Task 1: Reflection

A Reflection on Self-Balancing Trees

Revisiting the topic of trees, my initial understanding was grounded in the core promise of the Binary Search Tree (BST): its potential to achieve an impressive O(log n) time complexity for fundamental operations like insertion, deletion, and searching. This efficiency makes it seem like the ideal choice over simpler structures like sorted arrays or linked lists, which struggle with either slow searches or slow insertions/deletions.

However, the most crucial insight I gained from this material was the significant flaw in a standard BST's design: the problem of unbalancing. I now understand that if data is inserted in a sorted or near-sorted order, a BST can degenerate into a structure that behaves just like a linked list. This completely negates its primary advantage, as the operational complexity plummets to a linear O(n), making it no better than the structures it was meant to improve upon.

This realization led me to the core solution: self-balancing binary search trees. It was fascinating to learn that the central mechanism to achieve this is the tree rotation, a clever operation that can restructure the tree to reduce its height without violating the fundamental rules of a BST.

Delving deeper, I explored two key implementations of this concept:

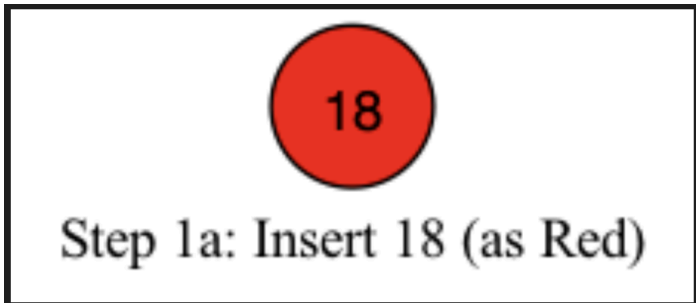
1. AVL Trees: What struck me about AVL trees was their strict and elegant rule—the height-balance property. The idea that for any node, the heights of its children can differ by at most one is a powerful guarantee of balance. I learned that this strictness means the tree must be checked and potentially rebalanced using rotations after every single insertion or deletion, ensuring it never becomes lopsided.
2. Red-Black (RB) Trees: The RB-Tree presented a different, more intricate approach. Instead of strict height differences, it uses a system of coloring nodes red or black and enforces a set of rules to maintain balance. The key takeaway for me was that these rules, while complex, cleverly ensure that the longest path from the root to a leaf is no more than twice as long as the shortest path. This guarantees a logarithmic height and, therefore, efficient performance.

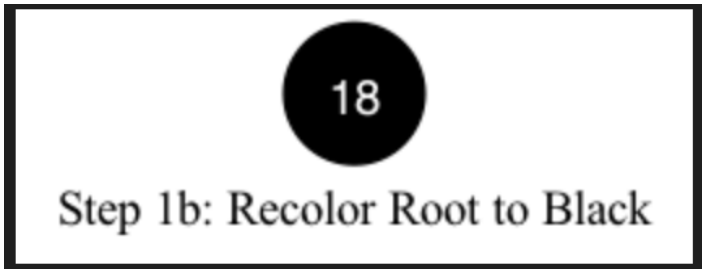
In conclusion, my perspective has shifted significantly. I now see that while a standard BST is a foundational concept, its vulnerability to unbalancing makes it unreliable for many real-world applications. The true power lies in self-balancing variants like AVL and RB trees. Although their implementation details, particularly the various rotation cases and fixup procedures, are complex, I now appreciate that this complexity is a necessary trade-off to guarantee the consistent O(log n) performance that makes tree data structures so valuable and efficient.

Tasks 2-4 have been submitted as code solutions

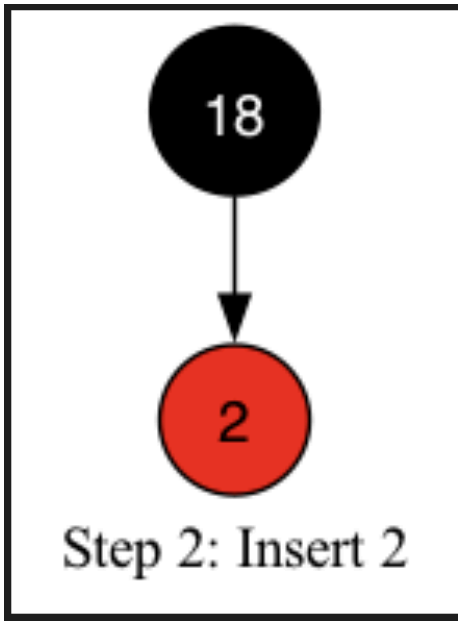
Task 5

Inserting 18,

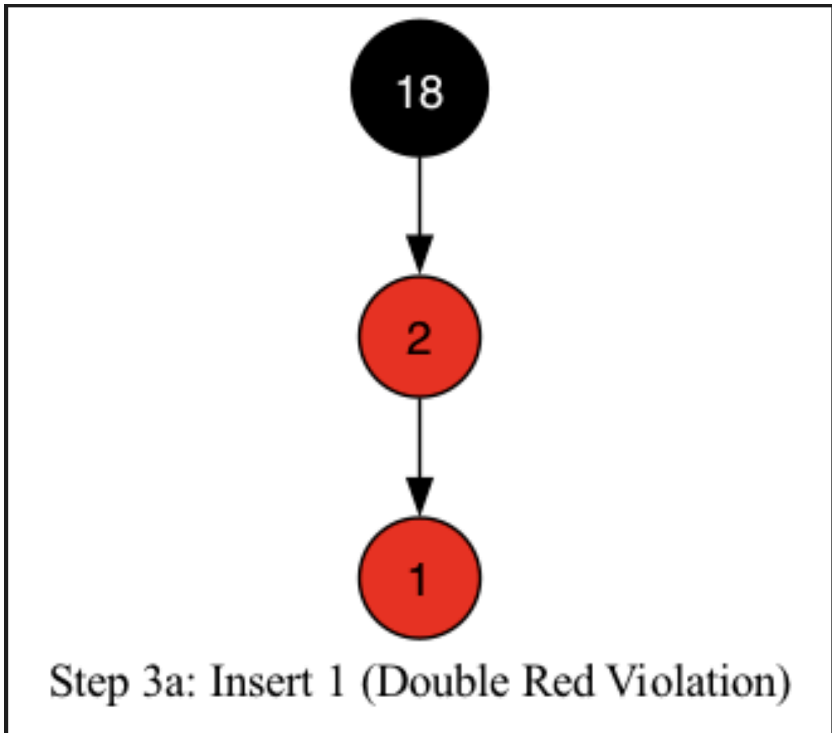




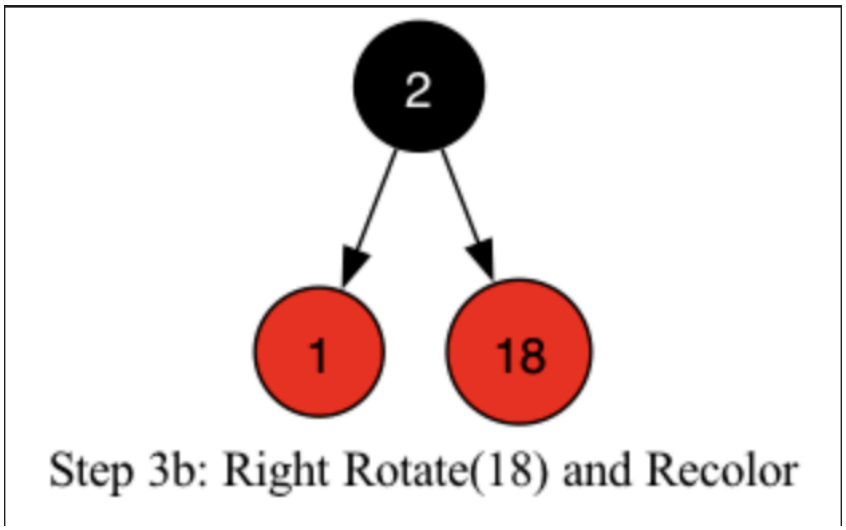
Inserting 2,



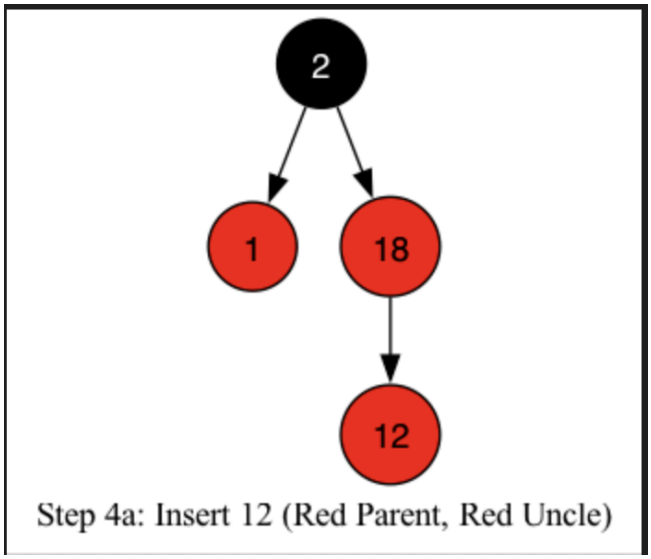
Inserting 1,



To fix the violation, we perform a Right Rotation on the grandparent (18) and then recolor the nodes. The new root of this subtree (2) becomes Black, and its children (1 and 18) become Red.

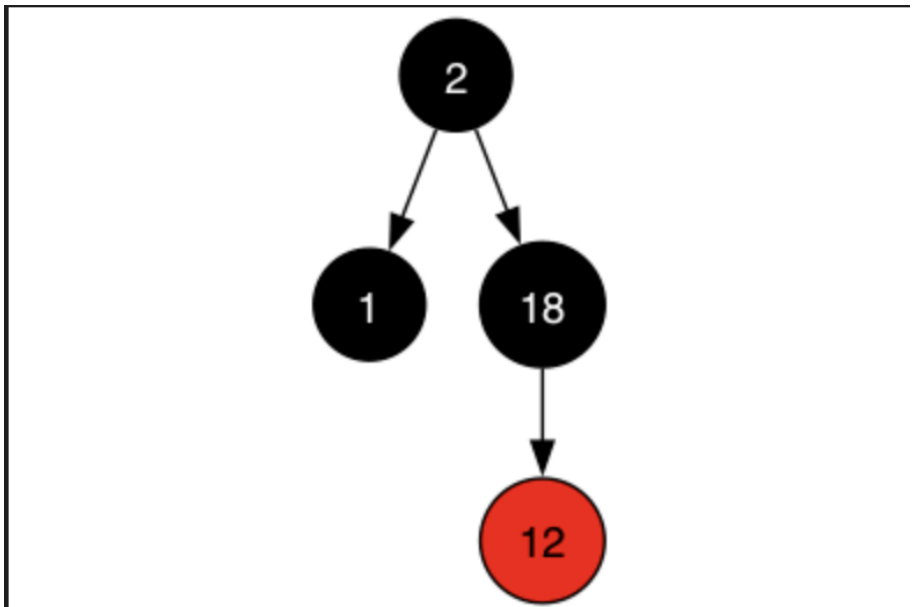


Inserting 12,



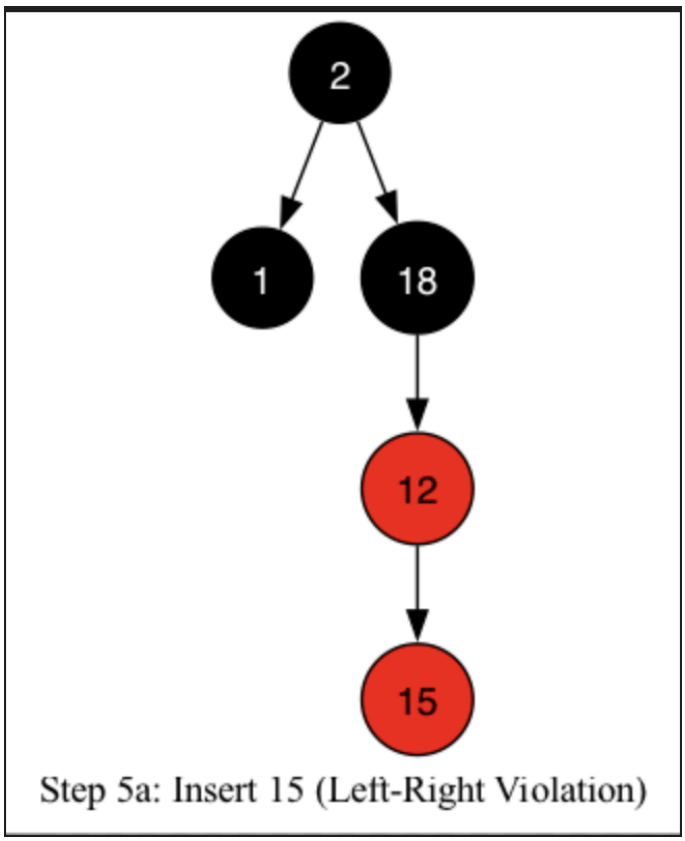
Step 4a: Insert 12 (Red Parent, Red Uncle)

The fix for a "Red uncle" case is to recolor. The parent (18) and uncle (1) are colored Black. The grandparent (2) is colored Red. However, since the grandparent (2) is also the root of the tree, it is immediately forced back to Black to satisfy Rule 2.



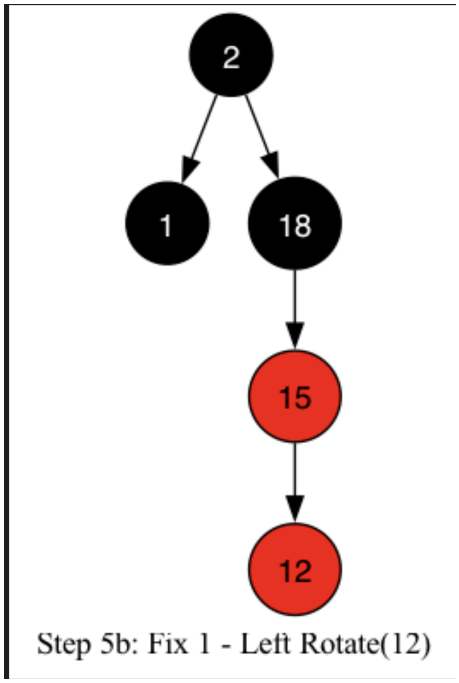
Step 4b: Recolor Parent, Uncle, and Grandparent

Inserting 15,



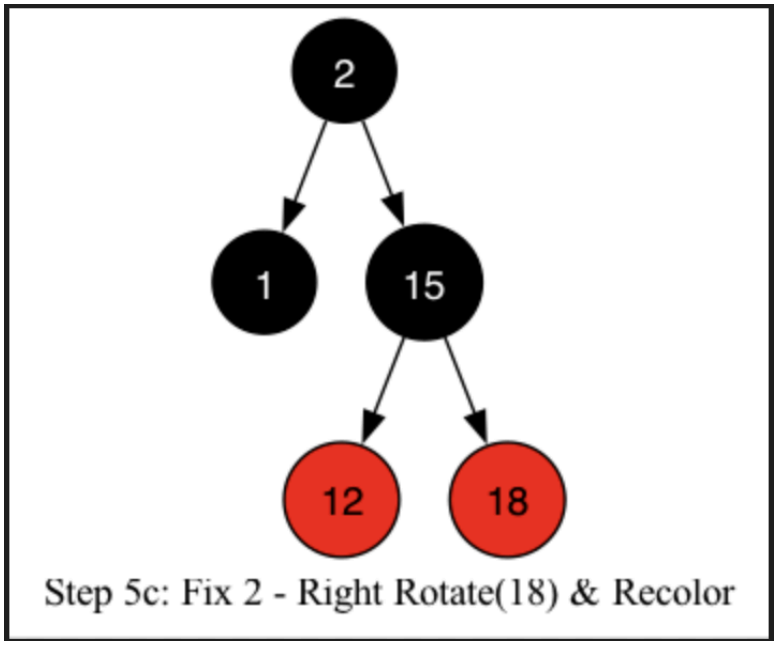
Step 5a: Insert 15 (Left-Right Violation)

The first part of fixing a "Left-Right" case is to perform a Left Rotation on the parent node (12). This rotation transforms the problem into a simpler "Left-Left" case.



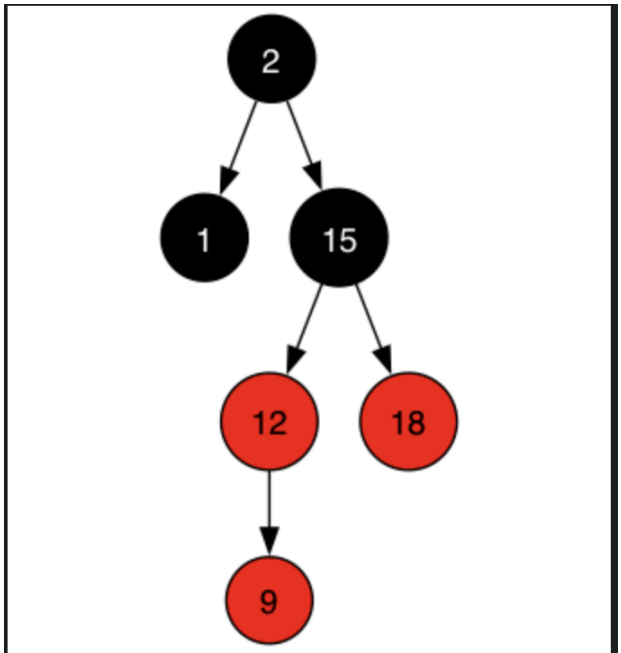
Step 5b: Fix 1 - Left Rotate(12)

Now we solve the newly formed "Left-Left" case. We perform a Right Rotation on the grandparent (18) and then recolor. The new root of this subtree (15) becomes Black, and its children (12 and 18) become Red.



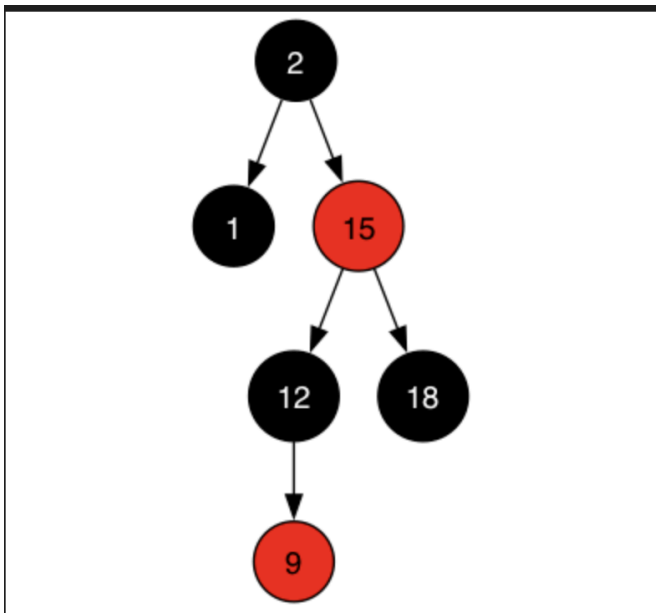
Step 5c: Fix 2 - Right Rotate(18) & Recolor

Inserting 9,



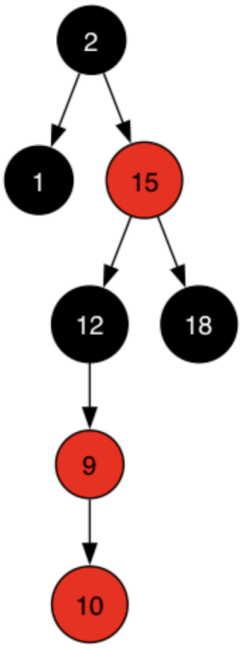
Step 6a: Insert 9 (Red Parent, Red Uncle)

The fix for a "Red uncle" case is to recolor. The parent (12) and uncle (18) are colored Black. The grandparent (15) is colored Red. Since the new Red node (15) has a Black parent (2), no further fixes are needed.



Step 6b: Recolor Parent, Uncle, and Grandparent

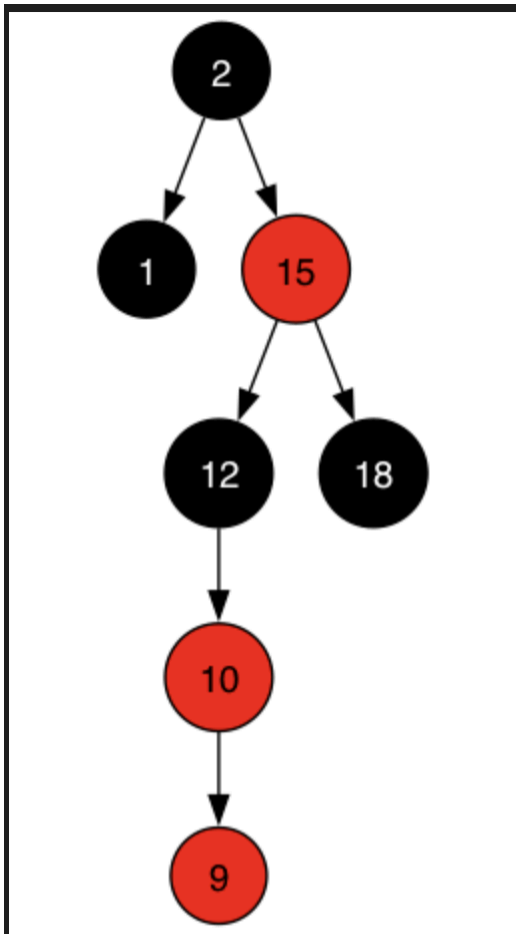
Inserting 10,



Step 7a: Insert 10 (Left-Right Violation)

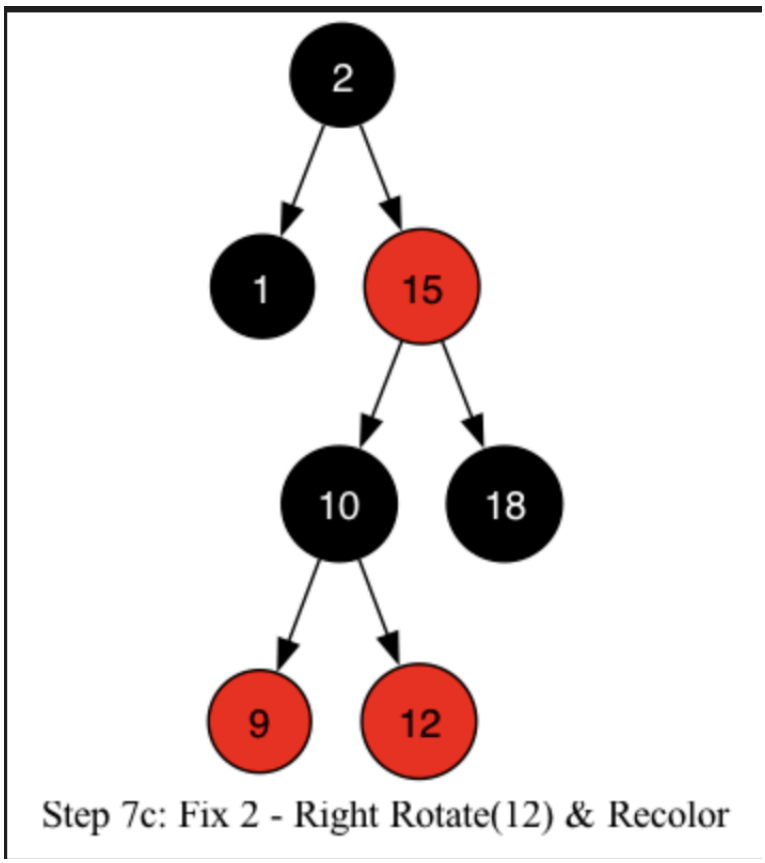
The value 10 is inserted as the Red right child of 9. This creates a "double red" violation because its parent, 9, is also Red. The uncle node is NIL (Black), which makes this a "Left-Right" case relative to the grandparent (12).

First, we perform a Left Rotation on the parent node (9). This rotation resolves the "zig-zag" shape and transforms the problem into a simpler "Left-Left" case.



Step 7b: Fix 1 - Left Rotate(9)

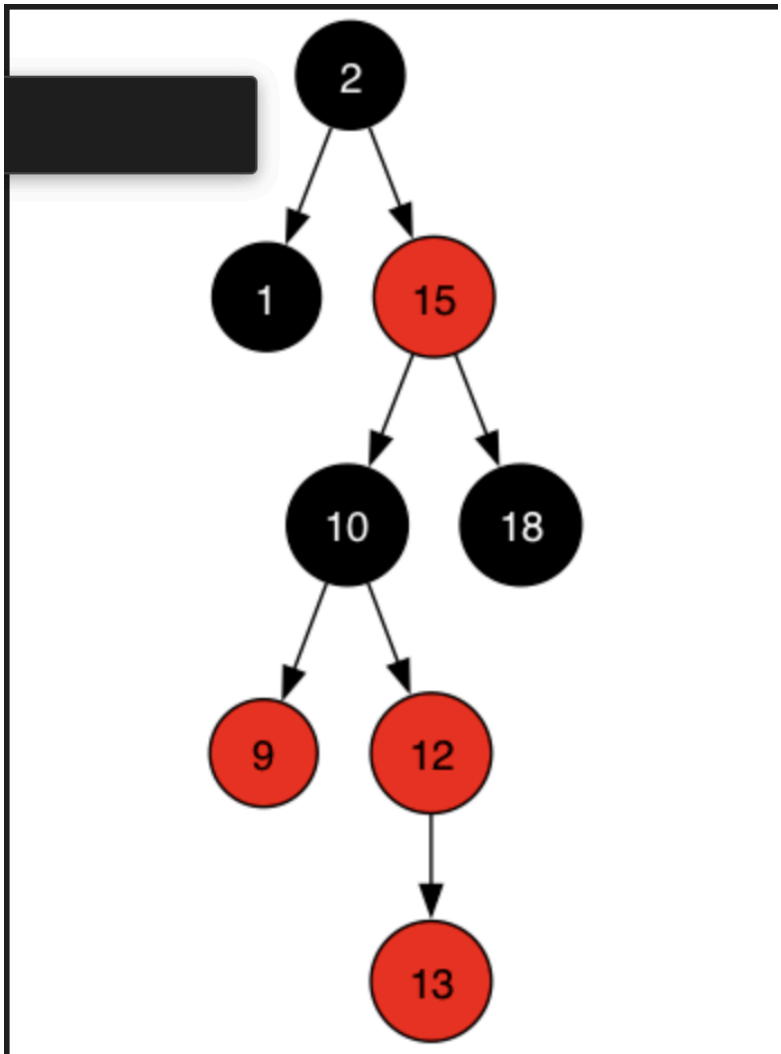
Now we solve the "Left-Left" case by performing a Right Rotation on the grandparent (12). We then recolor the new root of this subtree (10) to be Black and its children (9 and 12) to be Red.



Step 7c: Fix 2 - Right Rotate(12) & Recolor

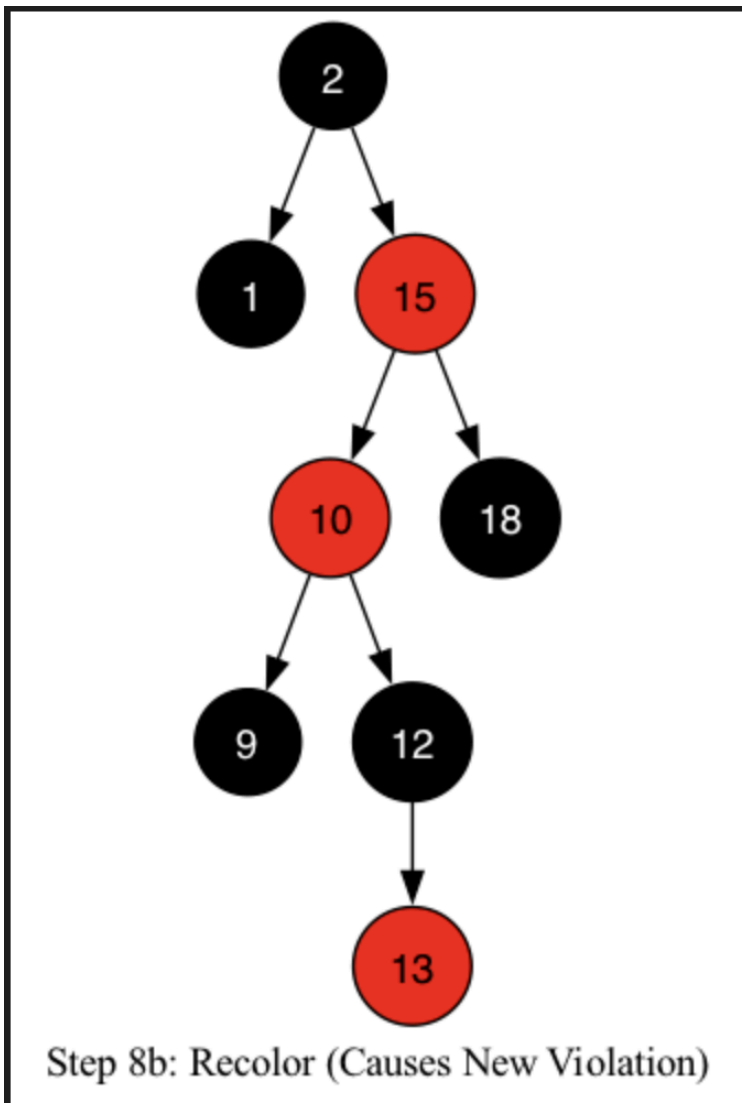
Inserting 13,

The value 13 is inserted as the Red right child of 12. This creates a "double red" violation with its parent 12(R). The uncle node (9) is also Red, which is Insertion Case 1.



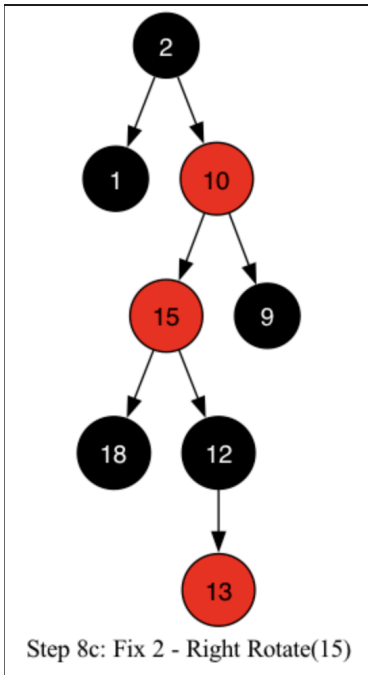
Step 8a: Insert 13 (Red Parent, Red Uncle)

We fix the first violation by recoloring the parent 12 and uncle 9 to Black, and the grandparent 10 to Red. This, however, creates a new double red violation, as 10(R) now has a Red parent, 15(R). The new uncle (1) is Black, creating a "Right-Left" case.



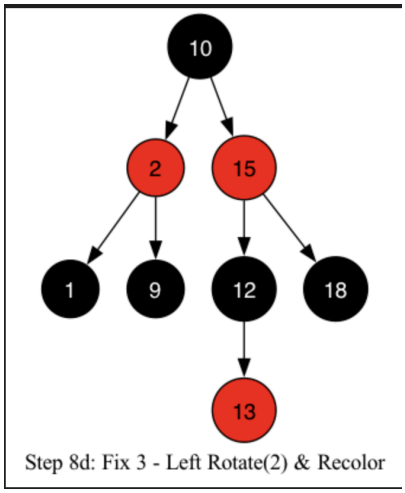
Step 8b: Recolor (Causes New Violation)

To solve the "Right-Left" case, we first perform a Right Rotation on the parent (15). This transforms the problem into a "Right-Right" case.



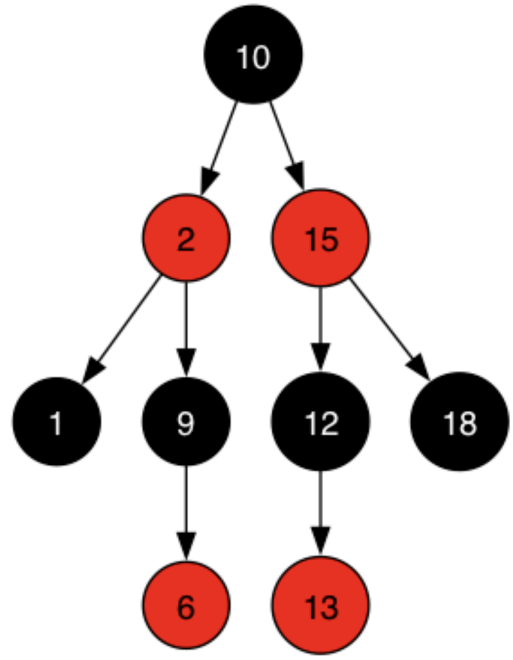
Step 8c: Fix 2 - Right Rotate(15)

Finally, we solve the "Right-Right" case with a Left Rotation on the grandparent (2). We then recolor the new root (10) to Black and its children (2 and 15) to Red.



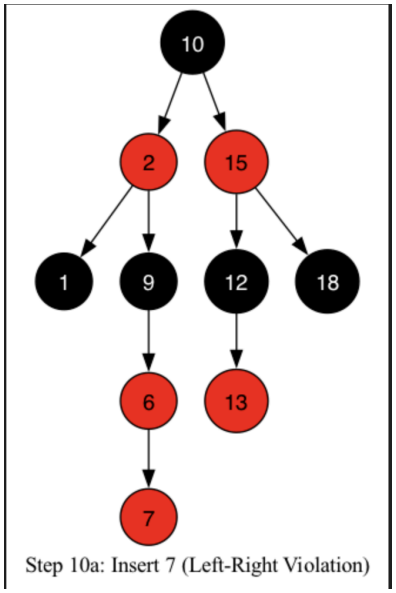
Step 8d: Fix 3 - Left Rotate(2) & Recolor

Inserting 6,
The value 6 is inserted as the Red left child of 9. Since the new Red node (6) has a Black parent (9), no Red-Black Tree rules are violated. No fix-up is required, so this is the final state for this step.



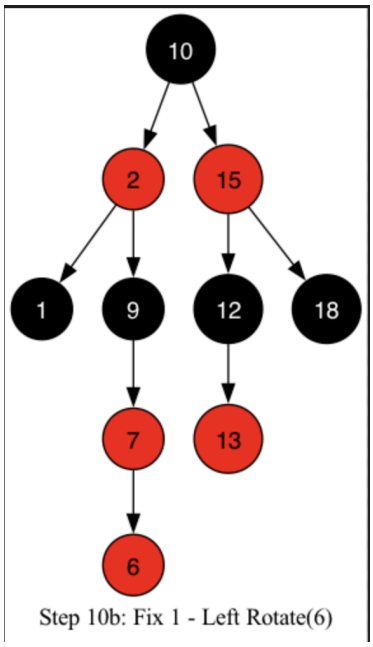
Step 9: Insert 6

Inserting 7,



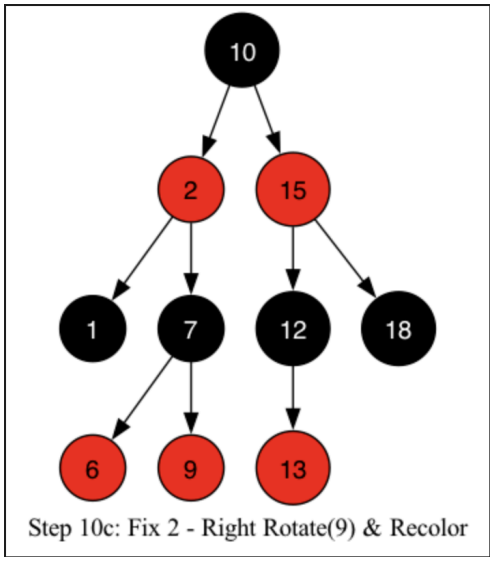
Step 10a: Insert 7 (Left-Right Violation)

First, we perform a Left Rotation on the parent node (6). This resolves the "zig-zag" and transforms the problem into a simpler "Left-Left" case.



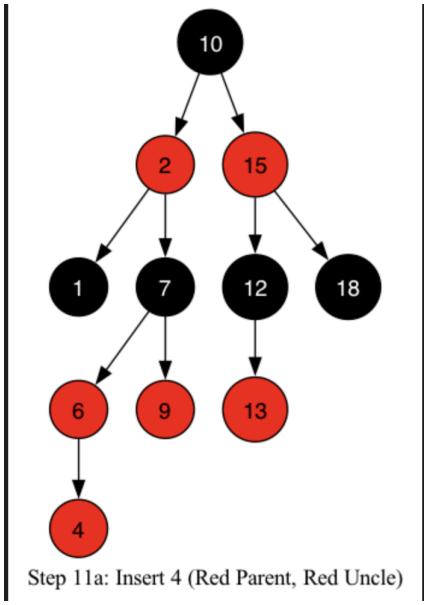
Step 10b: Fix 1 - Left Rotate(6)

Now we solve the "Left-Left" case with a Right Rotation on the grandparent (9). We then recolor: the new root of this subtree (7) becomes Black, and its children (6 and 9) become Red.



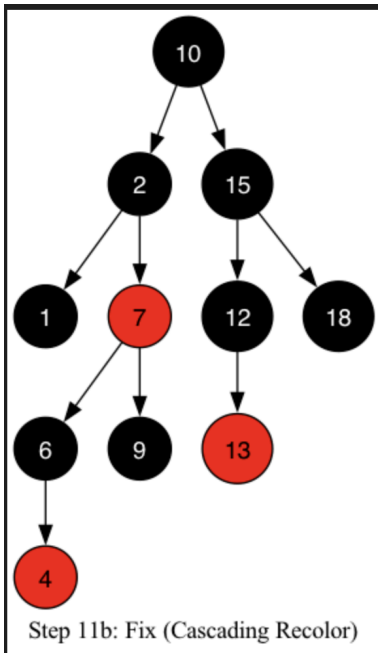
Step 10c: Fix 2 - Right Rotate(9) & Recolor

Insert 4,
The value 4 is inserted as the Red left child of 6. This creates a "double red" violation with its parent, 6(R). The uncle node (9) is also Red, which is Insertion Case 1.



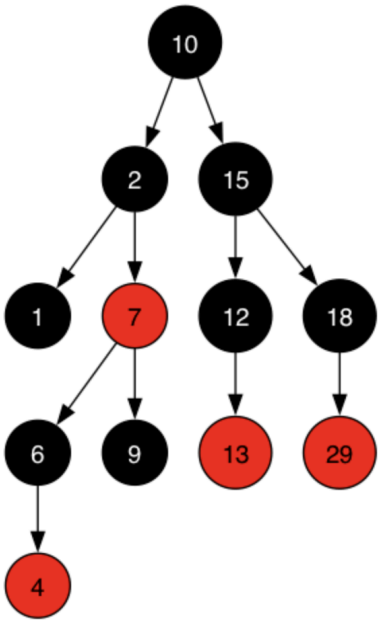
Step 11a: Insert 4 (Red Parent, Red Uncle)

First, we fix the initial violation by recoloring the parent 6 and uncle 9 to Black, and the grandparent 7 to Red.
This creates a new violation, as 7(R) now has a Red parent, 2(R). We look at the new uncle, 15, which is also Red.
We apply the same fix again: recolor parent 2 and uncle 15 to Black. We recolor their parent, the root 10, to Red, but since the root must be Black, it remains Black.



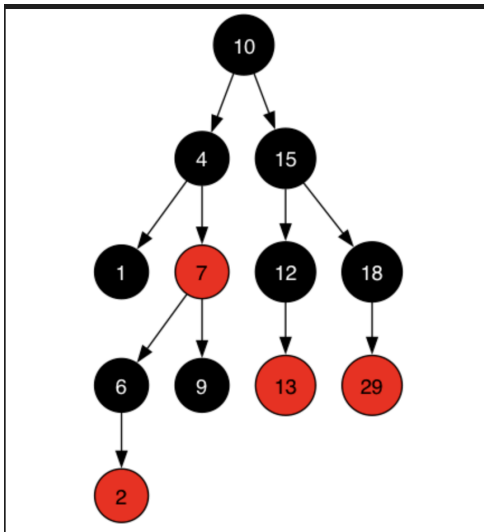
Step 11b: Fix (Cascading Recolor)

Insert 29,
The value 29 is inserted as the Red right child of 18. Since the new Red node (29) has a Black parent (18), no Red-Black Tree rules are violated. No fix-up is required. This is the final and correct state of the tree after all insertions are complete.



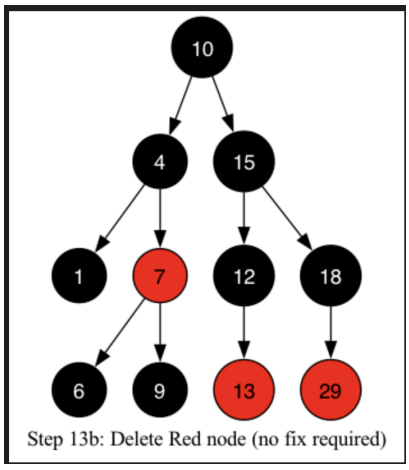
Step 12: Insert 29 (Final Tree)

Deleting 2,
Node 2 is an internal node. Its in-order successor (the next-largest value) is 4. We swap their values. The node we will now physically delete is the leaf node containing the value 2, which is a Red node.



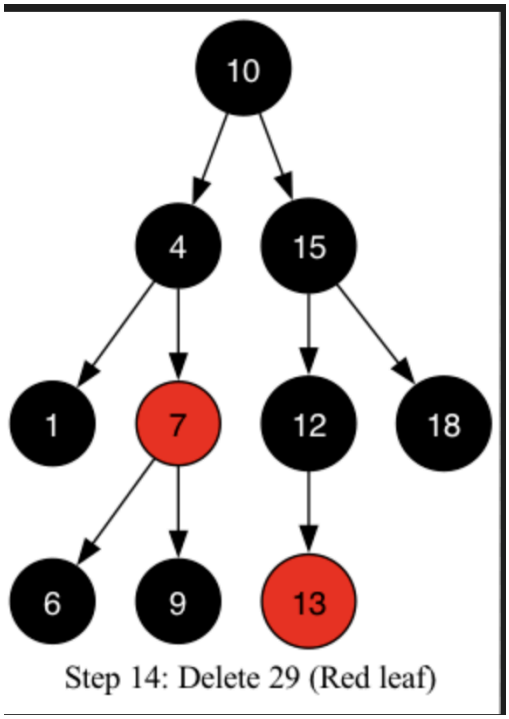
Step 13a: Delete 2 - Find successor(4) and swap values

Now we delete the leaf node containing 2. Since this node is Red, deleting it does not violate any Red-Black Tree properties (specifically, it doesn't change the black-height of any path). This is a trivial case requiring no rotations or recoloring.



Step 13b: Delete Red node (no fix required)

Deleting 29,

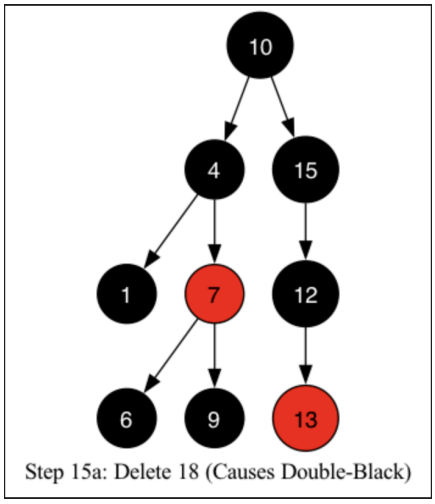


Step 14: Delete 29 (Red leaf)

Deleting 18,

The node to delete, 18, is a Black leaf node. Deleting a Black node violates the black-height rule, creating a "double-black" problem where the node used to be. To fix this, we analyze its sibling (12).

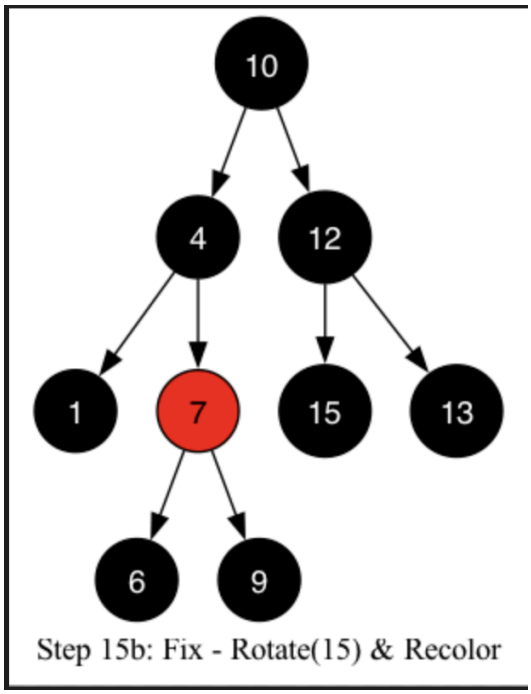
The sibling 12 is Black, and it has a "far" Red child (13). This is Deletion Case 4.



Step 15a: Delete 18 (Causes Double-Black)

The fix for this case involves a rotation and recoloring:

1. Perform a Left Rotation on the parent (15).
2. The new root of the subtree (12) takes the color of the old parent (15), so it remains Black.
3. The old parent (15) and the sibling (12) swap colors, so 15 also remains Black.
4. The Red nephew (13) is colored Black.

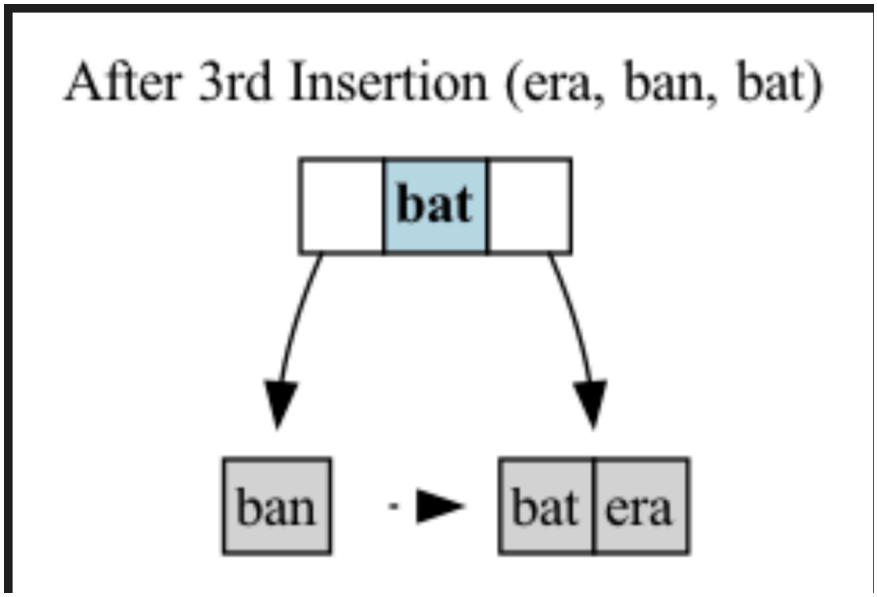


Step 15b: Fix - Rotate(15) & Recolor

Task 6:

After 3rd insertion,

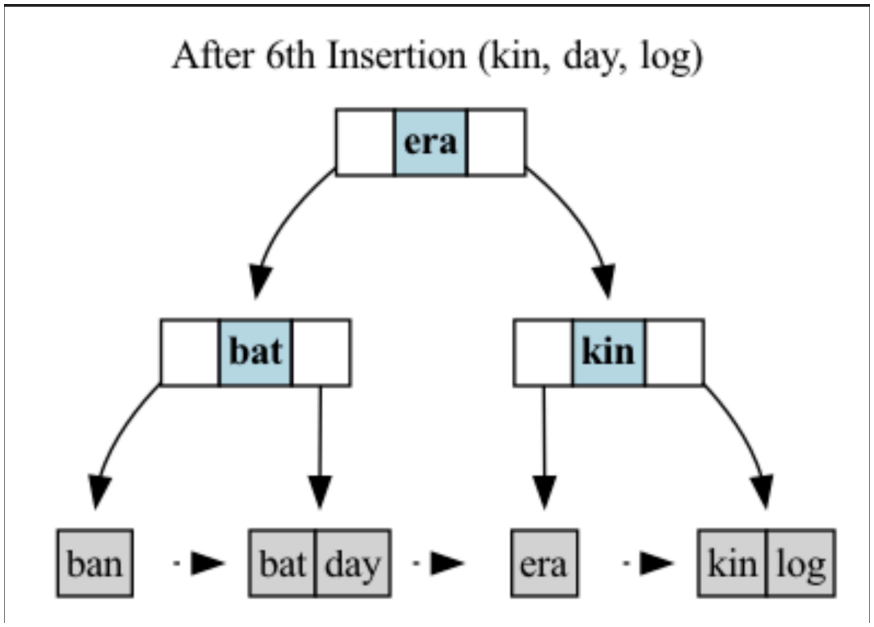
After inserting era, ban, and bat, the initial leaf node becomes overfull ([ban, bat, era]) and must split. The middle key, bat, is promoted to a new root node, which points to the two new leaf nodes.



After 6th insertion.

This sequence involves two leaf splits and one internal node split:

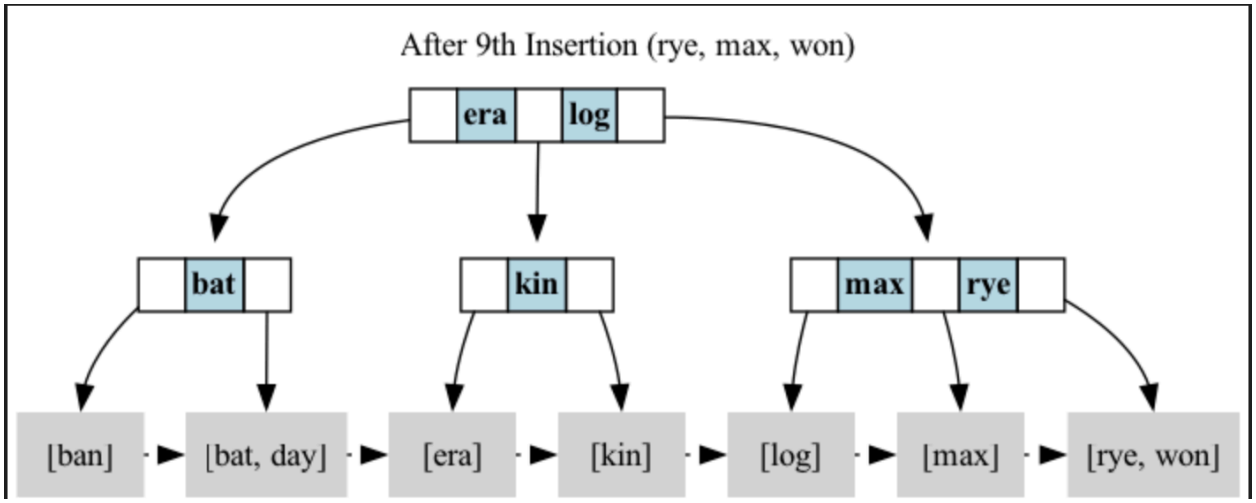
1. Inserting kin causes a leaf to split, promoting era. The root becomes [bat, era].
2. Inserting day fills a leaf node without a split.
3. Inserting log causes another leaf to split, promoting kin. The root becomes [bat, era, kin], which is now overfull.
4. Root Split: The root itself splits. The middle key, era, is promoted to become the new root. The tree now has 3 levels.



After 9th insertion,

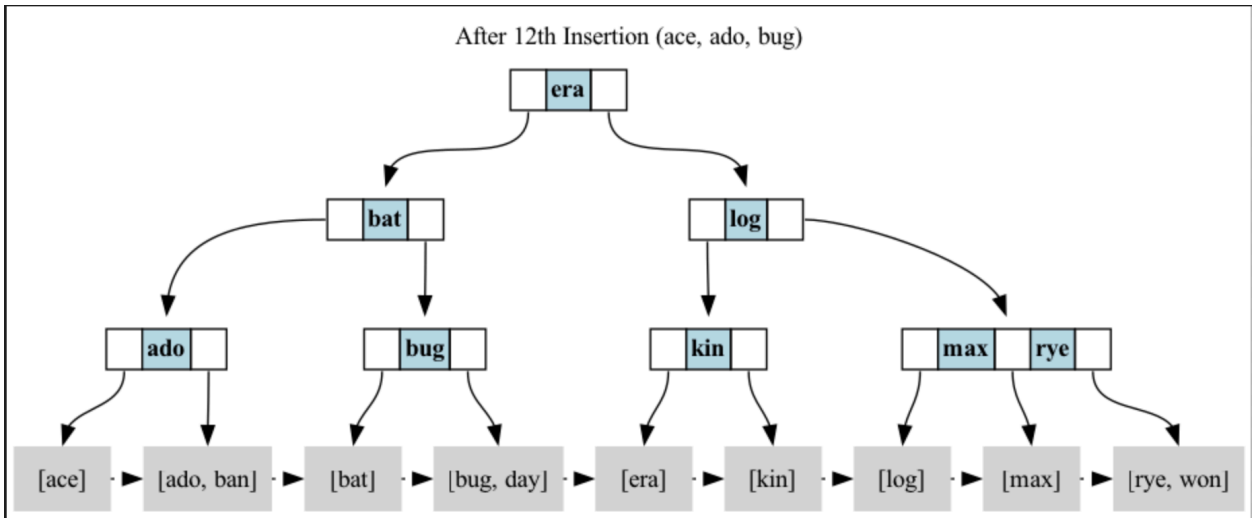
Description: This sequence involves multiple leaf splits and another internal node split.

1. Inserting rye and max each cause a leaf to split. The split from inserting max causes an internal node [kin, log] to become overfull [kin, log, max].
2. Internal Node Split: This node splits, promoting log up to the root. The root absorbs the new key, becoming [era, log].
3. Inserting won causes one final leaf split, which is absorbed by its parent internal node without causing further splits.

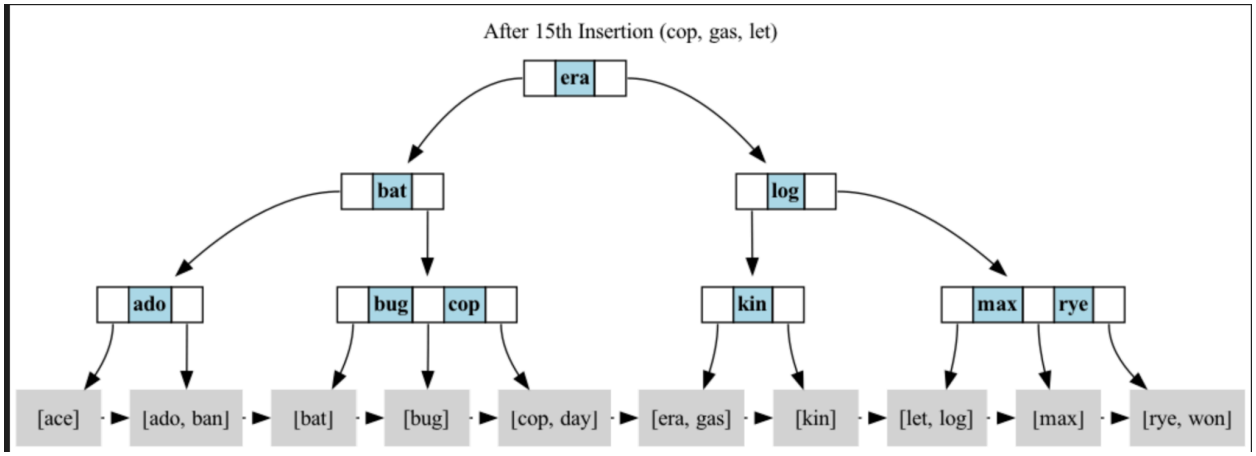


After 12th insertion,

This is a major step. The insertion of bug causes a leaf split, which promotes bug to its parent internal node. This makes the internal node overfull, causing it to split and promote bat to the root. The root, in turn, becomes overfull ([bat, era, log]) and must also split. The key era is promoted to become a new root, and the tree grows to 4 levels.

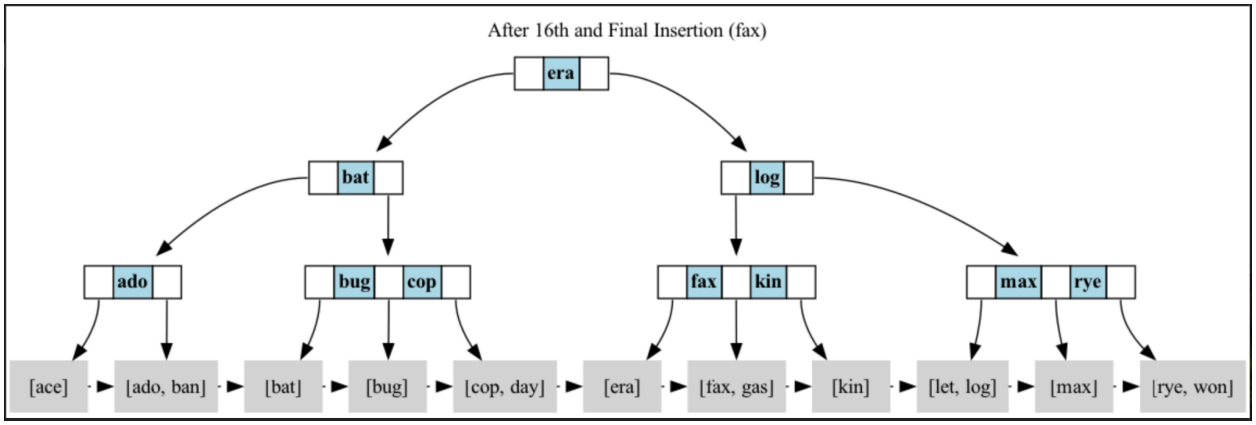


After 15th insertion,
The insertion of cop causes a leaf split, promoting cop to the parent internal node [bug], which becomes [bug, cop]. The other two insertions, gas and let, simply add keys to existing leaf nodes without causing splits. The tree remains 4 levels high.



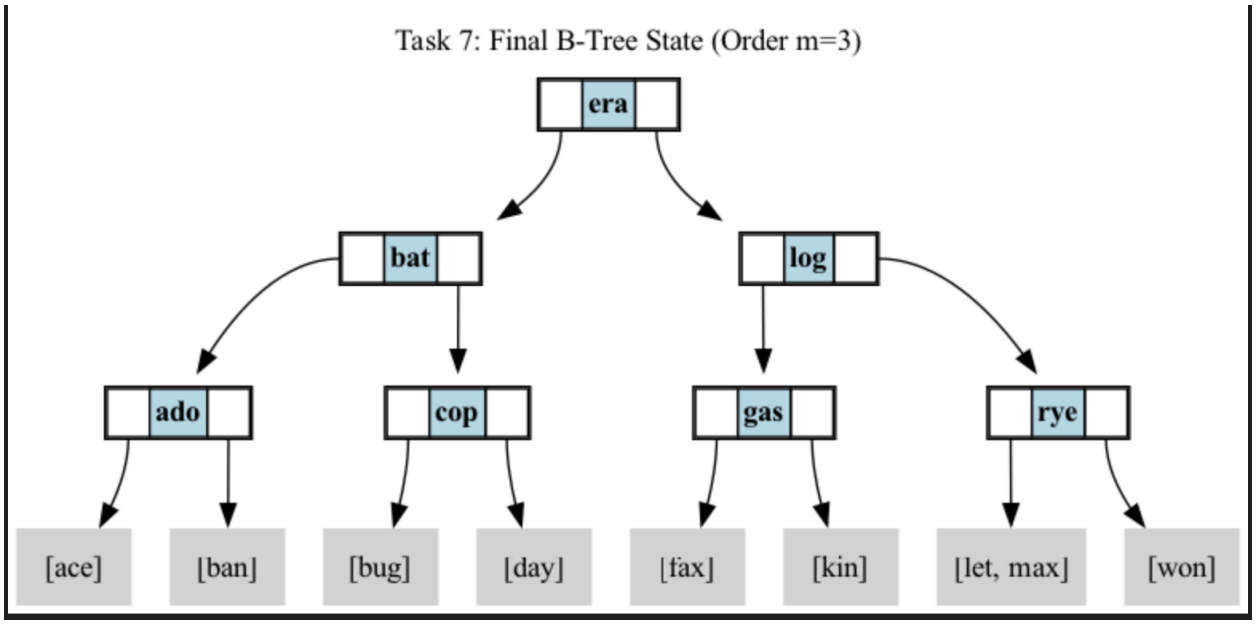
Last state,

The final key, fax, is inserted into the leaf node [era, gas], causing it to become overfull ([era, fax, gas]). The leaf splits, and the middle key, fax, is promoted to the parent internal node [kin], which becomes [fax, kin]. This does not cause any further splits.



Task 7

After performing all 16 insertions (era, ban, bat...fax) according to standard B-Tree rules for an order m=3 tree, multiple node splits and promotions occur. Keys are promoted up the tree as nodes overflow, with some splits cascading up to the root and causing the tree to grow in height. The final result is a 3-level B-Tree.



Task 8

Node Count Comparison

In the specific example we worked through, both the B-Tree and the B+ Tree finished with the same number of internal nodes (7).

However, as a general rule, a B+ Tree structure results in fewer internal nodes. This is because its internal nodes only store simple routing keys, not full data records. This allows more keys to fit per node (a higher "fan-out"), which makes the tree flatter.

Impact on Databases

This structural difference is critical for performance in large-scale databases.

1. Faster Lookups: A flatter tree requires fewer steps to find any piece of data. Since each step can mean reading from a disk, a B+ Tree's shorter height leads to significantly faster data retrieval.
2. Efficient Range Queries: In a B+ Tree, all data is stored at the bottom leaf level, and these leaves are linked together like a sorted list. This makes range queries (e.g., finding all records between 'A' and 'C') extremely fast, as the database can just scan along the leaf level instead of jumping around the tree.

Because of these two major performance advantages, B+ Trees are the standard choice for database indexing.

Task 9

Pseudocode:

```
FUNCTION range_query(root, start_key, end_key):
  // Phase 1: Find the starting leaf node

  current_node = root
  WHILE current_node is not a leaf:
    // Find the correct child pointer to follow
    found_path = false
    FOR each key 'k' in current_node:
      IF start_key < k:
        current_node = pointer to the left of 'k'
        found_path = true
        BREAK

    IF not found_path:
      current_node = the rightmost pointer of current_node

  // current_node is now the first leaf to check

  // Phase 2: Scan leaf nodes and collect results

  results = new empty list
  done = false

  WHILE current_node is not null AND not done:
    FOR each key 'k' in current_node:
      IF k >= start_key AND k <= end_key:
        add 'k' to results list

      IF k > end_key:
        done = true // We've passed the end of our range
        BREAK

    // Move to the next leaf in the sequence
    current_node = current_node.next_leaf_pointer

  RETURN results
```

2. The Role of Leaf-Level Pointers

The leaf-level pointers are the single most important feature that makes range queries in a B+ Tree highly efficient. Here’s how they help:

1. Enabling Sequential Scans: After the initial search descends the tree to find the first relevant leaf node, the algorithm does not need to traverse back up to the internal nodes. Instead, it can simply follow the next_leaf_pointer to the adjacent leaf.
2. Minimizing Disk I/O: This turns the process into a simple, fast, horizontal scan across the bottom level of the tree. In a real database, this is crucial because it means reading sequential data blocks from the disk, which is dramatically faster than the "random access" of jumping up and down different levels of the tree (which would be required in a standard B-Tree).

In short, the leaf-level pointers transform a potentially complex tree traversal into a simple linked-list traversal, which is the key to the B+ Tree's superior performance for range queries.