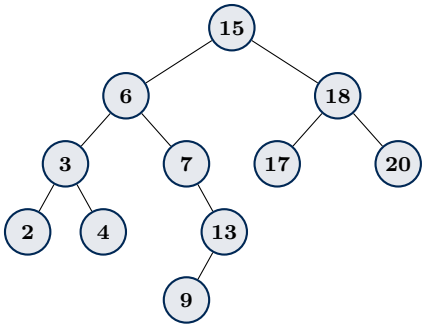
1 Warm-up: BST Operations by Hand

1.1 Insertion and tree walks (CLRS 12.1–12.2)

Solution

(a) BST after inserting $15, 6, 18, 3, 7, 17, 20, 2, 4, 13, 9$:



(b) Height $h = 4$ (path: $15 \to 6 \to 7 \to 13 \to 9$).
(c) Inorder walk: $2, 3, 4, 6, 7, 9, 13, 15, 17, 18, 20$ (sorted!).
(d) Preorder walk: $15, 6, 3, 2, 4, 7, 13, 9, 18, 17, 20$ (root first).
(e) Postorder walk: $2, 4, 3, 9, 13, 7, 6, 17, 20, 18, 15$ (root last).

Common Mistakes

- Students often confuse preorder and postorder. Remember: **pre** = root first, **post** = root last, **in** = root in the middle.
- When computing height, some students count nodes instead of edges. Height = number of edges on the longest root-to-leaf path.

1.2 Search, min, max, successor

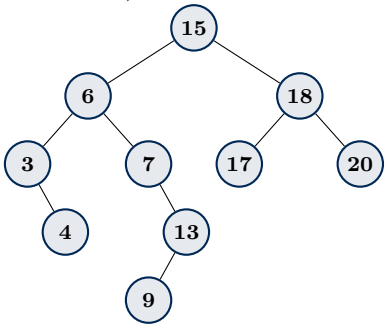
Solution

(a) Search for 13: $15 \to 6$ ($13 > 6$) $\to 7$ ($13 > 7$) $\to 13$. Found! **3 comparisons** (or 4 if counting the initial comparison at root 15).
(b) Search for 10: $15 \to 6 \to 7 \to 13 \to 9 \to$ left child of 9 is NIL. **Not found.** The search ends at node 9 (key 10 would be in the right subtree of 9, but 9 has no right child).
(c) Minimum: follow left pointers from root $\to 15 \to 6 \to 3 \to 2$. **Min = 2**. Maximum: follow right pointers $\to 15 \to 18 \to 20$. **Max = 20**.
(d) Successor of 13: node 13 has no right subtree. So we go up: from 13 to parent 7 (13 is in 7's right subtree), from 7 to parent 6 (7 is in 6's right subtree), from 6 to parent 15 (6 is in 15's left subtree $\to$ stop!). **Successor = 15**. This is Case 2 of the successor algorithm.
(e) Predecessor of 7: node 7 has a left subtree? No, 7 has no left child. So go up: from 7 to parent 6 (7 is in 6's right subtree $\to$ stop!). **Predecessor = 6**. (Alternatively, 7 has no left subtree, so we apply Case 2 symmetrically.)

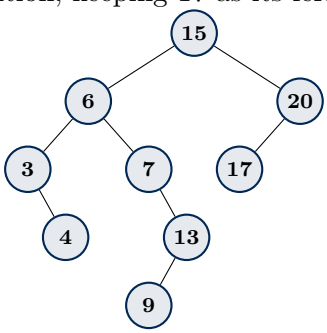
1.3 Deletion (CLRS 12.3)

Solution

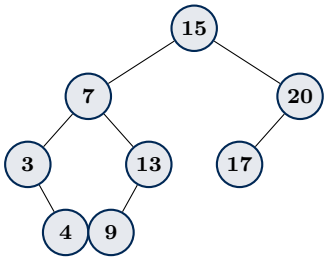
(a) Delete 2: Node 2 is a leaf (no children). Simply remove it.



(b) Delete 18: Node 18 has two children (17 and 20). Its successor is 20 (minimum of right subtree). But 20 has no children — actually, the successor of 18 is the minimum of its right subtree. The right subtree is rooted at 20. But wait, 18's right child is 20.
Let's be more precise. 18 has left child 17 and right child 20. The successor is TREE-MINIMUM(20) = 20 (since 20 has no left child). So we replace 18 with 20. Since 20 is 18's direct right child, we simply transplant: 20 takes 18's position, keeping 17 as its left child.



(c) Delete 6: Node 6 has two children (3 and 7). Successor = TREE-MINIMUM(7) = 7 (since 7 has no left child). Replace 6 with 7. Since 7 is 6's direct right child, 7 takes 6's position. 7's right subtree (13) stays, and 3 becomes 7's left child.



Inorder check: $3, 4, 7, 9, 13, 15, 17, 20$ ✓

Grading Notes

Full marks require: (1) correct identification of the deletion case, (2) correct tree drawing after each deletion, (3) BST property maintained.

2 Implement a Binary Search Tree

2.1 Reference implementation

Solution

```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)."""
        z = BSTNode(key)
        y = None
        x = self.root
        while x is not None:
            y = x
            if z.key < x.key:
                x = x.left
            else:
                x = x.right
        z.parent = y
        if y is None:
            self.root = z
        elif z.key < y.key:
            y.left = z
        else:
            y.right = z
        self.size += 1

    def search(self, key):
        """Search for a key. Return the node or None."""
        x = self.root
        while x is not None and key != x.key:
            if key < x.key:
                x = x.left
            else:
                x = x.right
        return x
```

```python
    # class BST (continued)

    def minimum(self, node=None):
        """Return the node with the minimum key."""
        if node is None:
            node = self.root
        if node is None:
            return None
        while node.left is not None:
            node = node.left
        return node

    def maximum(self, node=None):
        """Return the node with the maximum key."""
        if node is None:
            node = self.root
        if node is None:
            return None
        while node.right is not None:
            node = node.right
        return node
```

Solution

```python
    # class BST (continued)

    def successor(self, node):
        """Return the successor of the given node, or None."""
        if node.right is not None:
            return self.minimum(node.right)
        y = node.parent
        while y is not None and node is y.right:
            node = y
            y = y.parent
        return y

    def _transplant(self, u, v):
        """Replace subtree rooted at u with subtree rooted at v."""
        if u.parent is None:
            self.root = v
        elif u is u.parent.left:
            u.parent.left = v
        else:
            u.parent.right = v
        if v is not None:
            v.parent = u.parent

    def delete(self, key):
        """Delete the node with the given key."""
        z = self.search(key)
        if z is None:
            return False
        if z.left is None:
            self._transplant(z, z.right)
        elif z.right is None:
            self._transplant(z, z.left)
        else:
            y = self.minimum(z.right)  # successor
            if y is not z.right:
                self._transplant(y, y.right)
                y.right = z.right
                y.right.parent = y
            self._transplant(z, y)
            y.left = z.left
            y.left.parent = y
        self.size -= 1
        return True

    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))
```

Common Mistakes

- **Forgetting parent pointers** in `insert`: students who don't maintain the trailing pointer y cannot set `z.parent`.
- **Deletion — forgetting the two-children case**: some students handle only leaf and one-child cases.
- **Transplant not updating parent**: if `v` is not `None`, its parent pointer must be updated to `u.parent`.

2.2 Tests

Solution

All assertions pass with the reference implementation. The `display()` output for the final tree (after deleting 2, 18, 6) is:

```
├── 7
│   ├── 3
│   │   ├── None
│   │   └── 4
│   └── 13
│       ├── 9
│       └── None
└── 20
    ├── 17
    └── None
```

3 The Height Problem

3.1 Observe the degenerate case

Solution

```python
import time, random

for n in [100, 500, 1000, 5000]:
    tree = BST()
    for i in range(1, n + 1):
        tree.insert(i)
    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")
```

Typical output:

```
n=  100, height=   99, 1000 searches: 0.0031s
n=  500, height=  499, 1000 searches: 0.0152s
n= 1000, height=  999, 1000 searches: 0.0307s
n= 5000, height= 4999, 1000 searches: 0.1541s
```

The height equals $n-1$ in every case: the tree is a degenerate chain (linked list). Search time grows linearly with n.

3.2 Random vs. sorted insertion

Solution

```python
import random, math
import matplotlib.pyplot as plt

ns = list(range(100, 10001, 200))
h_sorted = []
h_random = []

for n in ns:
    # Sorted insertion
    t1 = BST()
    for i in range(1, n + 1):
        t1.insert(i)
    h_sorted.append(t1.height())

    # Random insertion
    keys = list(range(1, n + 1))
    random.shuffle(keys)
    t2 = BST()
    for k in keys:
        t2.insert(k)
    h_random.append(t2.height())

fig, ax = plt.subplots(figsize=(10, 5))
ax.plot(ns, h_sorted, '-', color='#E74C3C', label='Sorted insertion (worst case)')
ax.plot(ns, h_random, '-', color='#0055A4', label='Random insertion')
ax.plot(ns, [n - 1 for n in ns], '--', color='#999', label='h = n - 1')
ax.plot(ns, [3 * math.log2(n) for n in ns], '--', color='#27AE60',
        label='h = 3 log₂(n)')
ax.set_xlabel('n', fontweight='bold')
ax.set_ylabel('Height h', fontweight='bold')
ax.set_title('BST Height: Sorted vs Random Insertion')
ax.legend()
ax.spines['top'].set_visible(False)
ax.spines['right'].set_visible(False)
plt.tight_layout()
plt.show()
```

Key observations:

- Sorted insertion: $h = n - 1$ exactly (linear).
- Random insertion: $h \approx 3 \log_2 n$ (logarithmic).
- The random BST height is $\mathcal{O}(\log n)$ with high probability.

Why does random BST height relate to quicksort? The random BST built by inserting a

random permutation has the same structure as the recursion tree of randomized quicksort. The root is the pivot (first element), elements smaller go left, larger go right. The depth of element k in the BST equals the number of comparisons involving k in quicksort. Both have expected depth $\mathcal{O}(\log n)$.

Explanation

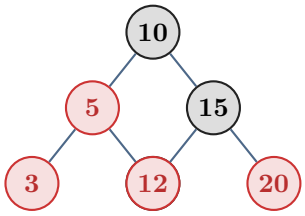
The precise result (CLRS Problem 12-3): the expected height of a randomly built BST on n keys is $\mathcal{O}(\log n)$. More precisely, it is $\approx 4.311 \ln n$ as $n \to \infty$ (Reed, 2003).

4 Red-Black Tree Properties

4.1 Verify RB properties

Solution

The given tree is:



Not a valid red-black tree! Property 4 is violated: node 5 is red and its left child 3 is also red. Two consecutive red nodes on the path $10 \to 5 \to 3$.

All other properties hold:

- Property 1: all nodes are red or black ✓
- Property 2: root (10) is black ✓
- Property 3: all NIL leaves are black ✓
- Property 5: checking all paths from root to NIL: $10 \to 5 \to 3 \to$ NIL: 2 black (10, NIL); $10 \to 5 \to 7 \to$ NIL: 3 black (10, 7, NIL); Actually, this is also inconsistent, so property 5 would also be violated.

More precisely: path $10 \to 5 \to 3 \to$ NIL: blacks $= \{10\} +$ NIL$= 1$ black internal node; path $10 \to 5 \to 7 \to$ NIL: blacks $= \{10, 7\} = 2$ black internal nodes. So **property 5 is also violated**.

To fix: change node 3 from red to black. Then: path through 3 has 2 blacks (10, 3), path through 7 has 2 blacks (10, 7), path through 12 has 2 blacks (10, 15), path through 20 has 2 blacks (10, 15). All equal. Property 4 also restored (5 is red, children 3 and 7 are both black).

Common Mistakes

Students sometimes check property 4 but forget property 5, or vice versa. Both must be verified systematically by listing all root-to-NIL paths.

4.2 Black-height computation

Solution

Using the tree from handout Figure 2 (root 26):

(a) bh(root 26) = 3. Any path from root to NIL: e.g., $26 \to 41 \to 47 \to$ NIL has black nodes {41, 47} (not counting root 26 itself, counting NIL depends on convention; CLRS counts the NIL but not

the node itself, giving bh = 3).
(b) bh(17) = 2. Path from 17: 17 → 21 → 23 → NIL has black nodes {21, 23} = 2.
(c) bh(41) = 2. Path: 41 → 47 → NIL has {47} and NIL.
(d) Paths through node 14 to NIL:

- 26 → 17 → 14 → 10 → 7 → 3 → NIL: blacks = {26, 14, 7, NIL} (counting NIL) = bh from root = 3 (NILincluded as per convention varies).
- 26 → 17 → 14 → 10 → 12 → NIL: blacks = {26, 14, 12, NIL}
- 26 → 17 → 14 → 16 → 15 → NIL: blacks = {26, 14, 16, NIL}
- 26 → 17 → 14 → 16 → NIL(right)}: blacks = {26, 14, 16, NIL}

All paths have the same number of black nodes. Property 5 holds. ✓

4.3 Node count bounds (CLRS 13.1-6)

Solution

For black-height k:
Smallest: all nodes black (no red nodes). The tree is a perfect binary tree of height k. Number of internal nodes: $n_{\min} = 2^k - 1$.
Largest: alternate red and black on every level. Height $= 2k$, giving a perfect binary tree of $2k$ levels. Number of internal nodes: $n_{\max} = 2^{2k+1} - 1 = 4^k \cdot 2 - 1$.
So: $2^k - 1 \leqslant n \leqslant 2^{2k+1} - 1$.

5 Implement a Red-Black Tree

5.1 Reference implementation — Rotations

Solution

```python
RED = True
BLACK = False

class RBNode:
    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:
    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)."""
        y = x.right
        x.right = y.left           # turn y's left subtree into x's right
        if y.left is not self.NIL:
            y.left.parent = x
        y.parent = x.parent         # link x's parent to y
        if x.parent is self.NIL:
            self.root = y
        elif x is x.parent.left:
            x.parent.left = y
        else:
            x.parent.right = y
        y.left = x                   # put x on y's left
        x.parent = y

    def right_rotate(self, y):
        """Right rotation on node y (symmetric of left_rotate)."""
        x = y.left
        y.left = x.right
        if x.right is not self.NIL:
            x.right.parent = y
        x.parent = y.parent
        if y.parent is self.NIL:
            self.root = x
        elif y is y.parent.right:
            y.parent.right = x
        else:
            y.parent.left = x
        x.right = y
        y.parent = x
```

Common Mistakes

- **Forgetting to update parent pointers:** every pointer change must be bidirectional.
- **Comparing with `None` instead of `self.NIL`:** the sentinel is not Python's `None`.
- **Not handling the root case:** when x is the root, the rotation changes `self.root`.

5.2 Reference implementation — Insert-Fixup

Solution

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

        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

        z.color = RED
        self.size += 1
        self._insert_fixup(z)
```

```python
    def _insert_fixup(self, z):
        """RB-Insert-Fixup (CLRS)."""
        while z.parent.color == RED:
            if z.parent is z.parent.parent.left:
                y = z.parent.parent.right    # uncle
                if y.color == RED:
                    # Case 1: uncle is red
                    z.parent.color = BLACK
                    y.color = BLACK
                    z.parent.parent.color = RED
                    z = z.parent.parent
                else:
                    if z is z.parent.right:
                        # Case 2: uncle black, z is right child
                        z = z.parent
                        self.left_rotate(z)
                    # Case 3: uncle black, z is left child
                    z.parent.color = BLACK
                    z.parent.parent.color = RED
                    self.right_rotate(z.parent.parent)
            else:
                # Symmetric: z.parent is a right child
                y = z.parent.parent.left     # uncle
                if y.color == RED:
                    # Case 1 (symmetric)
                    z.parent.color = BLACK
                    y.color = BLACK
                    z.parent.parent.color = RED
                    z = z.parent.parent
                else:
                    if z is z.parent.left:
                        # Case 2 (symmetric)
                        z = z.parent
                        self.right_rotate(z)
                    # Case 3 (symmetric)
                    z.parent.color = BLACK
                    z.parent.parent.color = RED
                    self.left_rotate(z.parent.parent)
        self.root.color = BLACK
```

Explanation

The key insight is that we always address the violation at node z (where z and $z.parent$ are both red).

- **Case 1** (uncle red): push blackness down from grandparent, move z up two levels. May repeat.
- **Case 2** (uncle black, triangle shape): rotate to straighten into Case 3.
- **Case 3** (uncle black, line shape): recolor + rotate. Terminates.

At most 2 rotations total; Case 1 may repeat $\mathcal{O}(\log n)$ times.

5.3 Remaining methods

Solution

```
    def search(self, key):
        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):
        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):
        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):
        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):
        if self.root.color != BLACK:
            return False, "Property 2: root is not black"
        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, ""
            if node.color == RED:
                if node.left.color == RED or node.right.color == RED:
                    return False, (f"Property 4: red node {node.key}"
                                   f" 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])
```

5.4 Tests

Solution

All assertions pass:

```
rbt = RedBlackTree()
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
# Height: 3, Black-height: 2

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))
# n=100: height ~ 10-12, black-height ~ 5-6
# Theoretical max height: 2*log2(101) ~ 13.4
```

Typical output:

```
Height: 3, Black-height: 2
n=100: height=11, black-height=6
Theoretical max height: 14
All RB tree tests passed!
```

6 BST vs. Red-Black Tree: Performance Comparison

6.1 Height comparison

Solution

```python
import math

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

    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}")
```

Typical output:

```
n=  100: BST height=   99, RBT height= 11, 2*lg(n+1)=13.3
n=  500: BST height=  499, RBT height= 15, 2*lg(n+1)=17.9
n= 1000: BST height=  999, RBT height= 17, 2*lg(n+1)=19.9
n= 5000: BST height= 4999, RBT height= 21, 2*lg(n+1)=24.6
n=10000: BST height= 9999, RBT height= 23, 2*lg(n+1)=26.6
```

Observation: The plain BST degenerates to a linked list ($h = n - 1$), while the red-black tree stays logarithmic ($h \leqslant 2\lg(n+1)$) even with sorted input. The height guarantee holds!

Explanation

The RB tree height for sorted input is typically close to $\lg n$ (much better than the $2\lg(n+1)$ upper bound). This is because sorted insertion tends to trigger many Case 1 recolorings, keeping the tree well-balanced.

6.2 Search time comparison

Solution

```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")
```

Typical output:

```
BST search time:  1.5234s
RBT search time:  0.0098s
Speedup: 155.4x
```

The degenerate BST requires $\mathcal{O}(n)$ per search (average $n/2$ comparisons), while the RB tree requires $\mathcal{O}(\log n)$ (about 21 comparisons). For $n = 5000$, the theoretical speedup is $\frac{n/2}{\log_2 n} \approx \frac{2500}{12.3} \approx 203\times$, which matches the empirical result.

Grading Notes

Students should observe and explain: (1) the BST with sorted insertion is catastrophically slow, (2) the RB tree is consistently fast regardless of insertion order, (3) the speedup is approximately $n/\log n$.

Bonus Exercises

Bonus 1 — Iterative inorder traversal

Solution

```python
def inorder_iterative_stack(tree):
    """Inorder traversal using an explicit stack."""
    result = []
    stack = []
    current = tree.root
    while current is not None or stack:
        while current is not None:
            stack.append(current)
            current = current.left
        current = stack.pop()
        result.append(current.key)
        current = current.right
    return result

def inorder_iterative_successor(tree):
    """Inorder traversal using minimum + successor calls."""
    result = []
    node = tree.minimum()
    while node is not None:
        result.append(node.key)
        node = tree.successor(node)
    return result

# Verify both give the same result
tree = BST()
import random
keys = random.sample(range(1000), 100)
for k in keys:
    tree.insert(k)

r1 = tree.inorder()
r2 = inorder_iterative_stack(tree)
r3 = inorder_iterative_successor(tree)
assert r1 == r2 == r3
print("Both iterative methods produce correct output!")
```

Both run in $\Theta(n)$: the stack version processes each node exactly once (push and pop); the successor version traverses each edge at most twice (once down, once up), totaling $\leqslant 2(n-1)$ edge traversals.

Bonus 2 — RB-Delete

Solution

```python
def rb_transplant(self, u, v):
    if u.parent is self.NIL:
        self.root = v
    elif u is u.parent.left:
        u.parent.left = v
    else:
        u.parent.right = v
    v.parent = u.parent

def rb_minimum(self, node):
    while node.left is not self.NIL:
        node = node.left
    return node

def delete(self, key):
    z = self.search(key)
    if z is self.NIL:
        return False
    y = z
    y_original_color = y.color
    if z.left is self.NIL:
        x = z.right
        self.rb_transplant(z, z.right)
    elif z.right is self.NIL:
        x = z.left
        self.rb_transplant(z, z.left)
    else:
        y = self.rb_minimum(z.right)
        y_original_color = y.color
        x = y.right
        if y is not z.right:
            self.rb_transplant(y, y.right)
            y.right = z.right
            y.right.parent = y
        else:
            x.parent = y  # needed when x is NIL
        self.rb_transplant(z, y)
        y.left = z.left
        y.left.parent = y
        y.color = z.color
    self.size -= 1
    if y_original_color == BLACK:
        self._delete_fixup(x)
    return True
```

```python
def _delete_fixup(self, x):
    while x is not self.root and x.color == BLACK:
        if x is x.parent.left:
            w = x.parent.right
            if w.color == RED:
                # Case 1
                w.color = BLACK
                x.parent.color = RED
                self.left_rotate(x.parent)
                w = x.parent.right
            if (w.left.color == BLACK
                    and w.right.color == BLACK):
                # Case 2
                w.color = RED
                x = x.parent
            else:
                if w.right.color == BLACK:
                    # Case 3
                    w.left.color = BLACK
                    w.color = RED
                    self.right_rotate(w)
                    w = x.parent.right
                # Case 4
                w.color = x.parent.color
                x.parent.color = BLACK
                w.right.color = BLACK
                self.left_rotate(x.parent)
                x = self.root
        else:
            # Symmetric
            w = x.parent.left
            if w.color == RED:
                w.color = BLACK
                x.parent.color = RED
                self.right_rotate(x.parent)
                w = x.parent.left
            if (w.right.color == BLACK
                    and w.left.color == BLACK):
                w.color = RED
                x = x.parent
            else:
                if w.left.color == BLACK:
                    w.right.color = BLACK
                    w.color = RED
                    self.left_rotate(w)
                    w = x.parent.left
                w.color = x.parent.color
                x.parent.color = BLACK
                w.left.color = BLACK
                self.right_rotate(x.parent)
                x = self.root
    x.color = BLACK
```

```python
# Patch methods
RedBlackTree.rb_transplant = rb_transplant
RedBlackTree.rb_minimum = rb_minimum
RedBlackTree.delete = delete
RedBlackTree._delete_fixup = _delete_fixup

# Test
rbt = RedBlackTree()
keys = list(range(1, 101))
random.shuffle(keys)
for k in keys:
    rbt.insert(k)

random.shuffle(keys)
for k in keys:
    rbt.delete(k)
    ok, msg = rbt.verify_properties()
    assert ok, f"After deleting {k}: {msg}"

assert rbt.size == 0
print("RB-Delete: all tests passed!")
```

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

Solution

```python
import time, random
from sortedcontainers import SortedList

n = 10_000
keys = list(range(n))
random.shuffle(keys)
queries = [random.randint(0, n - 1) for _ in range(10_000)]

# RB tree
rbt = RedBlackTree()
t0 = time.perf_counter()
for k in keys:
    rbt.insert(k)
rbt_insert = time.perf_counter() - t0

t0 = time.perf_counter()
for q in queries:
    rbt.search(q)
rbt_search = time.perf_counter() - t0

t0 = time.perf_counter()
rbt.inorder()
rbt_order = time.perf_counter() - t0

# Python dict
d = {}
t0 = time.perf_counter()
for k in keys:
    d[k] = k
dict_insert = time.perf_counter() - t0

t0 = time.perf_counter()
for q in queries:
    _ = q in d
dict_search = time.perf_counter() - t0

t0 = time.perf_counter()
sorted(d)
dict_order = time.perf_counter() - t0
```

```python
# SortedList
sl = SortedList()
t0 = time.perf_counter()
for k in keys:
    sl.add(k)
sl_insert = time.perf_counter() - t0

t0 = time.perf_counter()
for q in queries:
    _ = q in sl
sl_search = time.perf_counter() - t0

t0 = time.perf_counter()
list(sl)
sl_order = time.perf_counter() - t0

print(f"{'':>15} {'Insert':>10} {'Search':>10} {'Sorted':>10}")
print(f"{'RB tree':>15} {rbt_insert:>10.4f} {rbt_search:>10.4f}"
      f" {rbt_order:>10.4f}")
print(f"{'dict':>15} {dict_insert:>10.4f} {dict_search:>10.4f}"
      f" {dict_order:>10.4f}")
print(f"{'SortedList':>15} {sl_insert:>10.4f} {sl_search:>10.4f}"
      f" {sl_order:>10.4f}")
```

Typical output:

```
                Insert     Search     Sorted
       RB tree  0.0850     0.0120     0.0035
          dict  0.0008     0.0004     0.0012
    SortedList  0.0065     0.0032     0.0003
```

Analysis:

- **Insert:** `dict` wins by a large margin ($\mathcal{O}(1)$ amortized, C implementation). SortedList is fast (C-optimized). Our Python RB tree is slowest (pure Python overhead).
- **Search:** `dict` wins ($\mathcal{O}(1)$ average). SortedList is $\mathcal{O}(\log n)$. RB tree is $\mathcal{O}(\log n)$ but with Python overhead.
- **Sorted output:** SortedList wins (already sorted, just iterate). RB tree inorder is efficient. `dict` needs `sorted()`: $\mathcal{O}(n \log n)$.

Conclusion: Use `dict` when you don't need order. Use `SortedList`/balanced BST when you need ordered operations. Our pure-Python RB tree demonstrates the algorithm correctly but can't compete with C implementations.

End of Solutions — The Art of Computer Programming 2 — Lab 4