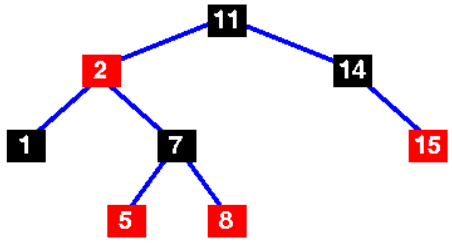
NOTES

Red-Black Tree

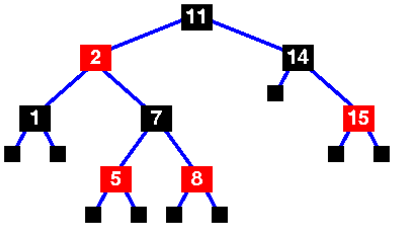
Red-Black tree is a self-balancing binary search tree in which each node contains an extra bit for denoting the color of the node, either red or black. It was created in 1972 by Rudolf Bayer who termed them "symmetric binary B-trees". A red-black tree satisfies the following properties:

1. Red/Black Property: Every node is colored, either red or black.
2. Root Property: The root is black.
3. Leaf Property: Every leaf (NIL) is black. It implies that on any path from the root to a leaf, red nodes must not be adjacent. However, any number of black nodes may appear in a sequence.
4. Red Property: If a red node has children then, the children are always black.
5. Depth Property: For each node, any simple path from this node to any of its descendant leaf has the same black-depth (the number of black nodes).

Each node has the attributes color, key, leftChild, rightChild, parent (except root node).



(1)



(2)

Basic red-black tree with the sentinel nodes added. Implementations of the red-black tree algorithms will usually include the sentinel nodes as a convenient means of flagging that you have reached a leaf node.

Red-black trees maintains a slightly looser height invariant than AVL trees. Because the height of the red-black tree is slightly larger, lookup will be slower in a red-black tree. However, the looser height invariant makes insertion and deletion faster. Also, red-black trees are popular due to the relative ease of implementation.

The number of black nodes on any path from, but not including, a node x to a leaf is called the **black-height** of a node, denoted bh(x). A red-black tree with n internal nodes has height at most 2log(n+1).

How the red-black tree maintains the property of self-balancing?

The red-black color is meant for balancing the tree. The limitations put on the node colors ensure that any simple path from the root to a leaf is not more than twice as long as any other such path. It helps in maintaining the self-balancing property of the red-black tree.

The search-tree operations TREE-INSERT and TREE-DELETE, when runs on a red-black tree with n keys, take O (log n) time. Because they customize the tree, the conclusion may violate the red-black properties. To restore these properties, we must change the color of some of the nodes in the tree and also change the pointer structure.

The insert operation

The goal of the insert operation is to insert key K into tree T, maintaining T's red-black tree properties. A special case is required for an empty tree. If T is empty, replace it with a single black node containing K. This ensures that the root property is satisfied.

If T is a non-empty tree, then we do the following:

1. Use the BST insert algorithm to add K to the tree
2. Color the node containing K red
3. Restore red-black tree properties (if necessary)

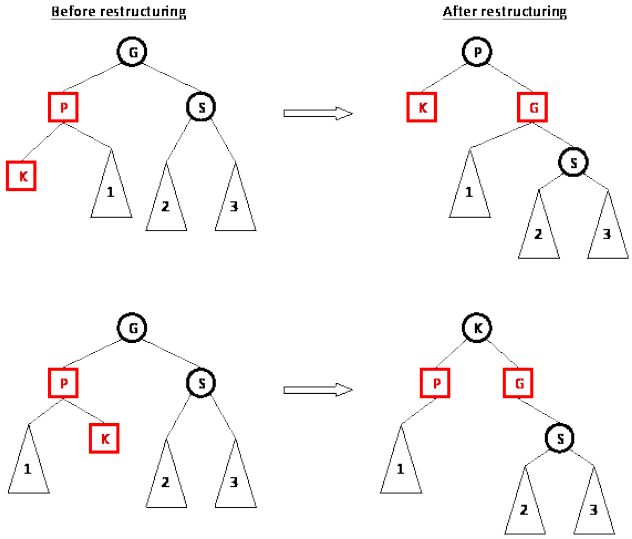
Recall that the BST insert algorithm always adds a leaf node. Because we are dealing with a non-empty red-black tree, adding a leaf node will not affect T's satisfaction of the root property. Moreover, adding a red leaf node will not affect T's satisfaction of the black property. However, adding a red leaf node may affect T's satisfaction of the red property, so we will need to check if that is the case and, if so, fix it (step 3). In fixing a red property violation, we will need to make sure that we don't end up with a tree that violates the root or black properties.

For step 3 for inserting into a non-empty tree, what we need to do will depend on the color of K's parent. Let P be K's parent. We need to consider two cases:

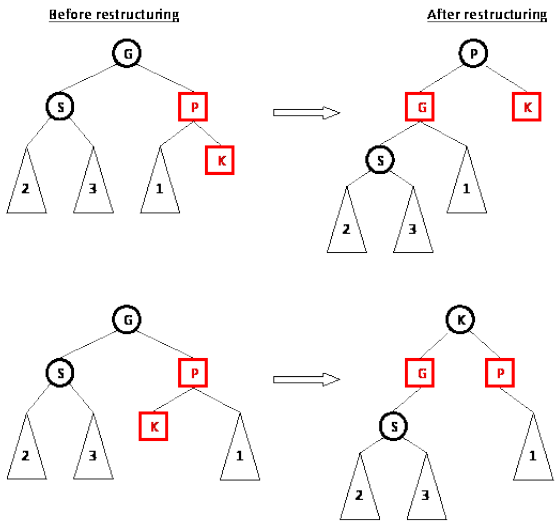
Case 1: If K's parent P is black, then the addition of K did not result in the red property being violated, so there's nothing more to do.

Case 2: If K's parent P is red, then P now has a red child, which violates the red property. Note that P's parent, G, (K's grandparent) must be black. In order to handle this **double-red situation**, we will need to consider the color of G's other child, that is, P's sibling, S. (Note that S might be null, i.e., G only has one child and that child is P.)

Case 2(a): If P's sibling **S is black or null**, then we will do a trinode restructuring of K (the newly added node), P (K's parent), and G (K's grandparent). To do a restructuring, we first put K, P, and G in order; let's call this order K, P, and G. We then make P the parent of K and G, color P black, and color K and G red. We also need to make sure that any subtrees of P and S (if S is not null) end up in the appropriate place once the restructuring is done. There are four possibilities for the relative ordering of K, P, and G. The way a restructuring is done for each of the possibilities is shown below.The first two possibilities are:



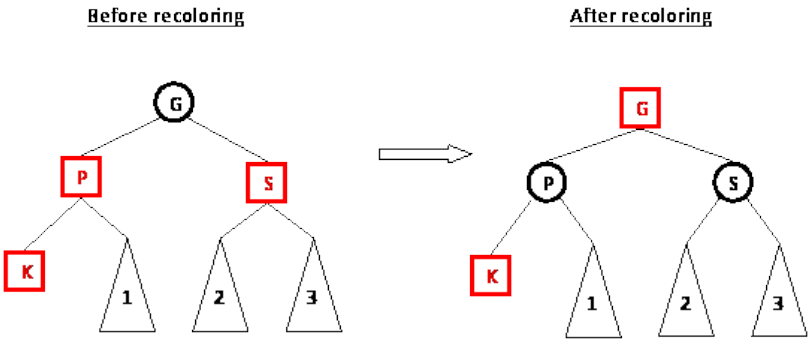
If S is null, then in the pictures above, S (and it's subtrees labelled 2 and 3) would be replaced with nothing (i.e., null). The second two possibilites are mirror-images of the first two possibilites:



Once a restructuring is done, the double-red situation has been handled and there's nothing more to do (you should convince yourself, by looking at the diagrams above, that restructuring will not result in a violation of the black property).

Summarization
P:Left K:Left=>Make P root with black color and K and G will be left and right children with red color
P:Right K:Right=>Make P root with black color and G and K will be left and right children with red color
P:Left K:Right=>Make K root with black color and P and G will be left and right children with red color
P:Right K:Left=>Make K root with black color and G and P will be left and right children with red color

Case 2(b): P's sibling **S is red** If P's sibling S is red, then we will do a **recoloring of P, S, and G**: the color of **P and S is changed to black and the color of G is changed to red (unless G is the root, in which case we leave G black to preserve the root property).**



Recoloring does not affect the black property of a tree: the number of black nodes on any path that goes through P and G is unchanged when P and G switch colors (similarly for S and G). But, the recoloring may have introduced a double-red situation between G and G's parent. If that is the case, then we recursively handle the double-red situation starting at G and G's parent (instead of K and K's parent).

Complexity of insertion

Inserting a key into a non-empty tree has three steps. In the first step, the BST insert operation is performed. The BST insert operation is O(height of tree) which is O(log N) because a red-black tree is balanced. The second step is to color the new node red. This step is O(1) since it just requires setting the value of one node's color field. In the third step, we restore any violated red-black properties.

Restructuring is O(1) since it involves changing at most five pointers to tree nodes. Once a restructuring is done, the insert algorithm is done, so at most 1 restructuring is done in step 3. So, in the worst-case, the restructuring that is done during insert is O(1).

Changing the colors of nodes during recoloring is O(1). However, we might then need to handle a double-red situation further up the path from the added node to the root. In the worst-case, we end up fixing a double-red situation along the entire path from the added node to the root. So, in the worst-case, the recoloring that is done during insert is O(log N) (= time for one recoloring * max number of recolorings done = O(1) * O(log N)). Thus, the third step (restoration of red-black properties) is O(log N) and the total time for insert is O(log N).

Create a red black tree with following sequence 8 18 5 15 17 25 40 and 80.

Step-1

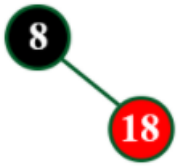
insert (8)

Tree is Empty. So insert newNode as Root node with black color.

Step-2

insert (18)

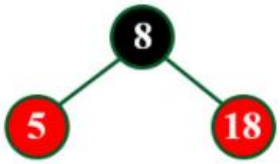
Tree is not Empty. So insert newNode with red color.



Step-3

insert (5)

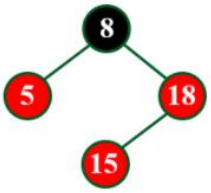
Tree is not Empty. So insert newNode with red color.



Step-4

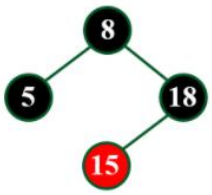
insert (15)

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

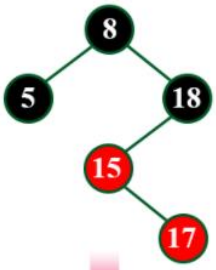


After Recolor operation, the tree is satisfying all Red Black Tree properties.

Step-5

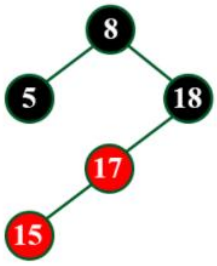
insert (17)

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

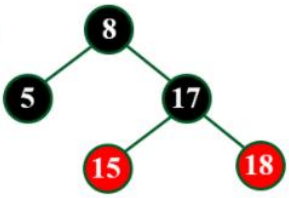


Here there are two consecutive Red nodes (15 & 17).
The newnode's parent sibling is NULL. So we need rotation.
Here, we need LR Rotation & Recolor.

After Left Rotation



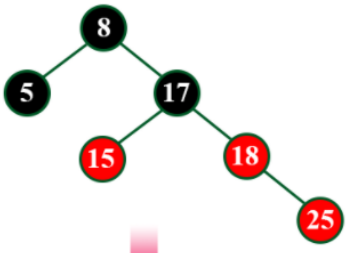
After Right Rotation & Recolor



Step-6

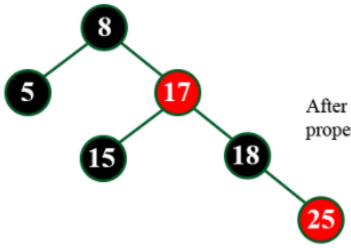
insert (25)

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

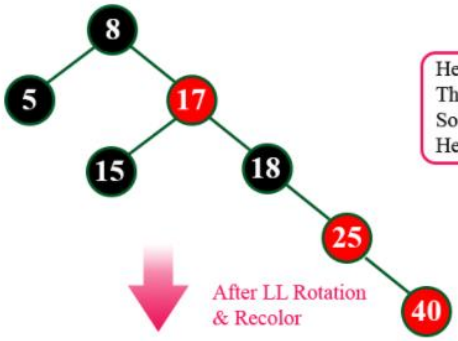


After Recolor operation, the tree is satisfying all Red Black Tree properties.

Step-7

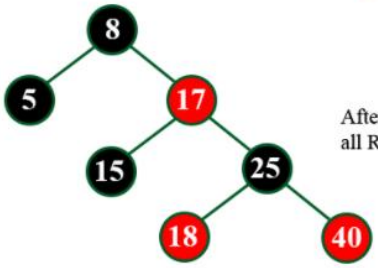
insert (40)

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



Here there are two consecutive Red nodes (25 & 40).
The newnode's parent sibling is NULL
So we need a Rotation & Recolor.
Here, we use LL Rotation and Recheck.

After LL Rotation
& Recolor

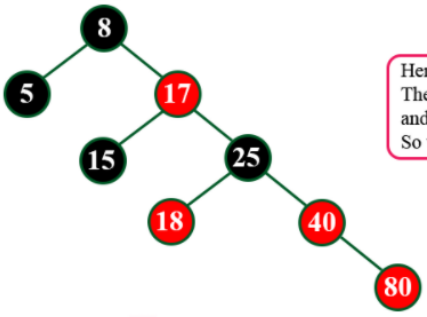


After LL Rotation & Recolor operation, the tree is satisfying all Red Black Tree properties.

Step-8

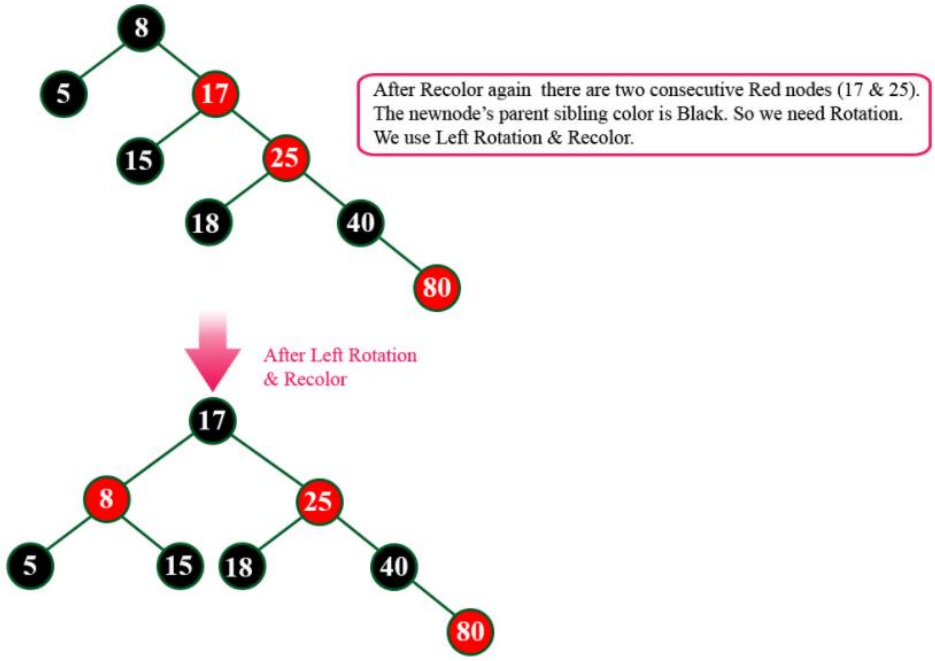
insert (80)

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



Here there are two consecutive Red nodes (40 & 80).
The newnode's parent sibling color is Red
and parent's parent is not root node.
So we use RECOLOR and Recheck.

After Recolor



Dijkstra's Algorithm

Dijkstra's Algorithm is to find the shortest path between nodes in a graph. Particularly, you can find the shortest path from a node (called the "source node") to all other nodes in the graph, producing a shortest-path tree. This algorithm is used in GPS devices to find the shortest path between the current location and the destination. It has broad applications in industry, especially in domains that require modeling networks.

Basics of Dijkstra's Algorithm

1. Dijkstra's Algorithm basically starts at the node that you choose (the source node) and it analyzes the graph to find the shortest path between that node and all the other nodes in the graph.
2. The algorithm keeps track of the currently known shortest distance from each node to the source node and it updates these values if it finds a shorter path.
3. Once the algorithm has found the shortest path between the source node and another node, that node is marked as "visited" and added to the path.
4. The process continues until all the nodes in the graph have been added to the path. This way, we have a path that connects the source node to all other nodes following the shortest path possible to reach each node.

[Example from PPT]

An Example

```
import sys
class Graph():
        def __init__(self, vertices):
                self.V = vertices
                self.graph = [[0 for column in range(vertices)] for row in range(vertices)]

        def printSolution(self, dist):
                print ("Vertex tDistance from Source")
                for node in range(self.V):
                        print (node, "t", dist[node])

        def minDistance(self, dist, sptSet):
                min = sys.maxsize
                for v in range(self.V):
                        if dist[v] < min and sptSet[v] == False:
                                min = dist[v]
                                min_index = v
                return min_index
```

```
        def dijkstra(self, src):
                dist = [sys.maxsize] * self.V
                dist[src] = 0
                sptSet = [False] * self.V
                for cout in range(self.V):
                        u = self.minDistance(dist, sptSet)
                        sptSet[u] = True
                        for v in range(self.V):
                                if self.graph[u][v] > 0 and sptSet[v] == False and dist[v] > dist[u] +
self.graph[u][v]:
                                        dist[v] = dist[u] + self.graph[u][v]

                self.printSolution(dist)

g = Graph(5)
g.graph = [[0,10,3,0,0],[0,0,1,2,0],[0,4,0,8,2],[0,0,0,0,7],[0,0,0,9,0]];
g.dijkstra(0)
```