Binary Search Trees and Red–Black Trees
Lecture Notes

Prepared for students

September 6, 2025

Abstract

These notes introduce Binary Search Tree (BST) variants (AVL, Splay, Treap, Red–Black, B-trees) with a focused, detailed treatment of **Red–Black Trees (RBT)**: invariants, rotations (left/right), insertion and deletion algorithms (with pseudocode), worked examples and complexity analysis (worst-case and amortized remarks).

Contents

1 Overview: Binary Search Tree Variants

A *binary search tree (BST)* stores keys in nodes so that for any node x, all keys in the left subtree are $< x$ and all keys in the right subtree are $> x$. BSTs support search, insert, delete; the running time depends on tree height h. A balanced BST keeps $h = O(\log n)$ so operations are $O(\log n)$.

Common BST variants (brief):

- **AVL tree (Adelson-Velskii Landis)**: strictly height-balanced; for every node heights of children differ by at most 1. Guarantees height $h \leq 1.44 \log n$. Insert/delete require at most $O(\log n)$ rotations (often constant).
- **Red–Black Tree (RBT)**: relaxed balance via node colors (red/black) and invariants. Guarantees height $h \leq 2 \log(n + 1)$. Very popular in libraries (e.g., Linux kernel, Java TreeMap).
- **Splay tree**: self-adjusting; every access "splays" the accessed node to root. Amortized $O(\log n)$ per operation (not worst-case).
- **Treap**: randomized BST mixing BST order with heap keys (random priorities). Expected height $O(\log n)$.
- **(B-)Trees and B+-Trees**: multi-way balanced search trees used in databases and filesystems to reduce disk I/O.

We now focus on Red–Black Trees.

2 Red–Black Trees: definition and invariants

2.1 RBT node and NIL sentinel

A Red–Black tree is a BST in which each node has a `color` either `red` or `black`. We also conceptually use NIL leaves (often implemented as a single sentinel node) which are `black`.

Typical node structure:

$$\text{Node } x : \{\text{key}, \text{left}, \text{right}, \text{parent}, \text{color}\}.$$

2.2 Red–Black invariants (properties)

Every RBT satisfies:

1. Each node is either red or black.
2. The root is black.
3. Every NIL leaf is black.
4. If a node is red, then both its children are black. (No two consecutive reds.)
5. For each node, all simple paths from the node to descendant NIL nodes contain the same number of black nodes. (This number is the node's *black-height*.)

2.3 Consequences (height bound)

Let n be the number of internal nodes. Using invariants one can show

$$\text{height } h \le 2\log_2(n+1).$$

Sketch: every path from root to leaf has at least $\lfloor h/2 \rfloor$ black nodes (because no two reds appear consecutively), so $2^{\lfloor h/2 \rfloor} - 1 \le n$. Rearranging yields $h \le 2\log_2(n+1)$.

Thus search, insert, delete are $O(h) = O(\log n)$ worst-case.

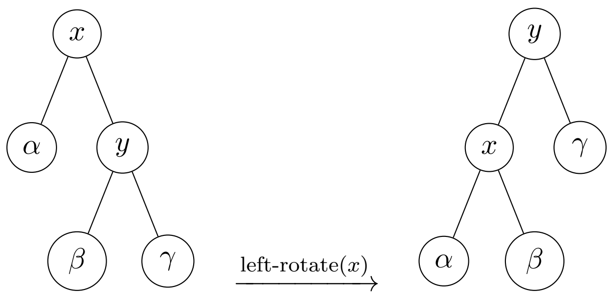
3 Rotations (left and right) – core primitive

Rotations are local tree operations that preserve the BST in-order sequence but change the shape. They run in constant time.

3.1 Left rotation

Given a node x with right child y, left-rotate around x:

$$\begin{aligned}
&y = x.\text{right},\\
&x.\text{right} \leftarrow y.\text{left},\\
&\text{if } y.\text{left} \neq \text{NIL then } y.\text{left.parent} \leftarrow x,\\
&y.\text{parent} \leftarrow x.\text{parent},\\
&\text{(adjust parent's child pointer to } y\text{)},\\
&y.\text{left} \leftarrow x,\\
&x.\text{parent} \leftarrow y.
\end{aligned}$$



3.2 Right rotation

Symmetric: given y with left child x, right-rotate around y. It is the inverse of a left rotation.

Properties: rotations preserve in-order traversal and take $O(1)$ time.

4 Red–Black Tree insertion

4.1 High-level idea

Insert like a BST (insert new node z as a red leaf), then fix any violations of RBT invariants by recoloring and/or rotations. The fix-up walks up the tree and uses local transformations to restore the properties.

4.2 Pseudocode (CLRS-style)

Algorithm 1 RB-INSERT(T, z)

Require: Red–Black tree T and new node z (with key) to insert

```
Insert z as in a BST (place as leaf); set z.left = z.right = NIL
Set z.color ← RED
Call RB-INSERT-FIXUP(T, z)
```

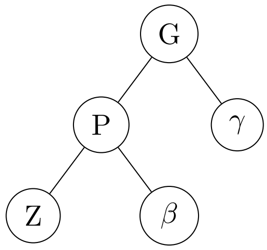
Algorithm 2 RB-INSERT-FIXUP(T, z)

```
while z.parent.color = RED do
  if z.parent = z.parent.parent.left then
    y ← z.parent.parent.right {uncle}
    if y.color = RED then
      {Case 1: recolor}
      z.parent.color ← BLACK
      y.color ← BLACK
      z.parent.parent.color ← RED
      z ← z.parent.parent
    else
      if z = z.parent.right then
        {Case 2: left-rotate parent}
        z ← z.parent
        LEFT-ROTATE(T, z)
      end if
      z.parent.color ← BLACK {Case 3: right-rotate grandparent}
      z.parent.parent.color ← RED
      RIGHT-ROTATE(T, z.parent.parent)
    end if
  else
    {symmetric: parent is right child}
    (mirror image of above with “left” and “right” exchanged)
  end if
end while
T.root.color ← BLACK
```

4.3 Insertion cases (intuition)

When the parent is red, the violation is a red parent with red child. Let uncle be sibling of the parent.

- **Case 1 (uncle is red):** recolor parent and uncle black, grandparent red, move z up to grandparent and continue.
- **Case 2 (uncle is black, z is “inner” child):** rotate parent (left or right) to transform into case 3.
- **Case 3 (uncle is black, z is “outer” child):** recolor parent black, grandparent red, rotate grandparent (right or left).



4.4 Number of rotations and time complexity

- **Time:** RB-INSERT runs in $O(h) = O(\log n)$ time worst-case (search to insertion point + fix-up).
- **Rotations:** RB-INSERT-FIXUP performs at most **2** rotations per insertion (this is a standard CLRS fact). Recolorings may occur up the tree, but rotations are very limited.
- **Amortized remark:** over a sequence of insertions the amortized number of rotations per insertion is $O(1)$ (indeed bounded by a small constant).

5 Red–Black Tree deletion

5.1 High-level idea

Deletion is more intricate. Remove node z following BST-delete semantics (if node has two children, swap with successor and remove) and track whether removal decreases the black-height of paths. If properties are violated, perform a sequence of recolorings and rotations in **RB-DELETE-FIXUP** to restore RBT invariants.

5.2 Pseudocode (CLRS-style)

Algorithm 3 RB-DELETE(T, z)

Require: Red–Black tree T and node z to delete

Standard BST deletion: remove z; let y be the node actually removed or the node moved (as in transplant)

If the removed node's color was BLACK, call **RB-DELETE-FIXUP**(T, x) where x is the node that replaced y (possibly NIL).

Algorithm 4 RB-DELETE-FIXUP(T, x)

```
while x ≠ T.root and x.color = BLACK do
  if x = x.parent.left then
    w ← x.parent.right {sibling}
    if w.color = RED then
      w.color ← BLACK
      x.parent.color ← RED
      LEFT-ROTATE(T, x.parent)
      w ← x.parent.right
    end if
    if w.left.color = BLACK and w.right.color = BLACK then
      w.color ← RED
      x ← x.parent
    else
      if w.right.color = BLACK then
        w.left.color ← BLACK
        w.color ← RED
        RIGHT-ROTATE(T, w)
        w ← x.parent.right
      end if
      w.color ← x.parent.color
      x.parent.color ← BLACK
      w.right.color ← BLACK
      LEFT-ROTATE(T, x.parent)
      x ← T.root
    end if
  else
    (mirror image with "left" and "right" exchanged)
  end if
end while
x.color ← BLACK
```

5.3 Deletion cases intuition

In RB-DELETE-FIXUP, x is a node that may have an extra *black deficit* (often called "double-black") which must be fixed. The algorithm examines x's sibling w and considers:

- **Case 1 (sibling red):** recolor and rotate to transform to a case where sibling is black.
- **Case 2 (sibling black, both sibling's children black):** recolor sibling red and move the problem up to the parent.
- **Case 3 (sibling black, sibling's outer child black, inner child red):** rotate sibling to convert to case 4.
- **Case 4 (sibling black, sibling's outer child red):** recolor and rotate to fix deficit and terminate.

5.4 Number of rotations and time complexity

- **Time:** RB-DELETE (including fixup) takes $O(h) = O(\log n)$ worst-case time.

- **Rotations:** RB-DELETE-FIXUP may perform multiple iterations and may do up to $O(h)$ rotations in the worst case. So deletion may invoke $O(\log n)$ rotations in pathological cases.

- **Amortized remark:** despite possible multiple rotations for a single deletion, per-operation amortized cost (time) is $O(\log n)$. In practice, rotations remain efficient; overall performance is logarithmic.

6 Worked example: insert sequence in a Red–Black Tree

We give a short, step-by-step example of inserting keys: 10, 20, 30 into an initially empty RBT.

1. Insert 10: becomes root; initially colored RED but then fixed to BLACK (root must be black).

2. Insert 20: inserted as red right child of 10; parent is black so no violations.

3. Insert 30: inserted as red right child of 20. Now parent (20) is red $\rightarrow$ violation (red parent and red child).

    - Uncle of 30 is NIL (black). This is the "uncle black" case (Case 2/3).
    - Because 30 is a right child of 20 and 20 is right child of 10 (outer case), recolor 20 black, 10 red and perform left-rotate at 10.
    - Resulting tree root becomes 20 (colored black), with left child 10 (black) and right child 30 (red).

A diagram before and after rotation clarifies the change.

7 Complexity summary and practical notes

- **Search:** $O(h) = O(\log n)$ worst-case.

- **Insert:** $O(h) = O(\log n)$ worst-case. At most **2** rotations per insertion (constant).

- **Delete:** $O(h) = O(\log n)$ worst-case. May perform $O(h)$ rotations in worst-case, but overall deletion time is $O(\log n)$.

- **Space:** $O(n)$ for nodes.

- **Amortized viewpoint:** across sequences of operations the per-operation running time remains $O(\log n)$; inserts are especially cheap in terms of rotations (constant number).

- **Why RBTs are popular:** they give provable logarithmic worst-case bounds, have relatively simple fix-up logic (compared to AVL's stricter balancing), and are efficient in practice with small constants. They form the basis of many standard library balanced-map implementations.

8 References and further reading

- T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms.* (CLRS) — chapter on red–black trees.

- Original red–black tree description and many textbooks discuss invariants and proofs of height bound.

- For implementations: study Linux kernel RB-tree macros or Java's TreeMap source (which uses red–black trees).

Notes for instructors: You can extend these notes by adding step-by-step diagrams for each insert/delete case, and by comparing AVL vs RB tradeoffs (AVL is more strictly balanced so slightly faster searches, RBTs have cheaper updates typically).