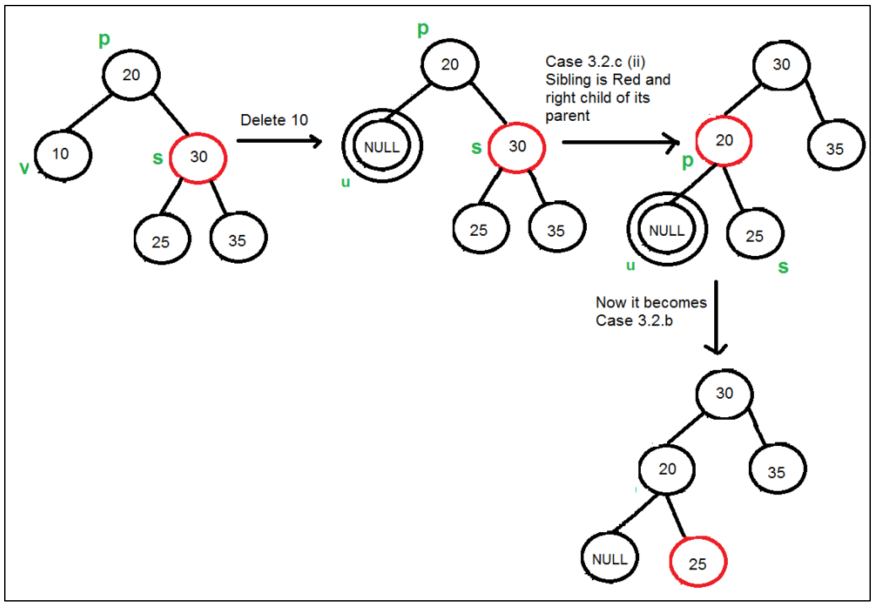
iii. Left-Left Case: S is a left child and R is a left child or S's children are both red when S is a left child. This is the opposite of Right-Right Case.

iv. Left-Right Case: S is a left child and R is a right child. This is the opposite of Right-Left Case.

C. S is red

This case make a rotation to move the old S and its parent (and recolor them), so the new S will be black, hence it will lead us to one of the previous cases (A, B). It can be divided as two subcases:

i. Right Case: S is a right child. The procedure shown in the image below.

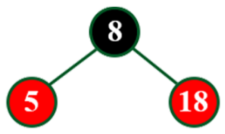


ii. Left Case: It's the opposite for the subcase above when S is a left child.

Time and Space Complexity

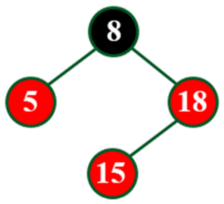
	Average	Worst Case
Space	O(n)	O(n)
Search	O(log n)	O(log n)
Insert	O(log n)	O(log n)
Delete	O(log n)	O(log n)

Example on Red-Black Tree's Operation



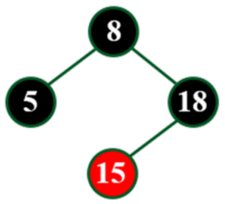
insert (15)

Tree is not Empty. So insert newNode with red color.



Here there are two consecutive Red nodes (18 & 15).
The newnode's parent sibling color is Red
and parent's parent is root node.
So we use RECOLOR to make it Red Black Tree.

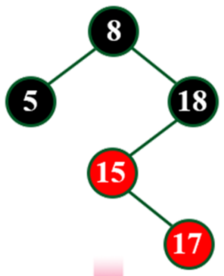
After RECOLOR



After Recolor operation, the tree is satisfying all Red Blacl properties.

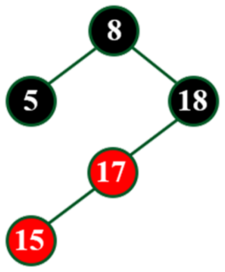
insert (17)

Tree is not Empty. So insert newNode with red color.

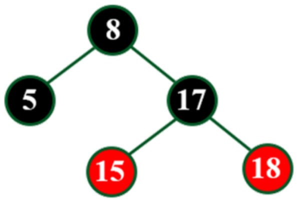


Here there are two consecutive Red nodes (15 & 17).
The newnode's parent sibling is NULL. So we need rotation.
Here, we need LR Rotation & Recolor.

After Left Rotation

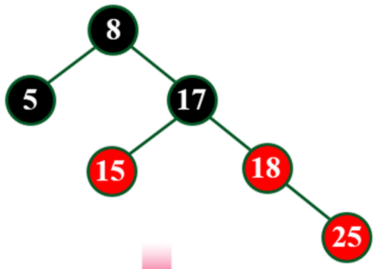


After Right Rotation & Recolor

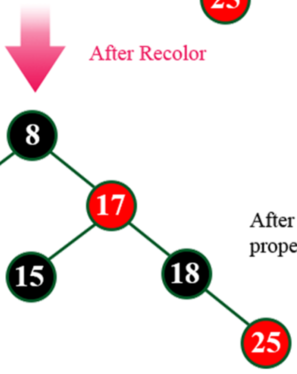


insert (25)

Tree is not Empty. So insert newNode with red color.

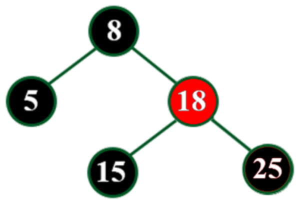


Here there are two consecutive Red nodes (18 & 25).
The newnode's parent sibling color is Red
and parent's parent is not root node.
So we use RECOLOR and Recheck.



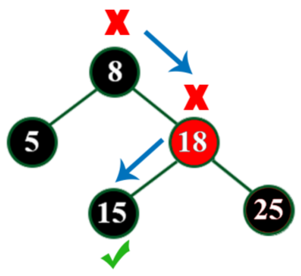
After Recolor operation, the tree is satisfying all Red Black Tree properties.

delete(17)



After Recolor operation, the tree is satisfying all Red Black Tree properties.

find(15)



References

- Data Structure Lecture Notes
- Dr. Radi Jarrar Slides
- https://en.wikipedia.org/wiki/Red%E2%80%93black_tree
- https://www.geeksforgeeks.org/red-black-tree-set-1-introduction-2/?ref=lbp
- https://stackoverflow.com/questions/9469858/how-to-easily-remember-red-black-tree-insert-and-delete
- https://www.youtube.com/watch?v=qvZGUFHWChY&list=PL9xmBV_5YoZNqDI8qfOZgzbqahCUmUEin