Red-Black Trees: Properties, Operations, and Applications

Vignan's Foundation for Science,
Technology & Research (Deemed to be University
M. Bhavani Prasad (231FA04139), Batch-04
Submitted to: Mr. S. Suresh Babu

I. Introduction

Definition: A Red-Black Tree is a self-balancing binary search tree in which each node is colored either red or black, and the tree follows certain rules about coloring that keep it balanced.

II. Properties

- Each node is either Red or Black.
- The root is always Black.
- All leaves (NULL/NIL pointers) are considered Black.
- No two Red nodes can be adjacent (a Red node cannot have a Red parent or child).
- Every path from a node to its descendant NIL nodes must have the same number of Black nodes (Black-Height Property).

III. Insertion Rules

- Root node color should be black.
- New leaf node should be red.
- If parent of new node is black then continue.
- If parent of new node is red:
  - Check colour of parent's sibling.
  - Case 1: If sibling is black or no sibling then do suitable rotation & recolour.
  - Case 2: If sibling is red, recolour parent & parent's sibling. If parent's parent is not root node then recolour & recheck else continue.

Note:

- Root is black.
- No Red–Red conflict.
- Number of black nodes from any leaf to root is same.

A. *Example: Insert 10,18,7,15,16,30*

Insertion Rules (Step Summary):

- Root node color should be black; new leaf node should be red.
- If parent is black, continue.
- If parent is red: check parent's sibling.
- If sibling is black or absent: rotate suitably and recolor.
- If sibling is red: recolor parent and sibling; if grandparent is not root, recolor and recheck.

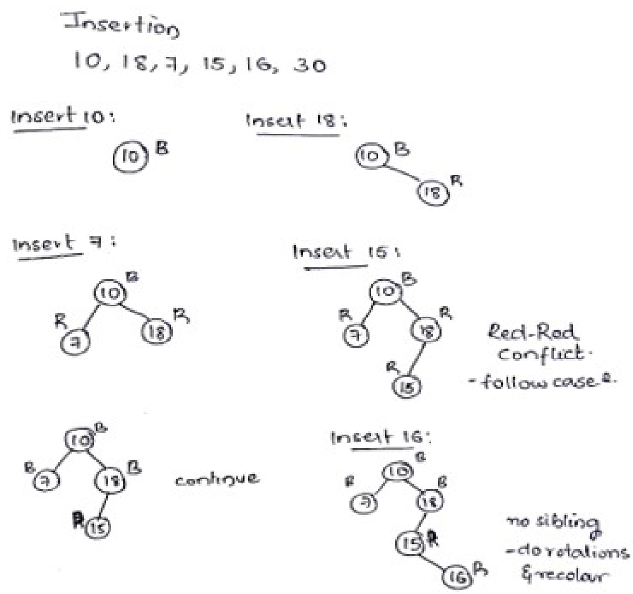

Fig. 1. Insertion process for 10, 18, 7, 15, 16, 30.

IV. Rotations

Rotations restore balance:

- **Left Rotation:** When right child is red-heavy.
- **Right Rotation:** When left child is red-heavy.
- **Double Rotations:** Combination of left and right for zig-zag cases.

V. Advantages and Disadvantages

Advantages:

- Height is strictly balanced (bounded within a constant factor of optimal).
- Guarantees $O(\log n)$ operations.
- Efficient memory use.

Disadvantages:

- More complex to implement than a basic BST and somewhat intricate compared to AVL.
- AVL trees are better for frequent searches (faster lookups due to stricter balance).
- RBTs are often better for frequent insertions/deletions (fewer rotations on average).

VI. Deletion

Deletion in a Red-Black Tree follows standard BST deletion first, then fixes any violation of Red-Black properties:

- If the deleted node is Red, no fixing is needed.
- If the deleted node is Black and replaced by a Red child, the child is recolored Black.
- If the deleted node is Black with a Black child (or NIL), a "double black" problem arises and must be fixed using recoloring and rotations.

A. Example: Delete 18

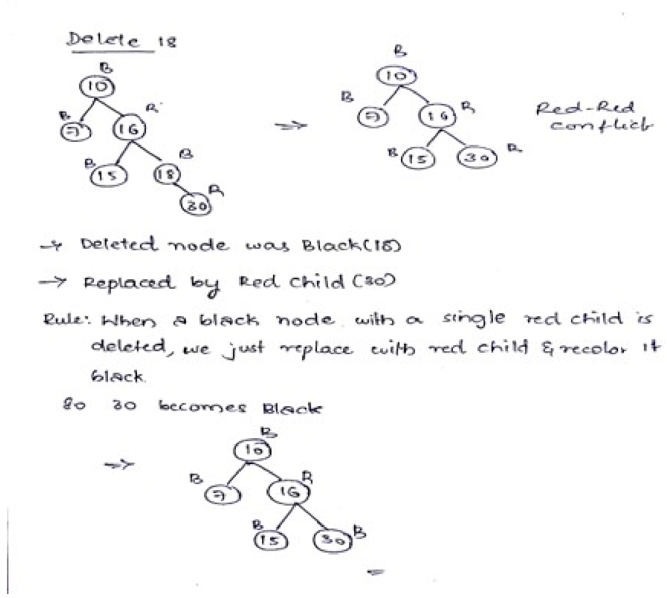

Fig. 2. Deletion of node 18 in a Red-Black Tree.

VII. Searching

Searching is the same as in a BST. Time complexity is $O(\log n)$ because the tree is balanced. Colors don't affect search logic—only balancing.

Example: Search for 15 in the tree rooted at 10: path $10 \rightarrow 16 \rightarrow 15$.

VIII. Update

- If only the value changes: no structural change, no rebalancing.
- If the key changes: delete the old key, insert the new key, and rebalance.

Example: Values $[10, 20, 30]$. Update $10 \rightarrow 25$.

IX. Real-Life Applications

- **Databases:** Used in indexing (PostgreSQL, Oracle).
- **Operating Systems:** Linux kernel scheduling, memory management.
- **Networking:** Routing tables, IP lookup.
- **Language Libraries:** C++ STL `map`, `set`, `multimap`, `multiset`.
- **File Systems:** NTFS, ext3, ext4.

Example:- Search of 15

```
      10(B)
      /   \
  7(B)    16(R)
           /   \
       15(B)   30(B)
```

1. Root = 10(B)
   15>10 move right
2. Arrive at 16(R)
   16>15 move left
3. Arrive at 15(B)
   15=15 element found

Path:10-16-15

Fig. 3. Searching for 15 in a Red-Black Tree.

Example:-

Let the values be [10,20,30] and let the update value be 25 in place of removing 10

```
   20(B)
   /   \
10(R)   30(R)
```

Update key 10 ⟶ 25:-

Delete 10

```
   20(B)
       \
       30(R)
```

Insert 25 (as Red):-

```
  20(B)
       \
       30(R)
       /
    25(R)
```

Fig. 4. Updating 10 to 25 in a Red-Black Tree.

X. Comparison with Other Trees (Step-by-Step)

1. **Balance Criterion:** AVL maintains stricter balance (smaller height); RBT is looser but still $O(\log n)$. B-Tree balances nodes across multiple keys per node to minimize disk I/O.
2. **Search Performance:** AVL often yields slightly faster lookups due to smaller height. RBT lookups are still $O(\log n)$ and very good in practice. B-Tree excels when data resides on disk/SSD.
3. **Insertion/Deletion Cost:** RBT typically performs fewer rotations than AVL under mixed workloads (good for frequent updates). B-Tree groups rebalancing at node/block granularity.
4. **Memory/Layout:** AVL/RBT store one key per node (pointer-heavy). B-Tree stores many keys per node (cache/disk-friendly).
5. **Use Cases:** RBTs: OS kernels, language libraries; AVLs: search-heavy in-memory indices; B-Trees: databases and filesystems.

XI. Python Implementation (Small)

```python
RED = True
BLACK = False

class Node:
    def __init__(self, data):
        self.data = data
        self.color = RED   # new node is red
        self.left = None
        self.right = None
        self.parent = None

class RedBlackTree:
    def __init__(self):
        self.NIL = Node(None)
        self.NIL.color = BLACK
        self.root = self.NIL

    def rotate_left(self, x):
        y = x.right
        x.right = y.left
        if y.left != self.NIL:
            y.left.parent = x
        y.parent = x.parent
        if x.parent is None:
            self.root = y
        elif x == x.parent.left:
            x.parent.left = y
        else:
            x.parent.right = y
        y.left = x
        x.parent = y

    def rotate_right(self, y):
        x = y.left
        y.left = x.right
        if x.right != self.NIL:
            x.right.parent = y
        x.parent = y.parent
        if y.parent is None:
            self.root = x
        elif y == y.parent.right:
            y.parent.right = x
        else:
            y.parent.left = x
        x.right = y
        y.parent = x

    def fix_insert(self, k):
        while k.parent and k.parent.color == RED:
            if k.parent == k.parent.parent.left:
                u = k.parent.parent.right
                if u and u.color == RED:
                    k.parent.color = BLACK
                    u.color = BLACK
                    k.parent.parent.color = RED
                    k = k.parent.parent
                else:
                    if k == k.parent.right:
                        k = k.parent
                        self.rotate_left(k)
                    k.parent.color = BLACK
                    k.parent.parent.color = RED
                    self.rotate_right(k.parent.parent)
            else:
                u = k.parent.parent.left
                if u and u.color == RED:
                    k.parent.color = BLACK
                    u.color = BLACK
                    k.parent.parent.color = RED
                    k = k.parent.parent
                else:
                    if k == k.parent.left:
                        k = k.parent
                        self.rotate_right(k)
                    k.parent.color = BLACK
                    k.parent.parent.color = RED
                    self.rotate_left(k.parent.parent)
        self.root.color = BLACK

    def insert(self, key):
        node = Node(key)
        node.left = self.NIL
        node.right = self.NIL
        y = None
        x = self.root
        while x != self.NIL:
            y = x
            if node.data < x.data: x = x.left
            else: x = x.right
        node.parent = y
        if y is None: self.root = node
        elif node.data < y.data: y.left = node
        else: y.right = node
        if node.parent is None:
            node.color = BLACK
            return
        if node.parent.parent is None:
            return
        self.fix_insert(node)

    def inorder(self, node=None):
        if node is None: node = self.root
        if node != self.NIL:
            self.inorder(node.left)
            print(f"{node.data} ({'R' if node.color else 'B'})", end=" ")
            self.inorder(node.right)

    def search(self, key, node=None):
        if node is None: node = self.root
        if node == self.NIL or key == node.data: return node != self.NIL
        if key < node.data: return self.search(key, node.left)
        return self.search(key, node.right)

if __name__ == "__main__":
    rbt = RedBlackTree()
    for v in [10, 18, 7, 15, 16, 30]: rbt.insert(v)
    print("Inorder Traversal:"); rbt.inorder(); print()
    print("Search 15:", rbt.search(15))
    print("Search 99:", rbt.search(99))
```

Output (example):

```
Inorder Traversal:
7 (R) 10 (B) 15 (B) 16 (R) 18 (B) 30 (R)
Search 15: True
Search 99: False
```

XII. Conclusion

Red-Black Tree is a self-balancing Binary Search Tree. It maintains balance using coloring rules and rotations, guarantees

$O(\log n)$ operations, prevents skewing, and provides a practical balance between search speed and update cost compared to AVL. Its applications span OS kernels, databases, and standard libraries.

Acknowledgment

Thank you.