Math 8650
Data Structures
Red Black Trees

Submitted By:
Ravisutha Sakrepatna Srinivasamurthy
Muaz Ahmad

December 8, 2017

Contents

1 Introduction

A red black tree is a self balancing binary search tree. It is built on top of the binary search tree with the promise of maintaining a tree depth of O(log(n)). The major limitation of binary search tree is that when a sorted array is inserted into it, its depth will be O(n). This also implies costlier search, insert and delete operations (in O(n) order). Thus by removing this limitation of BST, RBT finds its applications in C++ STL dictionary implementation etc.

BST offers three major operations:

- Search
- Insert
- Delete

As RBT is also a type of BST, it offers the same operations. The following sections describe properties of RBT and implementation of these operations in the same order.

2 Properties

In order to maintain a depth of O(logn), the RBT has some constraints defined. These constraints are the unique properties of a RBT. The following are its properties.

1. The root should always be black.
2. There can be no two consecutive red nodes.
3. Leaves should always end in black null nodes.
4. Same number of black nodes in each path form root to leaves.

3 Search

Search implementation in RBT is exactly same as the BST. If the value is lower than its parent, you go to the left subtree and if the value is greater than its parent, you go the right subtree until either you find the value or you reach the leaf.

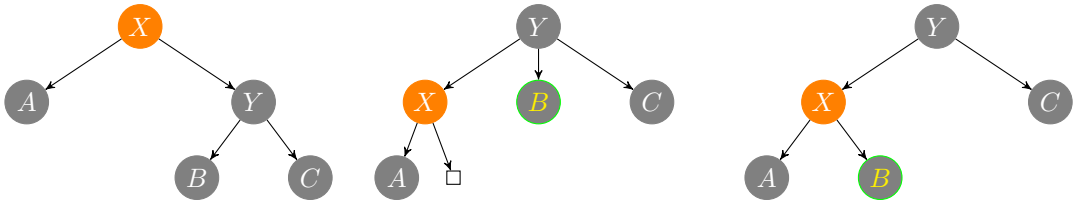
4 Insertion

Insertion in RBT is quite different from BST. It requires a few more operations. These operations are called rotations.

4.1 Rotations

There are two types of rotations.

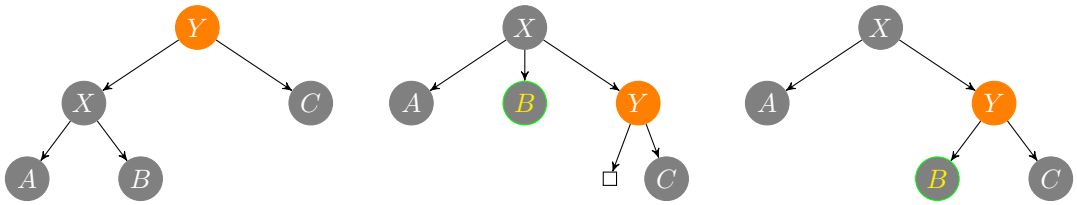
Left Rotation Left rotation is best explained with an example. Consider the tree shown in Figure 1. Now, left rotation on X implies that I am pulling X to the left subtree. Because X and Y are connected, Y will be pulled up and X becomes Y's left child. But Y already has a left child B. As B is greater than X, it will become the right child of X.



(a) Given tree (b) Transition (c) After rotation

Figure 1: Left Rotation

Right Rotation Right rotation is the exact opposite of left rotation. Like the previous case, lets consider a tree as shown in Figure 2 and do right rotation on Y. This implies that Y will be pulled towards the right subgraph. Now Y becomes right child of X. But X already has a right child B. Because B is smaller than Y, B will now be assigned as the left child of Y.



(a) Given tree (b) Transition (c) After rotation

Figure 2: Right Rotation

With these operations, we can now proceed to insertion and deletion.

4.2 Pseudo Code

These will be the steps for performing insertion in a RBT.

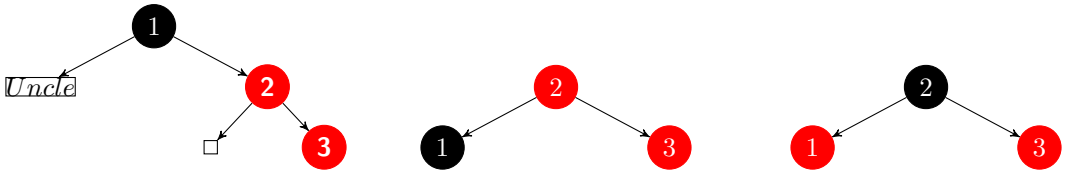
1. All new nodes will be colored as red except for the root node which will always be black.
2. As in BST, insert the new node to its proper position.
3. Eventually, you will end up violating the RBT properties. Fix these violations.

Fixing up the violations is what makes RBT insertion different from BST insertion. Now, we will look into various possible violations and how they can be fixed.

4.3 Fixing up insertion violations

We will now look at the violations on the right side of the subgraph. Fixing up violations on left side of the subgraph will be the mirror cases which just needs the operations to be preformed on the opposite directions.

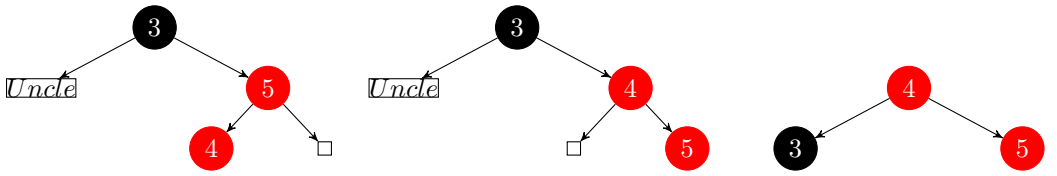
- Case 0: If the root is red, make it black.
- Case 1: Uncle is a black node and the violation is in line formation.
  Consider the following tree (Figure 3). We are trying to insert 3 into the RBT. We have violation between 2 and 3 red nodes. The grandparent, parent and the node (3) forms a line kind of structure. Hence the name line formation. In this case, to fix the violation, left rotate grand parent and swap the colors of grand parent and parent. Now the violation is fixed.



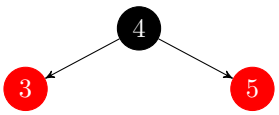
(a) Consecutive red nodes (b) Right rotation on GP (c) After recoloring

Figure 3: Showing transitions in the tree to fix the case 1 violations

- Case 2: Uncle is a black node and the violation is in triangle formation.
  Consider the tree in Figure 5. We are trying to insert 4. Now, 3, 5, 4 nodes forms a triangle kind of formation. Now, if we do right rotation on parent (node 5), we will end up with the case 1 violation and we already know how to fix case 1. Hence the solution is to first rotate parent(5) to the right. In this new line formation, the last node is 5 and its parent is 4 and its GP is 3. Now recurse on node 5. That is rotate grand parent(3) to the left and finally swap colors of parent(4) and grand parent(3).



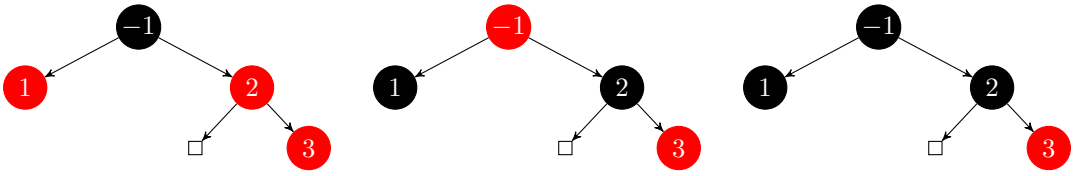
(a) Consecutive red nodes (b) Right rotation on parent (5) (c) Left rotation on GP (3)



(a) After recoloring

Figure 5: Showing transitions in the tree to fix case 2 violation

- Case 3: Uncle is a red node
  Consider a tree in Figure 6. We are trying to insert 3. It's uncle (1) is a red node. This is one of the trivial cases where you just have to recolor uncle and the parent to black and grand parent to red. Because we changed the color of grand parent, we have to recurse the fix operation on grand parent.



(a) Two consecutive red nodes (b) Recoloring parent and uncle (c) Recurse the fix on grand parent. In this example, GP is the root. Hence make it black

Figure 6: Showing transitions of the tree to fix case 3

The fixing operations for violation in the left subtree will be similar to above explained procedures but it will be in opposite direction. For example, right rotations will become left rotations. Following section explains the deletion of a value from a RBT tree.

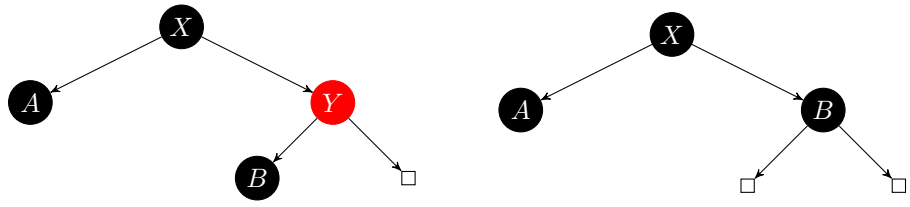
5 Deletion

Deletion in RBT builds on top of BST's, it applies the same concepts as BST initially, then it incorporates rotations to ensure the RBT is balanced and adheres to all constraints after the node has been deleted. The initial stage to the deletion is same as BST. The BST delete steps are enumerated below.

1. Node to be deleted has no children, simply delete it
2. Node to be deleted has one child only, then swap it with its child and delete it
3. Node has both children, swap node with its predecessor and delete predecessor

On RBT's, there are a lot more steps to be taken, based on various conditions before a node can be deleted to ensure that the RBT balancing properties are restored after a delete. We will consider all the cases with the node to be deleted at the left of the root and its sibling on the right of the root node. The other way round will be mirror cases and will require the exact opposite operations of the cases illustrated here. We will look at each of these cases stepwise.

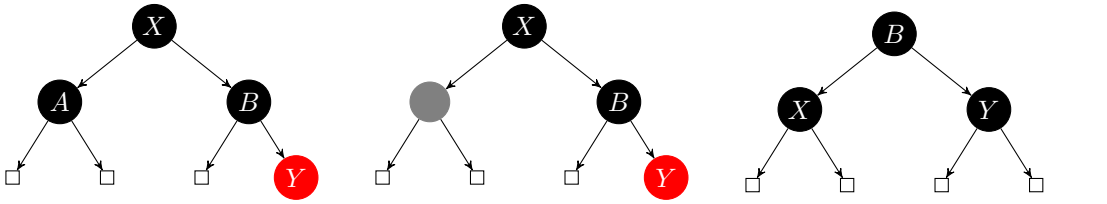
1. Case 1: Node to be deleted has 1 child and either the node or it's child is red. This is the trivial delete case, we simply replace the node to be deleted with its child and the replacement is always black *recolored black*. The illustration below shows the node Y being deleted.



(a) Delete node Y (b) After deleting Y

Figure 7: Trivial delete

2. Case 2: Node to be deleted is black and has no red children (null nodes are considered black).These are considered the double black cases, the node to be deleted is first replaced by a double black marker. Then we look at the nodes sibling and its children to apply different delete steps. In this case we look at a black sibling with a red right child as illustrated below.

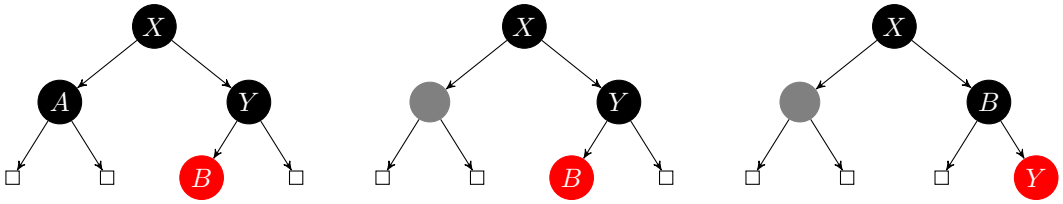


(a) Before deleting node A (b) Replace A with doubleblack (c) Left rotate on sibling's parent

Figure 8: Double black case with black sibling with right red child

This is a terminal case, after the right rotation, the double black node is eliminated, our objective would be to get other cases to this case and then apply these steps to complete the delete.

3. Case 3: Node to be deleted has no red children and at most a single black child (null children are considered black), it has a black sibling with a left red child. This is also a double black case and we apply the following steps.

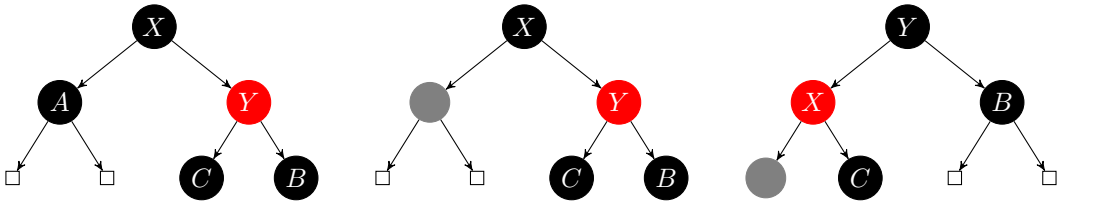


(a) Before deleting node A (b) Replacing A with double black (c) right rotation on sibling

Figure 9: Double black case with black sibling with left red child

We swap the sibling and its left child's color and do a right rotation on the sibling. This transforms the case to case(2). We simple recurse on the double black node and apply case(2) to eliminate the double black condition and complete the delete.

4. Case 4: Node to be deleted has no red child, has at most one black child or no children and its sibling is red. This is another case of double black and we apply the illustrated steps to delete the node.

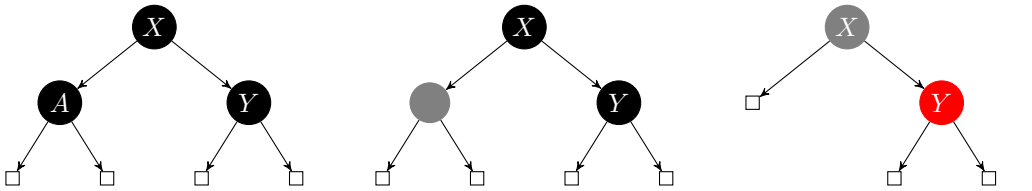


(a) Before deleting node A (b) Replacing A with double black (c) left rotation on sibling's parent

Figure 10: Double black case with red sibling

The sibling and its parent swap colors and left rotation is performed on the parent. In this case the double black node persists and and needs to be recursed untill it reaches a terminal case.

5. Case 5: Node to be deleted is black, has no red child and at most one black child or no children. Its sibling is black and has no red child. This is a double black case and the delete steps are illustrated below.



(a) Before deleting node A (b) Replacing A with double black (c) color sibling red

Figure 11: Double black case with black sibling with no red children

In this case, we simply recolor the sibling and if its parent is black, we make the parent double black and recurse. If the parent is red, we simply color it black and eliminate the double black.

These cover all the possible combinations of cases that could be encountered during a delete operation with the sibling in the right subtree. If the sibling were to be in the left subtree, they will appear as mirror cases to these and we simply apply opposite rotations to complete the delete successfully.

6 Applications

Because of the self balancing property which guarantees O(logn) depth, RBT finds its applications in following software platforms.

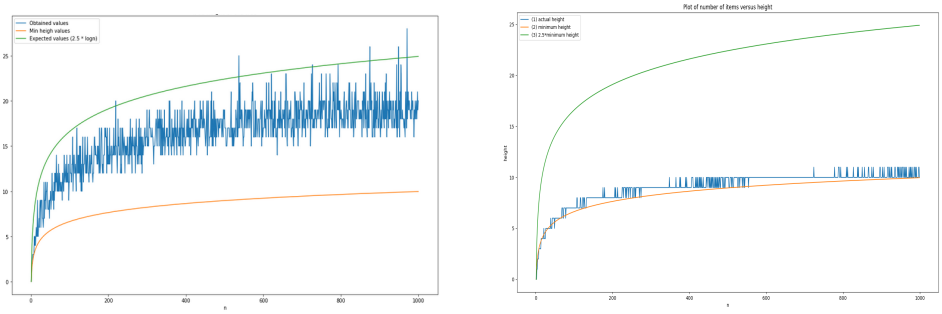
- C++ STL: map, multimap, multiset
- Java: java.util.TreeMap , java.util.TreeSet
- Linux kernel: completely fair scheduler, linux/rbtree.h etc.

7 Results

To see if the RBT fairs better than the BST, we compared the performance of RBT with BST. The following subsections discussed the depth, insert and delete performance of RBT with respect to BST.

7.1 Depth performance

Figure 12a plots the depth of BST with randomly generated values inserted into it. Similarly, Figure 12b plots the depth of RBT. In both the figures, the blue line describes actual depth, red describes the lower bound and green describes the upper bound.



(a) Depth of BST (b) Depth of RBT

Figure 12: Depth performance of BST and RBT

From the Figure 12b, it is evident that the RBT closely traces the log curve which proves that it holds up to its promise.

7.2 Insertion performance

Figure 13 plots the time taken to insert n randomly generated numbers into the RBT and BST.

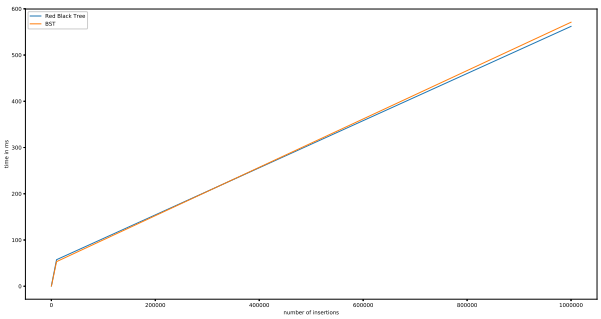


Figure 13: "n" insert performance

It is clear from the figure that for large number of insertions, RBT slightly leads BST. This is because the RBT is balanced. For small number of n, BST performs better. As n increases, (at around n=30000) both takes same time. And for large n, RST fairs better.

7.3 Deletion performance

Figure 14 plots the time taken to delete n values randomly from the RBT and BST.

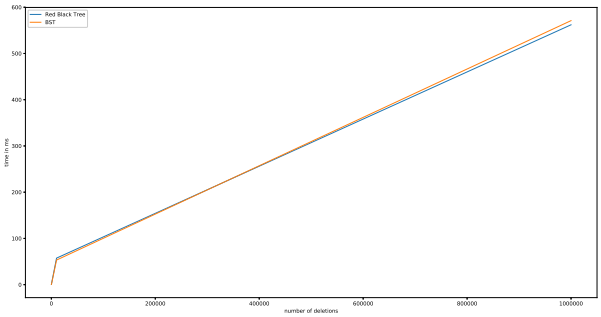


Figure 14: Delete Performance

Similar to n-insertions, n-deletion for RBT takes slightly more time than BST initially. This is because, deletion in RBT involves extra operations. But for larger values of n, because of self balancing, the RBT takes lesser time. This is actually an interesting observation.

8 Conclusion

The red black tree includes an extra bit of information in the node structure to indicate color and adds a few more steps to insert and delete operations such as rotations and recoloring. These operations are of complexity $O(1)$ and are responsible for the re-balancing property of the RBT. The balancing of the tree is not perfect but it is good enough to guarantee search, insert and delete operation in $O(\log n)$ time. Our performance analysis of the RBT showed slightly faster insertion and deletion for over a range of insertions and deletions and its timing got better with larger number of operations when compared to the BST. Also the RBT ensures its performance is $O(\log n$ when working with data in sorted order, which is the main limitation of BST.

References

[1] https://www.cs.auckland.ac.nz/software/AlgAnim/red_black.html

[2] https://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-046j-introduction-to-algorithms-sma-5503-fall-2005/video-lectures/lecture-10-red-black-trees-rotations-insertions-deletions/

[3] http://www.geeksforgeeks.org/red-black-tree-set-1-introduction-2/

[4] http://www.geeksforgeeks.org/red-black-tree-set-2-insert/

[5] http://www.geeksforgeeks.org/red-black-tree-set-3-delete-2/

[6] https://www.youtube.com/watch?v=qvZGUFHWChY