Binary Search Trees: Red–Black, AVL, Treap and Splay Trees (with Amortized Analysis and Access Bounds)

Prepared for students

September 6, 2025

Abstract

These notes survey common BST variants (AVL, Red–Black, Treap, B-tree) and present a detailed treatment of **Splay Trees**: splaying operation, algorithms (search/insert/delete), amortized analysis using the access-lemma / potential method, and the three central access bounds satisfied by splay trees: *static optimality*, *static finger*, and the *working-set bound*. We include diagrams, pseudocode, worked examples and practical use-cases.

Contents

1 Quick overview of BST variants and their costs

A binary search tree stores keys with the BST invariant: left subtree keys < node key < right subtree keys.

Common balanced BST variants:

- **AVL tree:** Strict height balance. Height $h \le 1.44 \log n$. Operations worst-case $O(\log n)$; rotations per update constant (at most 2 single/double rotations on insert).
- **Red–Black tree (RBT):** Relaxed balance via colors. Height $h \le 2 \log(n+1)$. Search/insert/delete worst-case $O(\log n)$. Popular in libraries.
- **Treap:** Randomized BST that stores a random priority per key and maintains heap order on priorities. Expected height $O(\log n)$; expected $O(\log n)$ per operation.
- **B-Tree / B$^+$-Tree:** Multiway balanced trees used for disks/DBs; operations $O(\log_b n)$ block accesses where b = branching factor.
- **Splay tree:** Self-adjusting BST that performs a *splay* (series of rotations) bringing the accessed node to the root. *Worst-case* cost of a single operation can be $O(n)$ but the *amortized* cost of access sequences is $O(\log n)$ and in fact satisfies stronger bounds (static optimality, static finger, working-set).

Table (summary):

Structure	Cost per operation	Notes
AVL	$O(\log n)$ worst-case	Very balanced; slightly more rotations on updates
Red–Black	$O(\log n)$ worst-case	Simpler update logic, widely used
Treap	$O(\log n)$ expected	Simple randomized, often practical
B-Tree	$O(\log_b n)$ I/Os	For external memory
Splay tree	$O(\log n)$ amortized	Strong access bounds; simple code; dynamic-optimality conjecture

We now focus on Splay Trees.

2 Splay trees: what and why

Splay trees (Sleator & Tarjan, 1985) are binary search trees that self-adjust: after every access (search, insert, delete) the accessed node is moved to the root by a sequence of rotations called *splaying*. The guiding idea is to adapt the tree to the access pattern: frequently accessed nodes become near the root and future operations are faster.

Pros:

- Very simple to implement (no stored balance info).
- Excellent amortized performance and adaptability: many locality properties (working-set etc).
- Often competitive in practice for workloads with locality or skewed access frequency.

Cons:

- Single-operation worst-case $O(n)$ (but amortized bounds remain good).
- Deletes are somewhat more intricate than in balanced BSTs.

3 Splaying: rotations and cases

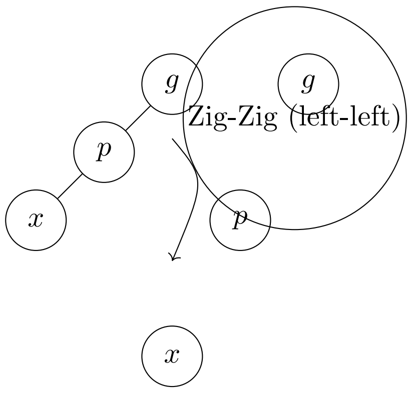
Splaying a node x moves it to the root by repeatedly performing one of three rotation patterns depending on the relationship of x with its parent (p) and grandparent (g).

Notation: $p = \text{parent}(x)$, $g = \text{parent}(p)$.

Cases:

- **Zig:** p is the root (no grandparent). Perform a single rotation (left or right) between p and x.
- **Zig-Zig:** x and p are both left children (or both right children) of their parents. Rotate p about g, then rotate x about p (two rotations in same direction).
- **Zig-Zag:** x is a left child and p is a right child (or vice-versa). Rotate x about p, then rotate x about g (two rotations in opposite directions).

These rotations preserve the BST order and progressively move x upward until it becomes root.



(You can find canonical zig/zig-zig/zig-zag diagrams in many references. The code above will draw simpler local diagrams — the important point is the pattern of rotations.)

4 Splay: algorithms (pseudocode)

We assume standard BST node structure with `left`, `right`, `parent`.

Algorithm 1 SPLAY(T, x)

```
while $x$ is not root of $T$ do
    $p \leftarrow x$.parent
    $g \leftarrow p$.parent
    if $g$ is null then
        {Zig}
        if $x$ is left child of $p$ then
            RIGHT-ROTATE($T, p$)
        else
            LEFT-ROTATE($T, p$)
        end if
    else if $x$ is left child of $p$ and $p$ is left child of $g$ then
        {Zig-Zig}
        RIGHT-ROTATE($T, g$)
        RIGHT-ROTATE($T, p$)
    else if $x$ is right child of $p$ and $p$ is right child of $g$ then
        {Zig-Zig symmetric}
        LEFT-ROTATE($T, g$)
        LEFT-ROTATE($T, p$)
    else
        {Zig-Zag}
        if $x$ is right child of $p$ and $p$ is left child of $g$ then
            LEFT-ROTATE($T, p$)
            RIGHT-ROTATE($T, g$)
        else
            RIGHT-ROTATE($T, p$)
            LEFT-ROTATE($T, g$)
        end if
    end if
end while
```

Algorithm 2 SEARCH(T, k) using splay

```
$x \leftarrow$ standard BST search for key $k$ (last accessed node on search path)
if $x$ is non-null and $x.key = k$ then
    SPLAY($T, x$)

    return $x$ {found, now at root}
else
    SPLAY($T, x$) {splay last accessed node (helps locality)}

    return not-found
end if
```

Algorithm 3 INSERT(T, k) using splay

```
If tree empty, create root node with $k$ and return
BST-insert new node $z$ as usual (leaf); SPLAY($T, z$)
```

Algorithm 4 DELETE(T, k) using splay

```
1: SEARCH($T, k$)
2: if found node $x$ at root then
3:     If $x$.left is NIL return $x$.right as new tree root
4:     Else let $L = x$.left; detach $L$ (set parent null); let $R = x$.right
5:     Find maximum node $m$ in $L$; SPLAY($T, m$) (so $m$ becomes root of $L$ and $m$.right is NIL)

6:     Set $m$.right $= R$ and update parents; $m$ is new root
7: end if
```

5 Amortized analysis: potential method and Access Lemma

Splay trees' power comes from amortized bounds proven using a clever potential function based on subtree sizes (or ranks).

5.1 Definitions

Let each node x of the current tree have size $s(x)$ = number of nodes in subtree rooted at x (including x). Define the *rank* of a node

$$r(x) = \log_2 s(x).$$

(Any base > 1 works; base 2 is convenient.) Define the global potential

$$\Phi(T) = \sum_{v \in T} r(v) = \sum_{v} \log s(v).$$

When we change the tree by rotations, Φ changes; amortized cost of an operation = actual cost + change in potential.

5.2 Access Lemma (informal statement)

Let T be a splay tree with root t and let x be a node we splay to the root. Let $s(\cdot)$ and $r(\cdot)$ be before-splay values (or analyze carefully with before/after). Then the amortized cost of splaying x (i.e. number of basic rotations plus potential change) is bounded by

$$\text{amortized-cost} \leq C\big(r(t) - r(x)\big) + O(1)$$

for some constant C (classically $C = 3$ suffices in the standard proof). Because $r(t) = \log n$ and $r(x) \geq 0$, splaying costs amortized $O(\log n)$.

Sketch of the proof idea: One analyzes the cost of each rotation (zig, zig-zig, zig-zag) and shows that the rotation decreases the sum of ranks in such a way that the actual rotation cost is charged against the drop in potential. Summing across the sequence gives the inequality above.

5.3 Amortized single-operation bound

From the access lemma it follows that splaying any node x costs amortized $O(\log n)$. In particular, for a sequence of m operations starting on an n-node tree, total time is $O(m \log n + n)$ (the $+n$ covers building potential if needed).

5.4 Remarks about constants

The classical Sleator–Tarjan proof shows amortized cost per access $\leq 3(\log n + 1)$ (in a more precise accounting). For teaching purposes, conveying the $O(\log n)$ amortized bound and the access-lemma proof sketch is sufficient.

6 Stronger bounds: Static optimality, Static finger, Working-set

Splay trees satisfy a collection of access bounds that make them adapt well to non-uniform and temporal locality of accesses.

We state the three classic bounds (theorems are stated informally, with intuitive proofs/sketches).

6.1 Static optimality theorem

Theorem (Static Optimality). Let S be an access sequence of length m over keys from a set of size n. Let q_x be the number of accesses to key x in S. Let $\mathrm{OPT}_{\mathrm{stat}}(S)$ denote the cost of an optimum static BST (one fixed tree chosen ahead of time) that minimizes total access cost for the sequence (i.e., put frequently accessed keys near root). Then the total time for splay tree to serve S is

$$\mathrm{Cost}_{\mathrm{splay}}(S) = O\Big(\sum_x q_x \log \frac{m}{q_x} + n\Big),$$

which is within a constant factor of the best static BST for the sequence.

Intuition / Sketch: - A static optimal tree places items with high frequency near the root; the information-theoretic lower bound for accessing x q_x times is $\sum_x q_x \log(m/q_x)$. - Using the access lemma with a suitable potential based on access frequencies one can massage splay's amortized cost into the above bound — splay does nearly as well as the optimal static tree without knowing the frequencies.

6.2 Static-finger theorem

Theorem (Static Finger). Fix a key f called the finger. For any access sequence S of length m, the total time splay needs satisfies

$$\mathrm{Cost}_{\mathrm{splay}}(S) = O\Big(\sum_{i=1}^{m} \log\big(1 + |\mathrm{rank}(a_i) - \mathrm{rank}(f)|\big)\Big),$$

where $\mathrm{rank}(\cdot)$ is an order-rank function (in-order index). In words: the time to access a key is logarithmic in its distance (rank difference) from the fixed finger.

Use / Intuition: If many accesses cluster around a fixed element (e.g., working near a cursor or pointer), splay trees serve them efficiently even though tree is not reorganized specifically for that finger.

6.3 Working-set theorem

Theorem (Working-Set Bound). For each access a_i at time i, define $w_i(a_i)$ to be the number of distinct items accessed since the previous access to a_i (or since the beginning if it hasn't been accessed yet). Then splay trees serve the sequence in total time

$$\mathrm{Cost}_{\mathrm{splay}}(S) = O\Big(\sum_{i=1}^{m} \log(1 + w_i(a_i))\Big).$$

This is also called the working-set bound and captures temporal locality: recently accessed items (small working-set) are cheap to access again.

Sketch / Intuition: Using the access lemma and a potential that depends on access times or recency ranks, Sleator–Tarjan prove this bound. Practically, it implies splay trees behave like an adaptive cache with logarithmic cost in the number of distinct items seen since last use.

6.4 Consequences and relationships

- Static optimality implies good performance when the access frequencies are skewed (hot keys). - Working-set implies locality in time is exploited. - Static finger implies locality in key-space (spatial locality) is exploited. - Combined, these show splay trees are highly adaptive without explicit tuning.

6.5 Dynamic optimality conjecture (brief)

Splay trees are conjectured to be dynamically optimal: for any access sequence S, the splay tree cost is within a constant factor of the cost of the best possible online BST algorithm. This remains open. Several designs (Tango trees, etc.) achieve $O(\log\log n)$ competitive ratio but not constant.

7 Worked examples and real-life use-cases

7.1 Worked numeric example: access sequence with locality

Start with keys $\{1,\dots,7\}$ stored in a splay tree (some initial tree). Access sequence: 5,6,5,6,5,6,... repeated many times. Splaying brings 5 and 6 near root; subsequent accesses cost $O(1)$ amortized per access because working-set for those keys is small and splay reorganizes the tree to keep them near root. This models a "hot pair" in an application (two frequently used variables).

7.2 Real-life example 1: self-adjusting caches

A splay tree can be used to maintain an ordered dictionary where accesses to recently-used keys should be fast. E.g., in an editor that maintains text or symbol tables, the pattern of accesses often has temporal locality (recently edited symbols are accessed again); splay's working-set bound gives theoretical justification for good performance.

7.3 Real-life example 2: IP routing and longest-prefix matches

Ordered dictionaries for prefixes where some prefixes are accessed very frequently can benefit from splay trees because repeated lookups bring hot prefixes near the top. (In practice, highly-tuned prefix structures are used, but the adaptivity idea is the same.)

7.4 Real-life example 3: compression / move-to-front analogy

The move-to-front heuristic for lists corresponds to a degenerate splay-like idea: bring most recently used items to front. Splay trees provide a balanced, BST-based analog supporting ordered operations.

8 Complexity summary (detailed) and practical notes

8.1 Worst-case vs amortized

- AVL, RBT: $O(\log n)$ worst-case per operation.
- Splay tree: $O(n)$ worst-case for a single operation, but amortized $O(\log n)$ per operation and stronger sequence bounds (working-set, static-optimality, static-finger).

8.2 When to use which

- Use AVL/RBT when worst-case guarantees are required (real-time systems).
- Use splay when access patterns exhibit locality or skewness and simple code + adaptivity are desired. Also useful when implementing caches or access-lists with recency properties.
- Treaps are great when you want randomized simplicity and expected guarantees.

9 Implementation tips and pitfalls

- Implement rotations carefully (update parent pointers).
- For splay trees, always splay the last accessed node even on unsuccessful search — this helps preserve working-set/finger properties.
- For delete, the standard trick is to splay the node, remove it, then combine left and right subtrees by splaying the max of left and attaching right.
- Use iterative rotations to avoid stack overflows on skewed trees.

10 References (selected)

- D. D. Sleator and R. E. Tarjan, "Self-adjusting binary search trees", *J. ACM*, 1985.
- Cormen, Leiserson, Rivest, Stein: *Introduction to Algorithms* (CLRS), chapters on balanced BSTs.
- Tarjan lecture notes and various algorithm textbooks for detailed proofs.

If you'd like next:

- I can produce a version of these notes in Beamer slides.
- I can add step-by-step TikZ animations for each of the three splay cases (zig/zig-zig/zig-zag).
- I can also include a tested C++/Python implementation (commented) of splay trees (search/insert/delete) that you can hand to students.