Red-Black Trees: Properties, Operations, and Applications

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 (called 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

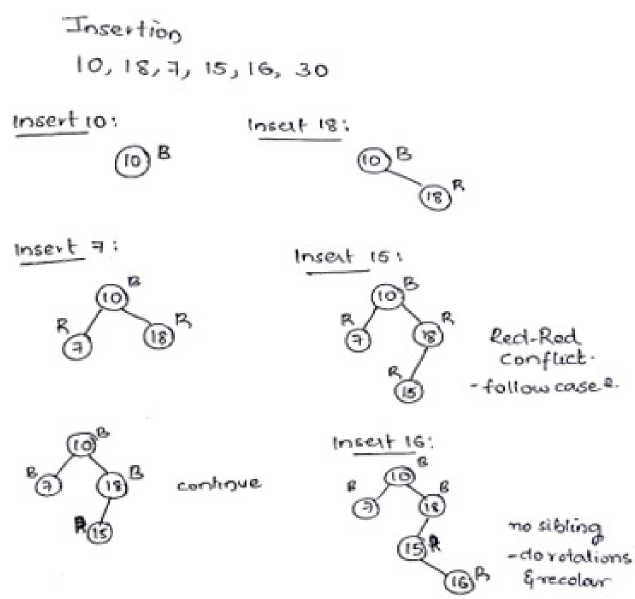

Fig. 1. Insertion process for 10, 18, 7, 15, 16, 30

IV. 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

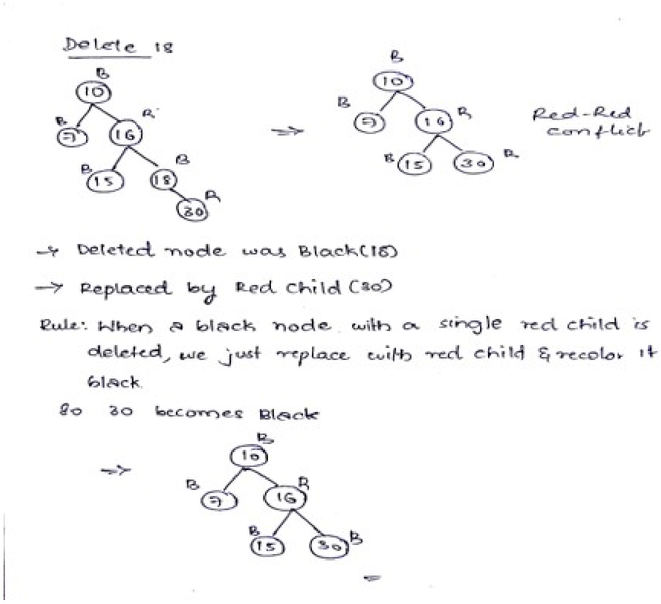

Fig. 2. Deletion of node 18 in a Red-Black Tree

V. Searching

Searching is done exactly like 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

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 Red-Black Tree

VI. Update

- If only the value changes $\rightarrow$ no change to structure, no rebalancing.
- If the key changes $\rightarrow$ delete the old key, insert the new key, and rebalance.

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

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 Red-Black Tree

VII. 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.

VIII. Comparison with Other Trees

Feature	Red-Black Tree	AVL Tree	B-Tree
Balance	Loosely balanced	Strictly balanced	Disk-balanced
Insertion/Deletion	Faster, fewer rotations	Slower, more rotations	Optimized for bulk
Use Cases	OS, DBs, libraries	Search-heavy apps	Databases, FS

TABLE I
Comparison of Red-Black Tree with AVL and B-Tree

IX. Python Implementation

```python
RED = True
BLACK = False

class Node:
    def __init__(self, data):
        self.data = data
        self.color = 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

    # Left rotate
    def rotate_left(self, x):
        ...

    # Right rotate
    def rotate_right(self, y):
        ...

    # Fix violations
    def fix_insert(self, k):
        ...

    def insert(self, key):
        ...

    def inorder(self, node=None):
        ...

    def search(self, key, node=None):
        ...
# --- MAIN ---
if __name__ == "__main__":
    rbt = RedBlackTree()
    for val in [10, 18, 7, 15, 16, 30]:
        rbt.insert(val)
    print("Inorder Traversal:")
    rbt.inorder()
    print("\nSearch 15:", rbt.search(15))
    print("Search 99:", rbt.search(99))
```

Output:

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

X. 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 balance between fast operations and easy implementation compared to AVL trees.

Acknowledgment

Thank you.