Week 4 — Binary Search Trees & Red-Black Trees

Félix Chavelli felix.chavelli@inria.fr

March 11, 2026 · Semester 2

PSL

Today's Agenda

Motivation & Overview

Binary Search Trees — Definitions (CLRS 12.1)

BST Operations (CLRS 12.2–12.3)

The Height Problem

Red-Black Trees — Properties (CLRS 13.1)

Rotations (CLRS 13.2)

Red-Black Tree Insertion (CLRS 13.3)

Red-Black Tree Deletion — Overview (CLRS 13.4)

In Practice: `std::set`, `std::map` & Python

Summary & What's Next

Part 1

Motivation & Overview

Why Trees?

- Arrays: $\mathcal{O}(1)$ access, but $\mathcal{O}(n)$ insert/delete
- Linked lists: $\mathcal{O}(1)$ insert/delete, but $\mathcal{O}(n)$ search
- Hash tables: $\mathcal{O}(1)$ average, but no ordering
- **Can we get $\mathcal{O}(\log n)$ for everything *and* maintain order?**

Key Idea

Binary search trees combine the advantages of sorted arrays (binary search) and linked lists (dynamic insertions). All operations run in $\mathcal{O}(h)$ where h is the tree height.

	Search	Insert	Order
Array	$\mathcal{O}(n)$	$\mathcal{O}(n)$	✓
Sorted arr.	$\mathcal{O}(\log n)$	$\mathcal{O}(n)$	✓
Linked list	$\mathcal{O}(n)$	$\mathcal{O}(1)$	✗
Hash table	$\mathcal{O}(1)$*	$\mathcal{O}(1)$*	✗
BST	$\mathcal{O}(h)$	$\mathcal{O}(h)$	✓

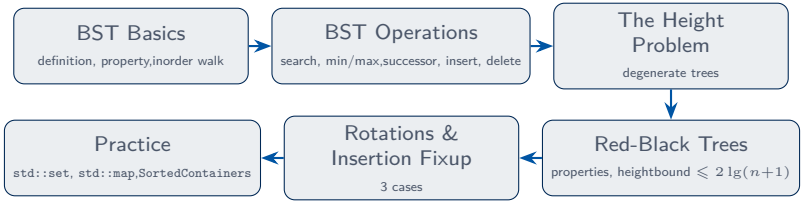
* amortized / average

Trees Everywhere in CS & AI

- **Databases:** B-trees index every SQL table; `std::map` / `std::set` use red-black trees
- **Compilers:** Abstract Syntax Trees (ASTs) are the foundation of code analysis
- **AI — Game trees:** Minimax, Alpha-Beta, MCTS (Monte Carlo Tree Search) power chess engines
- **AI — Decision trees:** Random Forests, Gradient Boosted Trees (XGBoost, LightGBM)
- **File systems:** directory hierarchies are trees
- **Networking:** routing tables, DNS hierarchy

→ Today: the fundamentals — BST operations & balanced trees

Today's Roadmap



Part 2

Binary Search Trees — Definitions (CLRS 12.1)

What is a Binary Search Tree?

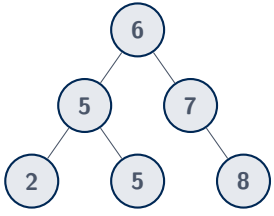
Binary Search Tree (BST)

A **binary search tree** is a binary tree where each node x stores a key (and optional satellite data), and satisfies the **BST property**:

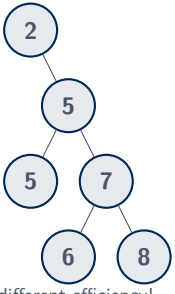
- If y is in the left subtree of x, then $y.key \leqslant x.key$
- If y is in the right subtree of x, then $y.key \geqslant x.key$

Each node has attributes: `key`, `left`, `right`, `p` (parent).

(a) Balanced BST — height 2



(b) Same keys — height 4



Both trees store $\{2, 5, 5, 6, 7, 8\}$. Same keys, different shapes $\Rightarrow$ different efficiency!

Inorder Tree Walk

Inorder Walk

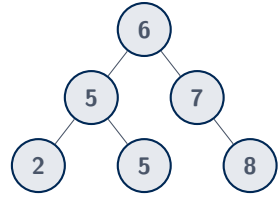
Visit the **left** subtree, then the **root**, then the **right** subtree.
On a BST, this prints all keys in **sorted order**.

Input: Node x (initially $T.root$)

1: **if** $x \neq$ NIL **then**
2: INORDER-WALK($x.left$)
3: **print** $x.key$
4: INORDER-WALK($x.right$)
5: **end if**

Other traversals:

- **Preorder:** root, left, right
- **Postorder:** left, right, root



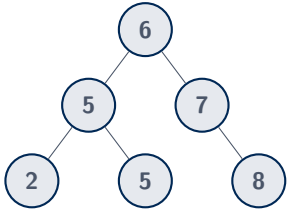
Inorder: $\underset{\uparrow}{2} \rightarrow 5 \rightarrow 5 \rightarrow 6 \rightarrow 7 \rightarrow 8$

✔ Sorted!

Theorem 12.1 (CLRS)

An inorder walk of an n-node BST takes $\Theta(n)$ time.

Preorder & Postorder Walks



Traversal	Order	Output
Inorder	left, root, right	2, 5, 5, 6, 7, 8
Preorder	root, left, right	6, 5, 2, 5, 7, 8
Postorder	left, right, root	2, 5, 5, 8, 7, 6

- **Preorder** → copy/serialize a tree
- **Postorder** → delete/free a tree, expression evaluation
- **Inorder** → sorted sequence from BST

Part 3

BST Operations (CLRS 12.2–12.3)

Search

Recursive version:

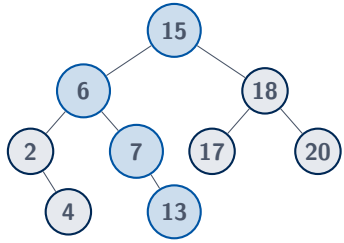
Input: Node x, key k

1: **if** $x = \text{NIL}$ **or** $k = x.key$ **then**
2: **return** x
3: **end if**
4: **if** $k < x.key$ **then**
5: **return** $\text{TREE-SEARCH}(x.left, k)$
6: **else**
7: **return** $\text{TREE-SEARCH}(x.right, k)$
8: **end if**

Iterative version:

1: **while** $x \neq \text{NIL}$ **and** $k \neq x.key$ **do**
2: **if** $k < x.key$ **then**
3: $x \leftarrow x.left$
4: **else**
5: $x \leftarrow x.right$
6: **end if**
7: **end while**
8: **return** x

Searching for key 13:



Path: $15 \to 6 \to 7 \to \mathbf{13}$ ✔

Follows a single root-to-node path.

Running time: $\mathcal{O}(h)$

Minimum & Maximum

Minimum

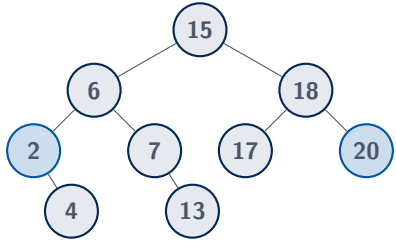
Follow `left` pointers until reaching NIL.

Input: Node x (non-NIL)
1: **while** $x.left \neq$ NIL **do**
2: $\quad x \leftarrow x.left$
3: **end while**
4: **return** x

Maximum

Follow `right` pointers until reaching NIL.

Input: Node x (non-NIL)
1: **while** $x.right \neq$ NIL **do**
2: $\quad x \leftarrow x.right$
3: **end while**
4: **return** x



Min = **2** (leftmost) Max = **20** (rightmost) Both $\mathcal{O}(h)$

Successor & Predecessor

Successor of node x

The node with the **smallest key greater than** $x.key$ in the inorder walk.

Two cases:

1. If x has a right subtree: successor $=$ TREE-MINIMUM$(x.right)$
2. Otherwise: go up until you find an ancestor whose left child is also an ancestor of x

Input: Node x

1: **if** $x.right \neq$ NIL **then**
2: **return** TREE-MINIMUM$(x.right)$ {leftmost in right subtree}
3: **end if**
4: $y \leftarrow x.p$
5: **while** $y \neq$ NIL **and** $x = y.right$ **do**
6: $x \leftarrow y$; $y \leftarrow y.p$
7: **end while**
8: **return** y

Case 1: Successor of 15 $\rightarrow$ min of right subtree $= 17$

Case 2: Successor of 13 $\rightarrow$ go up $\rightarrow 7 \rightarrow 6 \rightarrow 15$

Running time: $\mathcal{O}(h)$ — TREE-PREDECESSOR is symmetric.

Summary of Query Operations

Theorem 12.2 (CLRS)

The operations SEARCH, MINIMUM, MAXIMUM, SUCCESSOR, and PREDECESSOR each run in $\mathcal{O}(h)$ time on a BST of height h.

Operation	Time	Strategy
SEARCH(k)	$\mathcal{O}(h)$	Follow left/right from root
MINIMUM	$\mathcal{O}(h)$	Follow left pointers
MAXIMUM	$\mathcal{O}(h)$	Follow right pointers
SUCCESSOR(x)	$\mathcal{O}(h)$	Right-min or go up
PREDECESSOR(x)	$\mathcal{O}(h)$	Left-max or go up

Key Idea

Everything depends on the **height** h. Best case: $h = \Theta(\log n)$. Worst case: $h = \Theta(n)$.

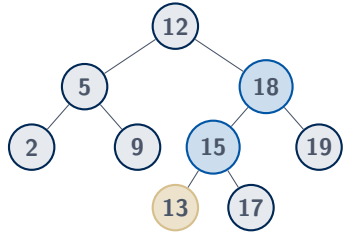
Insertion

Algorithm: descend from root, comparing keys, until finding a NIL position.

Input: Tree T, node z

1: $x \leftarrow T.root; \quad y \leftarrow \text{NIL}$
2: **while** $x \neq \text{NIL}$ **do**
3: $\quad y \leftarrow x$
4: $\quad$ **if** $z.key < x.key$ **then**
5: $\quad\quad x \leftarrow x.left$
6: $\quad$ **else**
7: $\quad\quad x \leftarrow x.right$
8: $\quad$ **end if**
9: **end while**
10: $z.p \leftarrow y$
11: **if** $y = \text{NIL}$ **then**
12: $\quad T.root \leftarrow z$
13: **else if** $z.key < y.key$ **then**
14: $\quad y.left \leftarrow z$
15: **else**
16: $\quad y.right \leftarrow z$
17: **end if**

Inserting key 13:



Path: $12 \rightarrow 18 \rightarrow 15 \rightarrow$ insert at left of 15

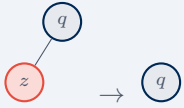
Time: $\mathcal{O}(h)$

Deletion — Overview

Deleting a node z from a BST has **three cases**:

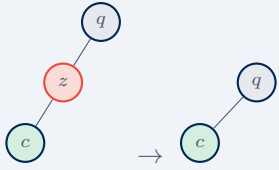
Case 1: No children

Simply remove z.
Replace z by NIL in its parent.



Case 2: One child

Replace z by its only child.
"Elevate" the child.



Case 3: Two children

Find z's successor y
(min of right subtree).
Move y into z's position.

y has no left child
(by Exercise 12.2-5)

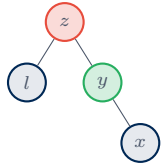
Key Idea

The helper $\textsc{Transplant}(T, u, v)$ replaces the subtree rooted at u with the subtree rooted at v, updating u's parent to point to v.

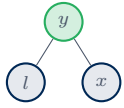
Deletion — Two-Children Case (Detail)

Case 3 has two sub-cases:

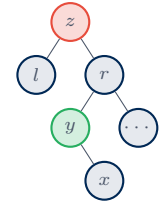
3a) Successor y is z's right child:



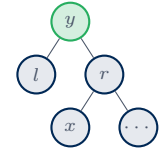
↓ Replace z by y, keep y's right child



3b) Successor y is deeper in right subtree:



↓ First replace y by x, then replace z by y



Theorem 12.3 (CLRS)

INSERT and DELETE each run in $\mathcal{O}(h)$ time on a BST of height h.

Deletion — Pseudocode

Transplant(T, u, v)**:**

```
1:  if u.p = NIL then
2:      T.root ← v
3:  else if u = u.p.left then
4:      u.p.left ← v
5:  else
6:      u.p.right ← v
7:  end if
8:  if v ≠ NIL then
9:      v.p ← u.p
10: end if
```

Tree-Delete(T, z)**:**

```
1:  if z.left = NIL then
2:      TRANSPLANT(T, z, z.right)
3:  else if z.right = NIL then
4:      TRANSPLANT(T, z, z.left)
5:  else
6:      y ← TREE-MINIMUM(z.right)
7:      if y ≠ z.right then
8:          TRANSPLANT(T, y, y.right)
9:          y.right ← z.right
10:         y.right.p ← y
11:     end if
12:     TRANSPLANT(T, z, y)
13:     y.left ← z.left
14:     y.left.p ← y
15: end if
```

Part 4

The Height Problem

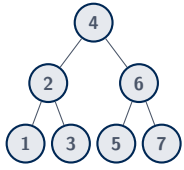
Best Case vs. Worst Case

All BST operations are $\mathcal{O}(h)$ — but what is h?

✓ Best case: balanced tree

Insert keys in a "balanced" order
$h = \lfloor \log_2 n \rfloor = \Theta(\log n)$

All operations $\mathcal{O}(\log n)$

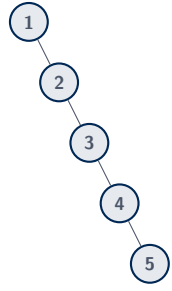


Keys inserted: 4, 2, 6, 1, 3, 5, 7

✗ Worst case: degenerate

Insert keys in **sorted order**
$h = n - 1 = \Theta(n)$

BST degenerates to a linked list!



Keys inserted: 1, 2, 3, 4, 5

Expected Height of a Random BST

Key Idea

If you insert n keys **in random order** (each of the $n!$ permutations equally likely), the expected height is $\mathcal{O}(\log n)$.

(Proof: relates to randomized quicksort — see CLRS Problem 12-3.)

But in practice, data is often not random:

- Sorted or nearly-sorted data (timestamps, auto-increment IDs)
- Data with patterns (alphabetical names, sequential readings)
- After many insertions/deletions, the tree can become unbalanced

→ We need a **guarantee**.

Enter balanced search trees: height always $\mathcal{O}(\log n)$.

Part 5

Red-Black Trees — Properties (CLRS 13.1)

What is a Red-Black Tree?

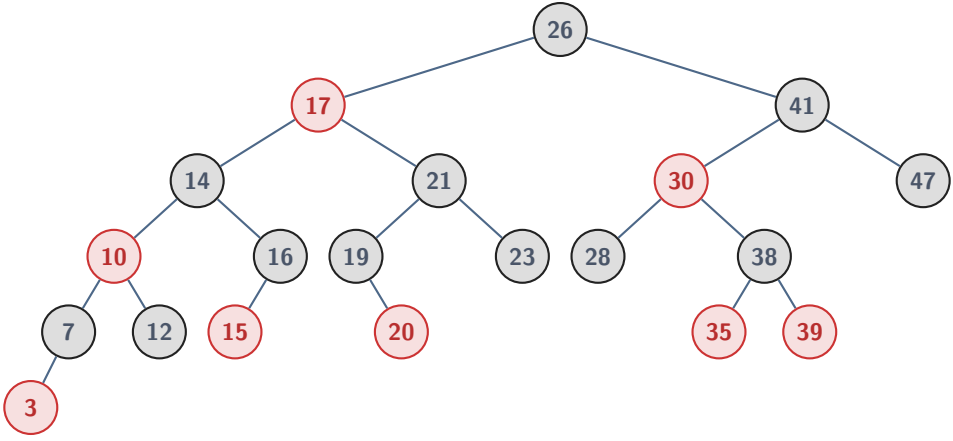
Red-Black Tree

A **red-black tree** is a BST where each node is colored **RED** or **BLACK**, satisfying:

1. Every node is either red or black.
2. The **root** is black.
3. Every **leaf** (NIL) is black.
4. If a node is red, then both its children are **black**.
   (No two consecutive red nodes on any path.)
5. For each node, all simple paths from that node to descendant leaves contain the **same number of black nodes**.

- Property 4 $\Rightarrow$ no path can have two consecutive red nodes
- Property 5 $\Rightarrow$ **black-height** is well-defined: $\mathrm{bh}(x)$ = number of black nodes on any path from x down to a leaf (not counting x)
- In practice, all NIL pointers point to a single **sentinel** node $T.nil$

A Red-Black Tree Example



Red nodes in red, black nodes in dark. Verify the 5 properties!

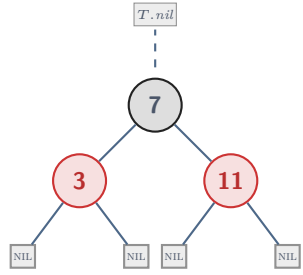
The Sentinel $T.nil$

- All NIL pointers are replaced by a **single sentinel** $T.nil$
- The sentinel is colored **black** (satisfies property 3)
- The root's parent is also $T.nil$
- **Simplifies code**: no special-casing for null pointers
- In drawings, we usually omit the sentinel leaves

Implementation Note

In C++, a single `static` sentinel node saves memory.
In Python, you can use a `NilNode` singleton.

With sentinel shown:



All NILs point to the same sentinel

Height Bound — The Key Guarantee

Lemma 13.1 (CLRS)

A red-black tree with n internal nodes has height at most $\boldsymbol{h \leqslant 2\lg(n+1)}$.

Proof sketch:

1. **Claim:** A subtree rooted at x has at least $2^{\mathrm{bh}(x)} - 1$ internal nodes.
   (Proof by induction on the height of x.)
2. By property 4, at least **half** the nodes on any root-to-leaf path are black.
   $\Rightarrow \mathrm{bh}(root) \geqslant h/2$
3. Therefore: $n \geqslant 2^{h/2} - 1$, so $n + 1 \geqslant 2^{h/2}$, giving $h \leqslant 2\lg(n+1)$.

Key Idea

All BST operations are $\mathcal{O}(\lg n)$ on a red-black tree!

Search, Min, Max, Successor, Predecessor: direct from BST theory.
Insert and Delete: need special care to maintain the RB properties.

Part 6

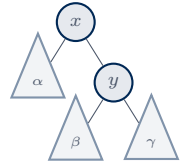
Rotations (CLRS 13.2)

Rotations — The Key Mechanism

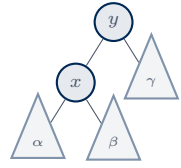
Rotation

A **rotation** is a local restructuring of a BST that preserves the BST property and runs in $\mathcal{O}(1)$.

- **Inorder preserved:** $\alpha < x < \beta < y < \gamma$ in both trees
- Only a constant number of pointers updated
- Mechanism used to rebalance after insert/delete



↓ LEFT-ROTATE(T, x)



Left Rotation — Pseudocode

Left-Rotate(T, x)**:**

Assumes $x.right \neq T.nil$

```
1:  y ← x.right
2:  x.right ← y.left {turn β into x's right subtree}
3:  if y.left ≠ T.nil then
4:      y.left.p ← x
5:  end if
6:  y.p ← x.p {link x's parent to y}
7:  if x.p = T.nil then
8:      T.root ← y
9:  else if x = x.p.left then
10:     x.p.left ← y
11: else
12:     x.p.right ← y
13: end if
14: y.left ← x {put x on y's left}
15: x.p ← y
```

Key Idea

Key properties:

- Runs in $\mathcal{O}(1)$ — only pointer changes
- Preserves BST property
- RIGHT-ROTATE is the symmetric operation
- In an n-node BST, there are exactly $n - 1$ possible rotations

Part 7

Red-Black Tree Insertion (CLRS 13.3)

Insertion Strategy

Step 1: Insert as in a regular BST (using $T.nil$ instead of NIL).
Step 2: Color the new node **red**.
Step 3: Call RB-INSERT-FIXUP to restore RB properties.

Why red?

Coloring the new node red preserves property 5 (black-heights unchanged).
But it may violate **property 2** (root is black) or **property 4** (no two consecutive reds).

The fixup procedure walks up the tree, resolving violations by:

- **Recoloring** nodes
- Performing at most **2 rotations**

The analysis considers the color of z's **uncle** (the sibling of z's parent).

RB-Insert-Fixup — The Three Cases

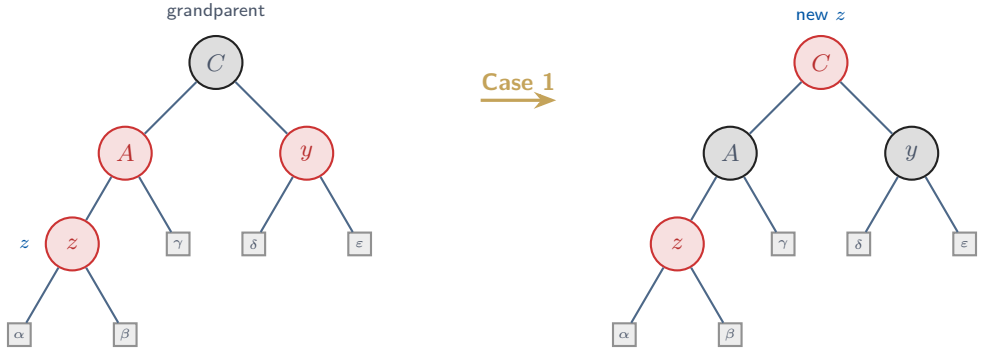
The while loop runs while $z.p$ is red (violation of property 4).
Let $z.p.p$ be the **grandparent** and y be the **uncle**.

(We show the case where $z.p$ is a **left** child of $z.p.p$; the symmetric case is analogous.)

Case	Condition	Action
1	Uncle y is red	Recolor: parent & uncle $\rightarrow$ black, grandparent $\rightarrow$ red. Move z up two levels. Continue loop.
2	Uncle y is black, z is a right child	Left-rotate on $z.p$ $\rightarrow$ transforms into Case 3.
3	Uncle y is black, z is a left child	Recolor parent $\rightarrow$ black, grandparent $\rightarrow$ red. Right-rotate on grandparent. **Done.**

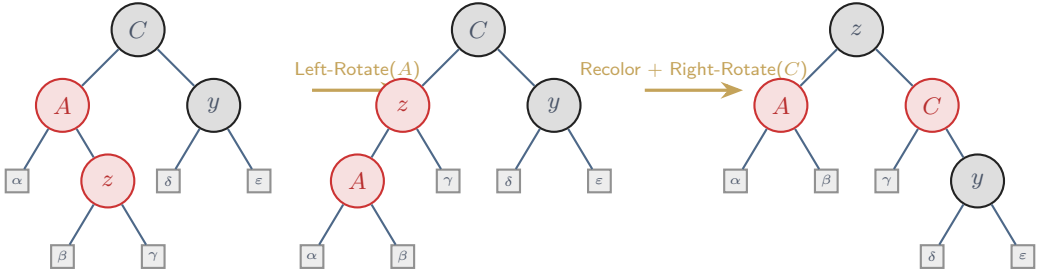
After the loop, set $T.root.color \leftarrow$ BLACK to fix any property 2 violation.

Case 1 — Uncle is Red



- Blackness **pushed down** from grandparent to parent & uncle
- Property 5 preserved: black-height unchanged on all paths
- Pointer z moves up to grandparent $\Rightarrow$ loop continues

Cases 2 & 3 — Uncle is Black



Case 2 **Case 3** ✔ **Done**

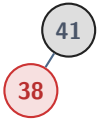
- **Case 2:** z is a right child → left-rotate to make it Case 3
- **Case 3:** z is a left child → recolor + right-rotate → terminates
- **At most 2 rotations total** in the entire fixup!

Insertion — Step-by-Step Example

Insert keys 41, 38, 31, 12, 19, 8 into an empty red-black tree:

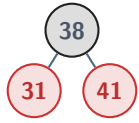
Insert 41 (root → black)

Insert 38 (red, left of 41)



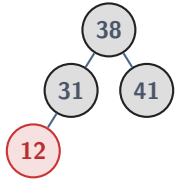
Insert 31 (red → violation!)

Case 3: right-rotate(41)

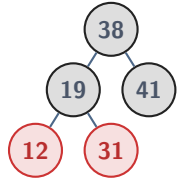


Insert 12 (red, left of 31)

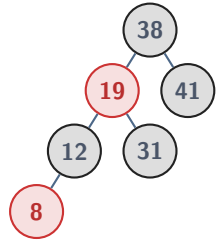
Case 1: recolor



Insert 19 (Case 2 → 3)



Insert 8 (Case 1 → recolor)



Insertion — Complexity Analysis

RB-Insert Complexity

RB-INSERT runs in $\mathcal{O}(\lg n)$ time and performs **at most 2 rotations**.

Breakdown:

- Lines 1–16 (BST insertion): $\mathcal{O}(\lg n)$ — descend from root
- RB-INSERT-FIXUP:
  - Case 1 repeats $\mathcal{O}(\lg n)$ times (pointer moves up 2 levels each time)
  - Cases 2 and 3 execute **at most once** each (then loop terminates)
  - Total rotations: at most 2
- Overall: $\mathcal{O}(\lg n)$ time

Part 8

Red-Black Tree Deletion — Overview (CLRS 13.4)

Deletion — High-Level View

- Deletion follows the same structure as BST deletion, using RB-TRANSPLANT
- After removing/moving a node, if the removed node was **black**:
  1. Property 2: new root might be red
  2. Property 4: two consecutive reds
  3. Property 5: a path is missing a black node → "**doubly black**" node x
- RB-DELETE-FIXUP resolves this via **4 cases** (+ 4 symmetric)

Case	Condition	Action
1	Sibling w is red	Recolor + rotate → convert to cases 2–4
2	w black, both children black	Remove extra black, push up
3	w black, left child red, right black	Recolor + rotate → convert to case 4
4	w black, right child red	Recolor + rotate → **done**

Key Idea

RB-DELETE runs in $\mathcal{O}(\lg n)$ time and performs **at most 3 rotations**.

Deletion — “Extra Black” Intuition

- When a **black node** is removed, one path loses a black → property 5 violated
- We give the replacement node x an “**extra black**”
  - If x was red → becomes “**red-and-black**” → just color it black → **done**
  - If x was black → becomes “**doubly black**” → need to fix
- The fixup **moves the extra black** up the tree until:
  1. It reaches a red-and-black node (color it black)
  2. It reaches the root (extra black disappears)
  3. A rotation resolves the issue

Note

Deletion is the **most complex operation** on red-black trees. Understanding insertion thoroughly is the essential first step. Deletion details are best mastered via implementation practice.

Part 9

In Practice: `std::set`, `std::map` & Python

Red-Black Trees in the Standard Library

C++ STL:

- `std::set<T>` — ordered set
- `std::map<K,V>` — ordered map
- `std::multiset`, `std::multimap`
- All implemented as red-black trees
- All operations: $\mathcal{O}(\log n)$

C++

```cpp
#include <set>
#include <map>
std::set<int> s = {5, 2, 8, 1, 9};
s.insert(3);       // O(log n)
s.find(8);         // O(log n)
s.erase(2);        // O(log n)
// Iterate in sorted order:
for (int x : s) { /* 1,3,5,8,9 */ }
```

Python:

- No built-in BST (`dict`/`set` use hash tables)
- `sortedcontainers` library:

Python

```python
from sortedcontainers import
↪  SortedList

sl = SortedList([5, 2, 8, 1, 9])
sl.add(3)          # O(log n)
sl.remove(2)       # O(log n)
# Iterate in sorted order:
for x in sl:
    print(x)       # 1, 3, 5, 8, 9
```

NOTE: Java's TreeMap/TreeSet also use RB trees.

When to Use What?

	Search	Insert	Delete	Ordered?
Hash table (`dict`)	$\mathcal{O}(1)^*$	$\mathcal{O}(1)^*$	$\mathcal{O}(1)^*$	✗
BST (unbalanced)	$\mathcal{O}(h)$	$\mathcal{O}(h)$	$\mathcal{O}(h)$	✓
Red-black tree (`std::map`)	$\mathcal{O}(\log n)$	$\mathcal{O}(\log n)$	$\mathcal{O}(\log n)$	✓
Sorted array	$\mathcal{O}(\log n)$	$\mathcal{O}(n)$	$\mathcal{O}(n)$	✓

* amortized average case

Decision Heuristic

- Need **ordered iteration**, **range queries**, **min/max**? → Balanced BST (`std::map`, `SortedList`)
- Just need fast lookup/insert/delete, order doesn't matter? → Hash table (`dict`, `unordered_map`)

Part 10

Summary & What's Next

Summary — Key Takeaways

1. **Binary Search Trees:** combine sorted order with dynamic operations, all in $\mathcal{O}(h)$
2. **BST operations:** search, min/max, successor/predecessor, insert, delete
3. **The height problem:** plain BSTs can degenerate to $\Theta(n)$ height
4. **Red-Black Trees:** 5 properties guarantee $h \leqslant 2\lg(n+1)$
5. **Rotations:** $\mathcal{O}(1)$ local restructuring that preserves BST property
6. **Insertion fixup:** 3 cases, at most 2 rotations, $\mathcal{O}(\lg n)$ total
7. **Deletion fixup:** 4 cases, at most 3 rotations, $\mathcal{O}(\lg n)$ total
8. **In practice:** `std::set`/`std::map` = red-black trees

Key Idea

The big picture: Red-black trees give us worst-case $\mathcal{O}(\log n)$ for all dictionary operations while maintaining sorted order.

What's Next — Week 4

→ Next week: Advanced Dynamic Programming (CLRS Ch. 14)

- Rod cutting: top-down vs bottom-up
- Matrix chain multiplication
- Optimal substructure & overlapping subproblems
- Longest common subsequence (LCS)
- DP vs Greedy: when to use which?

To prepare: Review your S1 notes on memoization and dynamic programming (knapsack, Levenshtein).

Questions?

References

- **Cormen, Leiserson, Rivest, Stein** — *Introduction to Algorithms*, 4th ed., MIT Press, 2022. **Chapters 12–13.**

PSL UNIVERSITÉ PARIS

PSL University · Bachelor of Science in AI · 2025–2026