Lab Session 4

Binary Search Trees & Red-Black Trees

The Art of Computer Programming 2 — Week 4 — March 13, 2026

Objectives

By the end of this lab, you should be able to:

- Trace BST operations by hand (insertion, search, deletion, tree walks)
- Implement a binary search tree from scratch in Python
- Observe and measure the height problem on degenerate inputs
- Verify red-black tree properties on given trees
- Implement rotations and RB-Insert-Fixup
- Compare BST vs. balanced BST vs. hash table performance empirically

1 Warm-up: BST Operations by Hand (15 min)

Pen and paper — no computer needed.

1.1 Insertion and tree walks (CLRS 12.1–12.2)

Consider an initially empty BST. Insert the following keys **in order**:

$$15,\ 6,\ 18,\ 3,\ 7,\ 17,\ 20,\ 2,\ 4,\ 13,\ 9$$

(a) Draw the resulting BST after all insertions.

(b) What is the height h of the tree?

(c) Write the output of an **inorder** walk.

(d) Write the output of a **preorder** walk.

(e) Write the output of a **postorder** walk.

1.2 Search, min, max, successor

Using the BST from Section 1.1:

(a) Trace the search path for key 13. How many comparisons are needed?

(b) Trace the search path for key 10 (not in the tree). Where does the search stop?

(c) What is the minimum? The maximum?

(d) What is the successor of node 13? Explain which case applies.

(e) What is the predecessor of node 7?

1.3 Deletion (CLRS 12.3)

Starting from the BST of Section 1.1, perform the following deletions **in order**:

(a) Delete key 2 (no children). Draw the resulting tree.

(b) Delete key 18 (two children). Which node replaces it? Draw the resulting tree.

(c) Delete key 6 (two children). Draw the final tree.

Hint

When deleting a node with two children, the successor (minimum of the right subtree) takes its place.

2 Implement a Binary Search Tree (25 min)

2.1 Node class and basic structure

```python
class BSTNode:
    """A node in a binary search tree."""
    def __init__(self, key, left=None, right=None, parent=None):
        self.key = key
        self.left = left
        self.right = right
        self.parent = parent

    def __repr__(self):
        return f"BSTNode({self.key})"


class BST:
    """Binary Search Tree."""

    def __init__(self):
        self.root = None
        self.size = 0

    def insert(self, key):
        """Insert a key into the BST (CLRS Tree-Insert)."""
        # YOUR CODE HERE
        pass

    def search(self, key):
        """Search for a key. Return the node or None."""
        # YOUR CODE HERE
        pass

    def minimum(self, node=None):
        """Return the node with the minimum key in the subtree."""
        # YOUR CODE HERE
        pass

    def maximum(self, node=None):
        """Return the node with the maximum key in the subtree."""
        # YOUR CODE HERE
        pass

    def successor(self, node):
        """Return the successor of the given node, or None."""
        # YOUR CODE HERE
        pass

    def delete(self, key):
```

```
        """Delete the node with the given key."""
        # YOUR CODE HERE
        pass

    def _transplant(self, u, v):
        """Replace subtree rooted at u with subtree rooted at v."""
        # YOUR CODE HERE
        pass

    def inorder(self, node=None, result=None):
        """Return a list of keys in inorder."""
        if result is None:
            result = []
            if node is None:
                node = self.root
        if node is not None:
            self.inorder(node.left, result)
            result.append(node.key)
            self.inorder(node.right, result)
        return result

    def height(self, node=None):
        """Return the height of the tree (or subtree)."""
        if node is None:
            node = self.root
        if node is None:
            return -1
        return 1 + max(self.height(node.left), self.height(node.right))

    def display(self, node=None, prefix="", is_left=True):
        """Pretty-print the tree."""
        if node is None:
            node = self.root
        if node is not None:
            print(prefix + ("├── " if is_left else "└── ") + str(node.key))
            new_prefix = prefix + ("│   " if is_left else "    ")
            if node.left or node.right:
                self.display(node.left, new_prefix, True)
                self.display(node.right, new_prefix, False)
```

2.2 Test your implementation

```
tree = BST()
for key in [15, 6, 18, 3, 7, 17, 20, 2, 4, 13, 9]:
    tree.insert(key)

# Verify inorder gives sorted output
assert tree.inorder() == [2, 3, 4, 6, 7, 9, 13, 15, 17, 18, 20]

# Search
assert tree.search(13).key == 13
assert tree.search(10) is None

# Min / Max
assert tree.minimum().key == 2
assert tree.maximum().key == 20
```

```python
# Successor
node_13 = tree.search(13)
assert tree.successor(node_13).key == 15

# Delete
tree.delete(2)
assert tree.search(2) is None
assert tree.inorder() == [3, 4, 6, 7, 9, 13, 15, 17, 18, 20]

tree.delete(18)
assert tree.inorder() == [3, 4, 6, 7, 9, 13, 15, 17, 20]

tree.delete(6)
assert tree.inorder() == [3, 4, 7, 9, 13, 15, 17, 20]

tree.display()
print("All tests passed!")
```

3 The Height Problem (15 min)

Recall

All BST operations are $\mathcal{O}(h)$. Best case: $h = \Theta(\log n)$ (balanced). Worst case: $h = \Theta(n)$ (degenerate, e.g., sorted input).

3.1 Observe the degenerate case

1. Build a BST by inserting keys $1, 2, 3, \ldots, n$ in sorted order for $n \in \{100, 500, 1000, 5000\}$.
2. Measure the height of each tree. What do you observe?
3. Measure the time to search for a random key in each tree.

```python
import time, random

for n in [100, 500, 1000, 5000]:
    tree = BST()
    for i in range(1, n + 1):
        tree.insert(i)  # sorted order!
    h = tree.height()

    start = time.perf_counter()
    for _ in range(1000):
        tree.search(random.randint(1, n))
    elapsed = time.perf_counter() - start

    print(f"n={n:>5}, height={h:>5}, "
          f"1000 searches: {elapsed:.4f}s")
```

3.2 Random vs. sorted insertion

1. For $n = 10000$, build two BSTs: one with keys inserted in **sorted order**, one with keys in **random order**.
2. Compare heights and average search times.

3. Plot the height as a function of n (from 100 to 10000) for both insertion orders. On the same plot, draw the lines $y = n - 1$ (worst case) and $y = c \cdot \log_2 n$ (best case).

Hint

Use `random.shuffle()` to generate random insertion order. Use `matplotlib` for the plot.

Question: Why does the expected height of a random BST relate to the analysis of randomized quicksort?

4 Red-Black Tree Properties (10 min)

Recall

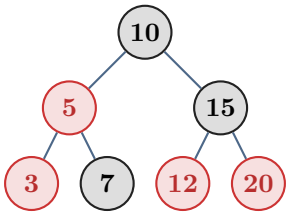
A red-black tree is a BST with 5 properties:

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, both its children are black.
5. All paths from a node to descendant leaves have the same number of black nodes.

Height guarantee: $h \leqslant 2\lg(n+1)$.

4.1 Verify RB properties

For the following tree, verify whether it is a valid red-black tree. If not, identify which property is violated.



4.2 Black-height computation

For the red-black tree shown in the handout (Figure 2, with root 26), compute:

(a) bh(root)

(b) bh(17)

(c) bh(41)

(d) Verify property 5 by listing all root-to-NIL paths through node 14 and counting black nodes.

4.3 Node count bounds (CLRS 13.1-6)

What are the **largest** and **smallest** possible numbers of internal nodes in a red-black tree with black-height k?

5 Implement a Red-Black Tree (20 min)

5.1 RB-Tree node and structure

```python
RED = True
BLACK = False

class RBNode:
    """A node in a red-black tree."""
    def __init__(self, key, color=RED):
        self.key = key
        self.color = color
        self.left = None
        self.right = None
        self.parent = None

    def __repr__(self):
        c = "R" if self.color == RED else "B"
        return f"RBNode({self.key}, {c})"


class RedBlackTree:
    """Red-Black Tree (CLRS Chapter 13)."""

    def __init__(self):
        self.NIL = RBNode(key=None, color=BLACK)
        self.root = self.NIL
        self.size = 0

    def left_rotate(self, x):
        """Left rotation on node x (CLRS Left-Rotate)."""
        # YOUR CODE HERE
        pass

    def right_rotate(self, y):
        """Right rotation on node y (symmetric of left_rotate)."""
        # YOUR CODE HERE
        pass

    def insert(self, key):
        """Insert key into the RB tree (CLRS RB-Insert)."""
        z = RBNode(key)
        z.left = self.NIL
        z.right = self.NIL

        # Phase 1: BST insertion (using self.NIL instead of None)
        y = self.NIL
        x = self.root
        while x is not self.NIL:
            y = x
            if z.key < x.key:
                x = x.left
            else:
                x = x.right
        z.parent = y
        if y is self.NIL:
            self.root = z
        elif z.key < y.key:
            y.left = z
```

```
        else:
            y.right = z

        # Phase 2: Color red and fix up
        z.color = RED
        self.size += 1
        self._insert_fixup(z)

    def _insert_fixup(self, z):
        """RB-Insert-Fixup: restore RB properties after insertion."""
        # YOUR CODE HERE
        pass

    def search(self, key):
        """Search for a key. Return the node or self.NIL."""
        x = self.root
        while x is not self.NIL and key != x.key:
            if key < x.key:
                x = x.left
            else:
                x = x.right
        return x

    def inorder(self, node=None, result=None):
        """Return a list of keys in inorder."""
        if result is None:
            result = []
            if node is None:
                node = self.root
        if node is not self.NIL and node is not None:
            self.inorder(node.left, result)
            result.append(node.key)
            self.inorder(node.right, result)
        return result

    def height(self, node=None):
        """Return the height of the tree."""
        if node is None:
            node = self.root
        if node is self.NIL:
            return -1
        return 1 + max(self.height(node.left), self.height(node.right))

    def black_height(self, node=None):
        """Compute the black-height of a node."""
        if node is None:
            node = self.root
        bh = 0
        x = node
        while x is not self.NIL:
            if x.color == BLACK:
                bh += 1
            x = x.left
        return bh

    def verify_properties(self):
        """Check all 5 RB properties. Return True if valid."""
```

```
        # Property 2: root is black
        if self.root.color != BLACK:
            return False, "Property 2: root is not black"

        # Properties 4 and 5
        def check(node, black_count, path_black_count):
            if node is self.NIL:
                if path_black_count[0] is None:
                    path_black_count[0] = black_count
                elif black_count != path_black_count[0]:
                    return False, "Property 5: unequal black counts"
                return True, ""

            # Property 4
            if node.color == RED:
                if node.left.color == RED or node.right.color == RED:
                    return False, f"Property 4: red node {node.key} has a red child"

            new_count = black_count + (1 if node.color == BLACK else 0)
            ok, msg = check(node.left, new_count, path_black_count)
            if not ok:
                return ok, msg
            return check(node.right, new_count, path_black_count)

        return check(self.root, 0, [None])
```

Important

The sentinel `self.NIL` is shared by all nodes. Every leaf and the root's parent point to it. It is always **black**. Don't confuse `self.NIL` (sentinel node) with Python's `None`.

5.2 Implement rotations

Implement `left_rotate` and `right_rotate` following the pseudocode from CLRS.

Hint

A left rotation on node x: x's right child y becomes the new subtree root, x goes to the left of y, and y's left subtree β becomes x's right subtree. Remember to update parent pointers!

5.3 Implement RB-Insert-Fixup

Implement `_insert_fixup` following the pseudocode from CLRS. Handle the 3 cases (and their symmetric versions):

1. Uncle is red → recolor
2. Uncle is black, z is a right child → left-rotate parent
3. Uncle is black, z is a left child → recolor + right-rotate grandparent

5.4 Test your RB tree

```
rbt = RedBlackTree()

# Insert the example from the handout
for key in [41, 38, 31, 12, 19, 8]:
```

```
    rbt.insert(key)
    ok, msg = rbt.verify_properties()
    assert ok, f"After inserting {key}: {msg}"

assert rbt.inorder() == [8, 12, 19, 31, 38, 41]
assert rbt.root.key == 38
assert rbt.root.color == BLACK
print(f"Height: {rbt.height()}, Black-height: {rbt.black_height()}")

# Larger test
rbt2 = RedBlackTree()
import random
keys = list(range(1, 101))
random.shuffle(keys)
for k in keys:
    rbt2.insert(k)
    ok, msg = rbt2.verify_properties()
    assert ok, f"After inserting {k}: {msg}"

assert rbt2.inorder() == list(range(1, 101))
print(f"n=100: height={rbt2.height()}, "
      f"black-height={rbt2.black_height()}")
print(f"Theoretical max height: {2 * (100+1).bit_length()}")

print("All RB tree tests passed!")
```

6 BST vs. Red-Black Tree: Performance Comparison (10 min)

6.1 Height comparison

For $n \in \{100, 500, 1000, 5000, 10000\}$:

1. Build a plain BST with keys inserted in **sorted order**.
2. Build a red-black tree with keys inserted in **sorted order**.
3. Compare the heights.

```
import math

for n in [100, 500, 1000, 5000, 10000]:
    # Plain BST (sorted insertion)
    bst = BST()
    for i in range(1, n + 1):
        bst.insert(i)

    # Red-black tree (sorted insertion)
    rbt = RedBlackTree()
    for i in range(1, n + 1):
        rbt.insert(i)

    print(f"n={n:>5}: BST height={bst.height():>5}, "
          f"RBT height={rbt.height():>3}, "
          f"2*lg(n+1)={2*math.log2(n+1):.1f}")
```

Question: What happens to the BST? What about the red-black tree?

6.2 Search time comparison

Measure the average time to search for 10 000 random keys in a BST (sorted insertion) vs. a red-black tree (sorted insertion) for $n = 5000$.

```python
import time, random

n = 5000
bst = BST()
rbt = RedBlackTree()
for i in range(1, n + 1):
    bst.insert(i)
    rbt.insert(i)

queries = [random.randint(1, n) for _ in range(10_000)]

start = time.perf_counter()
for q in queries:
    bst.search(q)
bst_time = time.perf_counter() - start

start = time.perf_counter()
for q in queries:
    rbt.search(q)
rbt_time = time.perf_counter() - start

print(f"BST search time:  {bst_time:.4f}s")
print(f"RBT search time:  {rbt_time:.4f}s")
print(f"Speedup: {bst_time / rbt_time:.1f}x")
```

Bonus Exercises

For students who finish early.

Bonus 1 — Iterative inorder traversal

Implement an iterative (non-recursive) inorder traversal using an explicit stack. Then implement another version using successor calls (call `minimum`, then $n-1$ calls to `successor`). Verify that both approaches produce the same output and both run in $\Theta(n)$.

```python
def inorder_iterative_stack(tree):
    """Inorder traversal using an explicit stack."""
    # YOUR CODE HERE
    pass

def inorder_iterative_successor(tree):
    """Inorder traversal using minimum + successor calls."""
    # YOUR CODE HERE
    pass
```

Bonus 2 — RB-Delete

Implement RB-Delete and RB-Delete-Fixup following CLRS Section 13.4. Test by inserting 100 random keys and then deleting them in random order, verifying RB properties after each deletion.

```python
def delete(self, key):
    """Delete a key from the RB tree (CLRS RB-Delete)."""
    # YOUR CODE HERE
    pass

def _delete_fixup(self, x):
    """RB-Delete-Fixup: handle the 4 cases."""
    # YOUR CODE HERE
    pass
```

Bonus 3 — BST vs. hash table vs. sorted list ⋆

Compare empirically:

1. Your red-black tree (`insert`, `search`, `inorder`)
2. Python's `dict` (`d[k]=v`, `k in d`, `sorted(d)`)
3. `sortedcontainers.SortedList` (`add`, `__contains__`, iteration)

For $n = 10000$ random insertions, measure: (a) total insertion time, (b) 10 000 random lookups, (c) producing sorted output. Which wins at each task?

Key Formulas

BST operations (all)	$\mathcal{O}(h)$
BST best-case height	$h = \lfloor \log_2 n \rfloor = \Theta(\log n)$
BST worst-case height	$h = n - 1 = \Theta(n)$
RB tree height bound	$h \leqslant 2\lg(n+1)$
RB tree — all operations	$\mathcal{O}(\log n)$ worst case
RB insert — max rotations	2
RB delete — max rotations	3
Inorder walk	$\Theta(n)$

*Next week: Dynamic Programming (CLRS Ch. 14–15) — when subproblems overlap, **memoize**.*