LECTURE 7 OF 12

Balanced Search Trees (Red-Black BSTs)

Data Structures & Algorithms

Based on Robert Sedgewick & Kevin Wayne

Algorithms, 4th Edition (Addison-Wesley, 2011)

Chapter 3.3

Duration: 4 hours (with breaks)

Learning Objectives

By the end of this lecture, students will be able to:

1. Define 2-3 trees, explain their perfect balance invariant, and trace the process of insertion with node splitting through concrete examples, analyzing the resulting height bounds.
2. Describe the left-leaning red-black BST representation, establish the one-to-one correspondence between red-black BSTs and 2-3 trees, and explain how red links encode 3-nodes.
3. Implement and trace the three elementary operations on red-black BSTs -- left rotation, right rotation, and color flip -- and explain why each preserves the BST and black-height invariants.
4. Implement insertion in a left-leaning red-black BST by handling the three fix-up cases (rotate left, rotate right, flip colors), and trace insertion of a complete sequence step by step with tree diagrams.
5. Explain and implement deletion in left-leaning red-black BSTs, including the delete-minimum operation and the move-red-left and move-red-right transformations.
6. Prove that the height of a red-black BST with N keys is at most 2 lg(N+1), and compare the performance guarantees of red-black BSTs against AVL trees, B-trees, and hash tables.

Section 1: Motivation and Context (~10 minutes)

Estimated time: 10 minutes

1.1 The Problem with Unbalanced BSTs

In Lecture 6, we studied binary search trees and saw that they provide elegant, efficient implementations of the ordered symbol table API. On *average*, over random insertions, a BST of N keys has height approximately 4.311 ln N, and search, insert, floor, ceiling, rank, and select all take O(lg N) time.

But the operative word is "average." In the worst case, a BST degenerates into a linked list. If you insert keys in sorted order -- 1, 2, 3, 4, 5, 6, 7 -- every key becomes the right child of the previous one, the tree has height N, and every operation takes linear time. This is catastrophic for applications that demand consistent performance, such as database indexing, network routing tables, or real-time systems.

```
BST built from sorted insertions: 1, 2, 3, 4, 5

    1
     \
      2
       \
        3
         \
          4
           \
            5

Height = 5 (worst case: N)
```

Hibbard deletion makes things worse: even starting from a well-balanced tree, a long sequence of random insertions and deletions causes the height to grow as sqrt(N), which is far worse than the lg N we want. This is a deep and somewhat surprising result -- random deletions destroy the randomness of the tree's shape.

1.2 The Goal: Guaranteed Logarithmic Height

We want a search tree structure that guarantees height O(lg N) regardless of the insertion and deletion order. This would give us **worst-case** logarithmic time for all operations:

Operation	Unbalanced BST (worst)	Balanced BST (worst)
Search	N	lg N
Insert	N	lg N
Delete	N	lg N
Min/Max	N	lg N
Floor/Ceiling	N	lg N
Rank/Select	N	lg N

Several balanced tree structures achieve this: AVL trees (Adelson-Velsky and Landis, 1962), 2-3 trees (Hopcroft, 1970), B-trees (Bayer and McCreight, 1970), and red-black trees (Guibas and Sedgewick, 1978). In this lecture, we focus on **2-3 trees** and their binary representation as **left-leaning red-black BSTs** (Sedgewick, 2008).

1.3 Historical Context

The story of balanced search trees is one of the most important chapters in the history of algorithms. AVL trees, invented by Georgy Adelson-Velsky and Evgenii Landis in 1962, were the first self-balancing binary search tree data structure. They maintain the invariant that the heights of the two child subtrees of any node differ by at most one.

In 1970, John Hopcroft introduced 2-3 trees as a simpler conceptual framework for balanced search. Rudolf Bayer and Edward McCreight independently developed B-trees -- a generalization to multi-way trees -- for use in database systems at Boeing Research Labs.

In 1978, Leonidas Guibas and Robert Sedgewick introduced red-black trees as a way to represent 2-3-4 trees (a slight generalization of 2-3 trees) as binary trees with colored links. This was a breakthrough because it combined the conceptual simplicity of 2-3 trees with the implementation simplicity of binary search trees.

In 2008, Sedgewick published a simplified variant -- the **left-leaning red-black BST** (LLRB tree) -- that maintains the additional invariant that red links lean left. This reduces the number of cases in insertion and deletion from the dozen or so in a standard red-black tree to just three, making the code remarkably concise.

The LLRB tree is the primary data structure we study in this lecture. It is the implementation behind Java's `TreeMap` and `TreeSet` (which use a closely related variant), C++'s `std::map` and `std::set`, and many other standard library implementations of ordered symbol tables.

Section 2: 2-3 Trees (~30 minutes)

Estimated time: 30 minutes

2.1 Definition

DEFINITION 2.1 -- 2-3 TREE. A *2-3 search tree* is a tree that is either empty or satisfies the following recursive definition:

- A **2-node** has one key and two children (left and right). All keys in the left subtree are less than the node's key; all keys in the right subtree are greater.
- A **3-node** has two keys and three children (left, middle, right). All keys in the left subtree are less than the smaller key; all keys in the middle subtree are between the two keys; all keys in the right subtree are greater than the larger key.
- **Perfect balance.** Every path from the root to a null link has the same length.

The perfect balance invariant is the crucial property. Unlike ordinary BSTs, where the tree shape depends on the insertion order, a 2-3 tree is always perfectly balanced. The height of a 2-3 tree with N keys is always between floor(log_3(N+1)) and floor(log_2(N+1)).

```
A 2-node:                A 3-node:

    [K]                   [J   K]
   /   \                 /    |    \
  <K     >K            <J    J-K    >K
```

Here is an example of a 2-3 tree containing the keys A C E H L M P R S X:

```
              [M]
            /     \
        [E  J]     [R]
       / |   \    /   \
     [A C] [H] [L] [P] [S X]
```

Every path from the root to a leaf traverses exactly 2 internal nodes (height = 2). The tree stores 10 keys and is perfectly balanced.

2.2 Search in a 2-3 Tree

Search in a 2-3 tree is a natural extension of binary search in a BST:

1. **Compare** the search key against the key(s) in the current node.
2. If the search key equals one of the node's keys, **search hit --** return the associated value.
3. Otherwise, follow the appropriate link to the child subtree:
4. For a 2-node with key K: go left if search key < K, go right if search key > K.
5. For a 3-node with keys J and K: go left if search key < J, go middle if J < search key < K, go right if search key > K.
6. If you reach a null link, **search miss**.

```
Search for H in the tree above:

    Start at root [M]: H < M, go left.
    At [E J]: E < H < J, go middle.
    At [H]: found! Search hit.

Search for B:

    Start at root [M]: B < M, go left.
    At [E J]: B < E, go left.
    At [A C]: A < B < C, but this is a leaf. B != A and B != C.
    Search miss.
```

The search cost is bounded by the height of the tree, which is logarithmic. Each node visit requires at most two key comparisons (for a 3-node), so the total number of comparisons is at most 2 * height.

2.3 Insertion into a 2-3 Tree

Insertion is where 2-3 trees truly shine. The algorithm maintains perfect balance through a process of **node splitting** that propagates upward. There are several cases:

Case 1: Insert into a 2-node leaf.

If the search leads to a 2-node leaf, simply absorb the new key into the node, converting it into a 3-node. The tree's height does not change, and perfect balance is preserved.

```
Insert K into a 2-node [H]:

  Before:  [H]        After:  [H K]
          /   \               / | \
```

Case 2: Insert into a 3-node leaf whose parent is a 2-node.

If the search leads to a 3-node leaf, temporarily absorb the new key to form a **4-node** (a node with three keys and four children). This 4-node is unstable and must be **split**: the middle key is promoted to the parent, and the remaining two keys become separate 2-nodes.

```
Insert Z into [S X] whose parent is [R]:

  Before:        [R]               After:        [R  X]
                /   \                           /   |   \
              ...   [S X]                     ...  [S]  [Z]

  Step 1: Temporarily form 4-node [S X Z].
  Step 2: Split: middle key X moves to parent [R], forming [R X].
          Left part [S] and right part [Z] become children of [R X].
```

Case 3: Insert into a 3-node leaf whose parent is also a 3-node.

The splitting process may cascade upward. After splitting the leaf's 4-node and promoting the middle key, the parent might itself become a 4-node. If so, split the parent too, promoting its middle key to its grandparent. This cascade continues until we reach a 2-node ancestor (which absorbs the promoted key and becomes a 3-node) or until we reach the root.

Case 4: Splitting the root.

If the cascade reaches the root and the root is a 3-node that has just absorbed a promoted key (making it a temporary 4-node), split the root into three 2-nodes. The middle key becomes the new root, and the tree's height increases by one. This is the **only** way a 2-3 tree grows taller, which is why the tree is always perfectly balanced -- it grows from the top, not from the bottom.

```
Splitting the root:

  Before:       [A  M  Z]          After:        [M]
               / |  |  \                        /    \
                                              [A]    [Z]
                                             /  \   /  \

  Height increases by 1. All paths grow by 1 simultaneously.
```

2.4 Insertion Trace: Building a 2-3 Tree

Let us trace the insertion of the keys S, E, A, R, C, H into an initially empty 2-3 tree.

Insert S: Tree is empty. Create a single 2-node.

```
  [S]
```

Insert E: The root is a 2-node. E < S, so E goes into the root, forming a 3-node.

```
  [E S]
```

Insert A: The root is a 3-node. A < E, so we need to insert A into the root (which is also a leaf). The root becomes a temporary 4-node [A E S]. Split it: E becomes the new root, A and S become its children.

```
    [E]
   /   \
 [A]   [S]
```

Height increased from 0 to 1.

Insert R: Search: R > E, go right to [S]. [S] is a 2-node leaf. R < S, so absorb R to form [R S].

```
    [E]
   /   \
 [A]   [R S]
```

Insert C: Search: C < E, so go left to [A]. [A] is a 2-node leaf. C > A, so absorb C to form [A C].

```
     [E]
    /   \
 [A C]  [R S]
```

Insert H: Search: H > E, go right to [R S]. [R S] is a 3-node leaf. H < R, so we temporarily form [H R S]. Split: R is promoted to parent [E], forming [E R]. H and S become separate 2-node children.

```
      [E R]
     / |   \
  [A C] [H] [S]
```

The final tree has height 1 and contains all six keys with perfect balance.

2.5 Height Analysis of 2-3 Trees

PROPOSITION 2.1. The height of a 2-3 tree with N keys satisfies:

floor(log_3(N+1)) - 1 <= height <= floor(log_2(N+1)) - 1

Proof. The height is maximized when every node is a 2-node (the tree is a complete binary tree). A complete binary tree of height h has at most 2^(h+1) - 1 keys, so N <= 2^(h+1) - 1, giving h >= floor(log_2(N+1)) - 1.

The height is minimized when every node is a 3-node. A complete ternary tree of height h has at most (3^(h+1) - 1)/2 nodes, each holding 2 keys, for a total of 3^(h+1) - 1 keys. So N <= 3^(h+1) - 1, giving h >= floor(log_3(N+1)) - 1.

For N = 1,000,000: the height is between floor(log_3(1000001)) - 1 = 12 and floor(log_2(1000001)) - 1 = 19.

For N = 1,000,000,000 (one billion): the height is between 18 and 29.

This is a remarkably tight range. No matter what sequence of insertions and deletions we perform, the height of a 2-3 tree never exceeds about 20 for a million keys. This is what we mean by a *guaranteed* logarithmic bound.

2.6 Why Not Implement 2-3 Trees Directly?

2-3 trees are conceptually elegant, but implementing them directly is painful. The code must handle multiple node types (2-nodes and 3-nodes), each with different numbers of keys and children. Insertion requires temporary 4-nodes, splitting, and promotion. The case analysis is tedious and error-prone.

Here is a taste of the complexity: a direct implementation needs separate classes or union types for 2-nodes and 3-nodes, with methods to convert between them. The insert method must check the type of each node, handle the temporary 4-node, split it, and promote the middle key -- all while maintaining the parent pointer or using a recursive approach that returns a potentially different node type.

This motivates the central idea of this lecture: representing a 2-3 tree as a **binary search tree** with colored links. We get the guaranteed balance of 2-3 trees with the simple, uniform node structure of BSTs. This representation is the **red-black BST**.

Section 3: Red-Black BST Representation (~20 minutes)

Estimated time: 20 minutes

3.1 Encoding 3-Nodes with Red Links

The key insight is that we can represent a 3-node as two 2-nodes connected by a special "red" link. All other links are "black." This transforms any 2-3 tree into a binary search tree:

DEFINITION 3.1 -- RED-BLACK BST. A *red-black BST* is a BST in which each link (equivalently, each node, since each node has exactly one incoming link) is colored either **red** or **black**, subject to the following constraints:

1. **Red links lean left.** If a node has a red link to a child, it is always the *left* child. (This is the "left-leaning" convention.)
2. **No node has two red links connected to it.** Specifically, no node is the parent of two red children, and no right child is red.
3. **Perfect black balance.** Every path from the root to a null link passes through the same number of black links.

Convention: we color the *node* red or black rather than the link, understanding that a red node means "the link from this node's parent to this node is red." The root is always black (its parent link does not exist, or equivalently, is black). Null links are considered black.

3.2 The 1-1 Correspondence with 2-3 Trees

The correspondence between red-black BSTs and 2-3 trees is exact:

- A **2-node** in the 2-3 tree corresponds to a single black node in the red-black BST.
- A **3-node** with keys J and K (J < K) corresponds to two nodes: K is a black node, and J is its red left child. The left subtree of J is the left child of the 3-node, the right subtree of J is the middle child, and the right subtree of K is the right child.

```
2-3 tree 3-node:              Red-Black BST encoding:

     [J  K]                        K (black)
    / |   \                       / \
   <J  J-K  >K                 J(red) >K
                               / \
                             <J   J-K

Red link connects J to K (J is K's red left child).
```

To visualize this, imagine "flattening" each red link so that the red child sits at the same level as its black parent. If you do this for the entire tree, you recover the original 2-3 tree.

```
Red-Black BST:                    Flatten red links:

        M (black)                       [E   M]
       / \                             / |     \
    E(red) R (black)              [A C] [H]   [R S]
   / \      / \
  ... ...  ... ...

  (When you "flatten" E up to M's level, you get the 3-node [E M].)
```

This correspondence is the reason red-black BSTs work: every structural invariant of 2-3 trees (perfect balance, bounded height) translates directly into properties of the red-black BST.

3.3 Left-Leaning Convention

In Sedgewick's original 1978 paper with Guibas, red links could lean either left or right, which led to a large number of cases in the insertion and deletion code. In 2008, Sedgewick observed that restricting red links to lean left -- the **left-leaning red-black BST** (LLRB) -- dramatically simplifies the algorithms.

The left-leaning convention means:

- A 3-node [J K] is always represented with K as the black parent and J as its red left child. Never the reverse.
- There is exactly one LLRB tree for each 2-3 tree (the correspondence is bijective).

- The code for insertion and deletion has only three fix-up cases, compared to six or more in a general red-black tree.

3.4 Node Structure

Each node in a left-leaning red-black BST stores:

```
Node:
    key         // the key
    value       // the associated value
    left        // left child (may be null)
    right       // right child (may be null)
    color       // RED or BLACK (color of the link from parent to this node)
    size        // number of nodes in the subtree rooted here
```

We define a helper function:

```
function isRed(node):
    if node is null:
        return false       // null links are black
    return node.color == RED
```

The `size` field is maintained in each node (as in Lecture 6's BST) to support rank and select operations in O(lg N) time.

3.5 Invariants Summarized

For reference, here are the complete LLRB invariants. Every operation we implement must preserve all of these:

1. **BST ordering.** Left subtree keys < node key < right subtree keys.
2. **No right-leaning red links.** `isRed(node.right) && !isRed(node.left)` is never true.
3. **No two consecutive left reds.** `isRed(node.left) && isRed(node.left.left)` is never true.
4. **No two red children.** `isRed(node.left) && isRed(node.right)` is never true.
5. **Perfect black balance.** Every root-to-null path has the same number of black links.

6. **Root is black.**

Invariants 2, 3, and 4 together ensure that red links form only valid encodings of 3-nodes (a single left-leaning red link). Invariant 5 is the balance condition inherited from 2-3 trees.

Section 4: Elementary Operations (~25 minutes)

Estimated time: 25 minutes

4.1 Left Rotation

A left rotation is applied when a **right-leaning red link** needs to be fixed. It rotates the right child up to become the parent, making the original parent the new left child. The color of the link between them changes accordingly.

DEFINITION 4.1 -- LEFT ROTATION. Given a node `h` whose right child `x` is red, a *left rotation* rearranges the subtree so that `x` becomes the root of the subtree, `h` becomes the left child of `x` (connected by a red link), and the subtrees are redistributed to maintain BST order.

```
function rotateLeft(h):
    x = h.right                       // x is the red right child
    h.right = x.left                  // x's left subtree becomes h's right subtree
    x.left = h                        // h becomes x's left child
    x.color = h.color                 // x inherits h's color (link from parent)
    h.color = RED                     // link from x to h is now red
    x.size = h.size                   // x takes h's position, same subtree size
    h.size = 1 + size(h.left) + size(h.right)  // update h's size
    return x                          // x is the new root of this subtree
```

Visually:

```
Before rotateLeft(h):                After rotateLeft(h):

       h (any color)                       x (same color as h was)
      / \                                 / \
    a    x (RED)                      h(RED)  c
        / \                           / \
       b   c                         a   b

  Keys: a < h < b < x < c  (preserved)
```

Key observations:

- The BST ordering is preserved: a < h < b < x < c holds in both arrangements.
- The link from the parent to the subtree root keeps the same color (x inherits h's color).
- The new link between x and h is red (h becomes a red left child of x).
- The black height of the subtree is unchanged, because no black link positions change relative to any path.

4.2 Right Rotation

A right rotation is the mirror image of a left rotation. It is applied when two consecutive left red links appear (a left child and a left-left grandchild are both red).

DEFINITION 4.2 -- RIGHT ROTATION. Given a node `h` whose left child `x` is red, a *right rotation* rearranges the subtree so that `x` becomes the root, `h` becomes the right child of `x` (connected by a red link), and subtrees are redistributed.

```
function rotateRight(h):
    x = h.left                        // x is the red left child
    h.left = x.right                  // x's right subtree becomes h's left subtree
    x.right = h                       // h becomes x's right child
    x.color = h.color                 // x inherits h's color
    h.color = RED                     // link from x to h is now red
    x.size = h.size                   // x takes h's position
    h.size = 1 + size(h.left) + size(h.right)  // update h's size
    return x                          // x is the new root of this subtree
```

Visually:

```
Before rotateRight(h):                After rotateRight(h):


         h (any color)                      x (same color as h was)
        / \                                / \
     x(RED) c                             a    h(RED)
    / \                                       / \
   a   b                                     b   c

  Keys: a < x < b < h < c  (preserved)
```

4.3 Color Flip

A color flip is applied when both children of a node are red. This corresponds to splitting a temporary 4-node in a 2-3 tree: the two red links going down are replaced by a single red link going up (to the parent).

DEFINITION 4.3 -- COLOR FLIP. Given a node `h` whose left and right children are both red, a *color flip* recolors the children black and the parent red.

```
function flipColors(h):
    h.color = RED                   // link from parent to h becomes red
    h.left.color = BLACK            // left child link becomes black
    h.right.color = BLACK           // right child link becomes black
```

Visually:

```
Before flipColors(h):                After flipColors(h):

         h (BLACK)                            h (RED)
        / \                                  / \
    a(RED)  b(RED)                       a(BLACK)  b(BLACK)
   / \     / \                           / \       / \
  ...     ...                           ...       ...

  No structural change. Only colors change.
  Corresponds to splitting a 4-node in a 2-3 tree.
```

Key observations:

- The tree structure does not change at all -- only colors.
- In 2-3 tree terms, this splits a temporary 4-node (represented by h with two red children) into two 2-nodes (the children turn black) and promotes the middle key (h turns red, joining its parent's node).
- The black height below h is unchanged: before the flip, a path going through h's left child passes through a black h and a red left child. After the flip, it passes through a red h and a black left child. The number of black links on any root-to-null path through this subtree is unchanged.
- After the flip, h is red, which may create a violation higher in the tree (e.g., h's parent might also be red). This is handled by the recursive fix-up in the insertion algorithm.

4.4 The Three Operations in Concert

The beauty of the LLRB tree is that these three operations -- rotateLeft, rotateRight, flipColors -- are the only building blocks needed for insertion and deletion. After inserting a new key (always as a red node at the bottom of the tree, just as in a regular BST), we apply these operations bottom-up to restore the LLRB invariants:

1. **Right-leaning red link:** Apply `rotateLeft`.
2. **Two consecutive left red links:** Apply `rotateRight`.
3. **Both children red:** Apply `flipColors`.

The order matters: we check and fix these conditions in the order listed above, bottom-up as we return from the recursive insertion. This sequence is sufficient to handle every possible violation.

4.5 Why These Operations Are Correct

Each operation is a **local transformation** that fixes one specific violation while preserving all other invariants:

- `rotateLeft` fixes a right-leaning red link by converting it to a left-leaning one. It preserves BST order and black balance.
- `rotateRight` fixes two consecutive left reds by moving one of them to the right. It preserves BST order and black balance.
- `flipColors` eliminates two red children (a temporary 4-node) by splitting. It preserves BST order and black balance, but may create a new violation higher up (which will be fixed by the parent's fix-up).

PROPOSITION 4.1. After applying the three fix-up steps (rotateLeft, rotateRight, flipColors) in order during the return from a recursive insertion, the resulting tree satisfies all LLRB invariants except possibly at the root, which may be red. Setting the root to black completes the restoration.

Proof sketch. The proof proceeds by case analysis on the possible local configurations after inserting a red node. We show that every configuration is handled by exactly one sequence of the three operations, and that each operation moves the tree closer to a valid LLRB configuration without introducing new violations below the current node. The detailed case analysis is given in Section 5.

Section 5: Insertion (~40 minutes)

Estimated time: 40 minutes

5.1 Insertion Algorithm

Insertion into a left-leaning red-black BST follows the same recursive structure as BST insertion, with three additional fix-up steps applied during the return from recursion.

The new key is always inserted as a **red** node. This corresponds to adding the new key to an existing node in the 2-3 tree (either converting a 2-node to a 3-node, or a 3-node to a temporary 4-node that must be split).

```
function insert(key, value):
    root = insertHelper(root, key, value)
    root.color = BLACK           // root is always black

function insertHelper(h, key, value):
    if h is null:
        return new Node(key, value, RED, 1)   // new red node

    // Standard BST insertion
    if key < h.key:
        h.left = insertHelper(h.left, key, value)
    else if key > h.key:
        h.right = insertHelper(h.right, key, value)
    else:
        h.value = value          // key already exists; update value

    // Fix-up: restore LLRB invariants (applied bottom-up)
    if isRed(h.right) and not isRed(h.left):       // Case 1: right-leaning red
        h = rotateLeft(h)
    if isRed(h.left) and isRed(h.left.left):       // Case 2: two consecutive left reds
        h = rotateRight(h)
    if isRed(h.left) and isRed(h.right):           // Case 3: both children red
        flipColors(h)

    h.size = 1 + size(h.left) + size(h.right)
    return h
```

5.2 The Three Fix-Up Cases

Let us examine each case in detail.

Case 1: Right-leaning red link (rotate left).

This occurs when a new key is inserted as the right child of a 2-node, or after other operations leave a red right child without a red left child.

```
Before (right-leaning red):          After rotateLeft(h):

     h (black)                            x (black)
    / \                                  / \
  (b)  x (RED)                       h(RED) (b)
      / \                            / \
    (b)  (b)                       (b)  (b)

  Now x is the root, h is a red left child. Valid LLRB locally.
```

Case 2: Two consecutive left red links (rotate right).

This occurs when a new key is inserted as the left child of a 3-node (i.e., the left child of a red left child).

```
Before (two left reds):              After rotateRight(h):

       h (black)                          x (black)
      / \                               / \
   x(RED)  (b)                       (b)   h(RED)
  / \                                      / \
a(RED) (b)                              (b)   (b)

  Now x is the root, with a(RED) left child and h(RED) right child.
  Both children are red -- this triggers Case 3 (flipColors).
```

Case 3: Both children red (flip colors).

This occurs when a rotation has placed two red children under a node, or when a new key is inserted between the two keys of a 3-node.

```
Before (both children red):          After flipColors(h):

       h (black)                          h (RED)
      / \                               / \
   a(RED)  b(RED)                    a(BLACK)  b(BLACK)
  / \     / \                        / \       / \
 ...     ...                        ...       ...

  The red h will be handled by h's parent's fix-up (recursion unwinding).
```

5.3 Complete Insertion Trace: S E A R C H X M P L

Let us trace the insertion of the keys S, E, A, R, C, H, X, M, P, L into an initially empty LLRB tree. We show the tree after each insertion and indicate which fix-up operations fire.

We use the notation: **(R)** for a red node and **(B)** for a black node.

Insert S: Create a new red node S. Set root to black.

```
S(B)
```

Insert E: E < S, so insert E as left child of S. E is red. No fix-ups needed (E is a red left child -- valid).

```
  S(B)
 /
E(R)
```

This encodes the 2-3 tree 3-node [E S].

Insert A: A < E, so insert A as left child of E.

After inserting A as a red node:

```
        S(B)
       /
      E(R)
     /
    A(R)
```

Fix-up at E: isRed(E.right)? No. isRed(E.left) and isRed(E.left.left)? A is red but A.left is null. Only one consecutive red here.

Fix-up at S (the parent) after recursive return from left subtree:
- Case 1: isRed(S.right)? No.
- Case 2: isRed(S.left) = true (E is red), isRed(S.left.left) = true (A is red). **Two consecutive left reds! Rotate right at S.**

```
After rotateRight(S):

    E(B)
   / \
  A(R) S(R)
```

- Case 3: isRed(E.left) = true (A is red), isRed(E.right) = true (S is red). **Both children red! Flip colors.**

```
After flipColors(E):

    E(R)
   / \
  A(B) S(B)
```

But E is the root, so we set root.color = BLACK:

```
    E(B)
   / \
  A(B) S(B)
```

This encodes the 2-3 tree with root [E] and children [A] and [S] -- three 2-nodes of height 1.

Insert R: R > E, go right. R < S, so insert R as left child of S. R is red.

```
      E(B)
     / \
   A(B)  S(B)
         /
        R(R)
```

No fix-ups fire at S: R is a red left child, no two consecutive left reds, not both children red. Valid.

This encodes: root [E], left child [A], right child [R S] (3-node).

Insert C: C > A, so insert C as right child of A. C is red.

```
       E(B)
      / \
    A(B)  S(B)
      \   /
   C(R) R(R)
```

Fix-up at A:
- Case 1: isRed(A.right) = true (C is red), and isRed(A.left) = false. **Right-leaning red! Rotate left at A.**

```
After rotateLeft(A):

       E(B)
      / \
    C(B)  S(B)
    /     /
  A(R)  R(R)
```

When we rotateLeft(A), C becomes the new subtree root and inherits A's color (black). A becomes C's red left child.

No further fix-ups at C or E. Valid tree.

This encodes: 2-3 tree root [E], left child [A C] (3-node), right child [R S] (3-node).

Insert H: H > E, go right. H < S, go left (to R). H < R, insert H as left child of R. H is red.

```
        E(B)
       / \
     C(B)  S(B)
     /     /
   A(R)  R(R)
        /
       H(R)
```

Fix-up at R: Case 1: isRed(R.right)? No. Case 2: isRed(R.left) and isRed(R.left.left)? H is red, H.left is null. No consecutive reds at R level.

Fix-up at S (returning from S.left = R subtree):
- Case 1: isRed(S.right)? No. Skip.
- Case 2: isRed(S.left) = true (R is red), isRed(S.left.left) = true (H is red). **Two consecutive left reds! Rotate right at S.**

```
After rotateRight(S):

        E(B)
       / \
     C(B)  R(B)
     /    / \
   A(R) H(R) S(R)
```

R was red, S was black. After rotateRight(S): R becomes the subtree root and inherits S's color (black). S becomes R's red right child.

- Case 3 at R: isRed(R.left) = true (H is red), isRed(R.right) = true (S is red). **Both children red! Flip colors.**

```
After flipColors(R):

        E(B)
       / \
     C(B)  R(R)
     /    / \
   A(R) H(B) S(B)
```

Fix-up at E (returning from E.right = R subtree):
- Case 1: isRed(E.right) = true (R is red), isRed(E.left) = false (C is black). **Right-leaning red! Rotate left at E.**

```
After rotateLeft(E):

        R(B)
       / \
     E(R)  S(B)
    / \
  C(B) H(B)
  /
A(R)
```

- Case 2 at R: isRed(R.left) = true (E is red), isRed(R.left.left)? E.left = C which is black. Not two consecutive reds. Skip.
- Case 3 at R: isRed(R.left) = true (E is red), isRed(R.right) = false (S is black). Not both children red. Skip.

Set root to black (already black).

```
        R(B)
       / \
     E(R)  S(B)
    / \
  C(B) H(B)
  /
A(R)
```

This encodes the 2-3 tree: root [E R] (3-node), children [A C] (3-node), [H] (2-node), [S] (2-node).

Insert X: X > R, go right. X > S, insert X as right child of S. X is red.

```
        R(B)
       / \
     E(R)  S(B)
    / \      \
  C(B) H(B)  X(R)
  /
A(R)
```

Fix-up at S:
- Case 1: isRed(S.right) = true (X is red), isRed(S.left) = false. **Right-leaning red! Rotate left at S.**

```
After rotateLeft(S):

        R(B)
       / \
     E(R)  X(B)
    / \    /
  C(B) H(B) S(R)
  /
A(R)
```

No further violations at R. Done.

Insert M: M < R. Go left to E. M > E, go right to H. M > H, insert M as right child of H. M is red.

```
          R(B)
         / \
       E(R)  X(B)
      / \    /
    C(B) H(B) S(R)
    /      \
  A(R)     M(R)
```

Fix-up at H:

- Case 1: isRed(H.right) = true (M is red), isRed(H.left) = false. **Right-leaning red! Rotate left at H.**

```
After rotateLeft(H):

          R(B)
         / \
       E(R)  X(B)
      / \    /
    C(B) M(B) S(R)
    /   /
  A(R) H(R)
```

No further violations propagate upward. Done.

Insert P: P > R. Go right to X. P < X, go left to S. P < S, insert P as left child of S. P is red.

```
          R(B)
         / \
       E(R)  X(B)
      / \    /
    C(B) M(B) S(R)
    /   /    /
  A(R) H(R) P(R)
```

Fix-up at S: No cases fire (P is red left child, only one red).

Fix-up at X:

- Case 1: isRed(X.right)? No. Skip.

- Case 2: isRed(X.left) = true (S is red), isRed(X.left.left) = true (P is red). **Two consecutive left reds! Rotate right at X.**

```
After rotateRight(X):

          R(B)
         / \
       E(R)  S(B)
      / \   / \
    C(B) M(B) P(R) X(R)
    /   /
  A(R) H(R)
```

- Case 3 at S: isRed(S.left) = true (P is red), isRed(S.right) = true (X is red). **Both children red! Flip colors.**

```
After flipColors(S):

          R(B)
         / \
       E(R)  S(R)
      / \   / \
    C(B) M(B) P(B) X(B)
    /   /
  A(R) H(R)
```

Fix-up at R:

- Case 1: isRed(R.right) = true (S is red), isRed(R.left) = true (E is red). Both are red, so the condition (right red AND left NOT red) is false. Skip Case 1.

- Case 2: isRed(R.left) = true (E is red), isRed(R.left.left) = false (C is black). Skip.

- Case 3: isRed(R.left) = true (E is red), isRed(R.right) = true (S is red). **Both children red! Flip colors.**

```
After flipColors(R):

          R(R)
         / \
       E(B)  S(B)
      / \   / \
    C(B) M(B) P(B) X(B)
    /   /
  A(R) H(R)
```

Set root to black:

```
          R(B)
         / \
       E(B)  S(B)
      / \   / \
    C(B) M(B) P(B) X(B)
    /   /
  A(R) H(R)
```

Insert L: L < R, go left to E. L > E, go right to M. L < M, go left to H. L > H, insert L as right child of H. L is red.

```
           R(B)
          / \
        E(B)  S(B)
       / \   / \
     C(B) M(B) P(B) X(B)
     /   /
   A(R) H(R)
          \
          L(R)
```

Fix-up at H:
- Case 1: isRed(H.right) = true (L is red), isRed(H.left) = false. **Right-leaning red! Rotate left at H.**

```
After rotateLeft(H):

           R(B)
          / \
        E(B)  S(B)
       / \   / \
     C(B) M(B) P(B) X(B)
     /   /
   A(R) L(R)
        /
      H(R)
```

Fix-up at M:

- Case 2: isRed(M.left) = true (L is red), isRed(M.left.left) = true (H is red). **Two consecutive left reds! Rotate right at M.**

```
After rotateRight(M):

           R(B)
          / \
        E(B)  S(B)
       / \   / \
     C(B) L(B) P(B) X(B)
     /   / \
   A(R) H(R) M(R)
```

- Case 3 at L: isRed(L.left) = true (H is red), isRed(L.right) = true (M is red). **Both children red! Flip colors.**

```
After flipColors(L):

           R(B)
          / \
        E(B)  S(B)
       / \   / \
     C(B) L(R) P(B) X(B)
     /   / \
   A(R) H(B) M(B)
```

Fix-up at E:

- Case 1: isRed(E.right) = true (L is red), isRed(E.left) = false (C is black). **Right-leaning red! Rotate left at E.**

```
After rotateLeft(E):

            R(B)
           / \
         L(B)  S(B)
        / \   / \
      E(R) M(B) P(B) X(B)
     / \
   C(B) H(B)
   /
  A(R)
```

No further fix-ups at R: L is a black left child. Done.

Final tree after inserting S, E, A, R, C, H, X, M, P, L:

```
              R(B)
             / \
           L(B)  S(B)
          / \   / \
        E(R) M(B) P(B) X(B)
       / \
     C(B) H(B)
     /
   A(R)
```

Let us verify the LLRB invariants:

1. **BST order:** In-order traversal gives A, C, E, H, L, M, P, R, S, X. Correct.
2. **No right-leaning red links:** The only red nodes are E (left child of L) and A (left child of C). Both lean left. Valid.
3. **No two consecutive left reds:** E is red, E.left = C is black. A is red, A.left is null. Valid.
4. **No two red children:** No node has both children red. Valid.
5. **Perfect black balance:** Let us count black links on several root-to-null paths:
6. R -> L -> E -> C -> A -> null: black nodes = R(B), L(B), C(B) = 3. (E is red, A is red.)
7. R -> L -> E -> H -> null: black nodes = R(B), L(B), H(B) = 3. (E is red.)
8. R -> L -> M -> null: black nodes = R(B), L(B), M(B) = 3.
9. R -> S -> P -> null: black nodes = R(B), S(B), P(B) = 3.
10. R -> S -> X -> null: black nodes = R(B), S(B), X(B) = 3.
    All paths have black height 3. Valid.
11. **Root is black:** R is black. Valid.

All invariants hold. The tree height is 4 (longest path: R -> L -> E -> C -> A), and 2 lg(10+1) = 2 * 3.46 = 6.92, so height 4 <= 6.92. The bound holds.

5.4 Analysis of Insertion

PROPOSITION 5.1. Insertion into an LLRB tree with N keys requires at most 2 lg N comparisons and at most 2 lg N color changes and rotations.

Proof. The insertion proceeds by searching down the tree (at most height comparisons) and then fixing up on the way back (at most one rotation and/or color flip per level). Since the height is at most 2 lg(N+1) (Proposition 7.1, proven in Section 7), the total cost is O(lg N). More precisely, each level of the fix-up does at most one rotation and one color flip, and there are at most 2 lg(N+1) levels.

In practice, most insertions require very few fix-ups. The rotations and color flips are concentrated near the bottom of the tree, and the cascading effect (a flipColors promoting a red link upward) rarely propagates more than one or two levels.

Section 6: Deletion (~30 minutes)

Estimated time: 30 minutes

6.1 Overview of Deletion

Deletion in red-black BSTs is the most complex operation. The fundamental challenge is: if we delete a black node (or a node reached through a black link), we disturb the black balance. To avoid this, the strategy is to **ensure that the node we delete is red**. If it is not red, we transform the tree to make it red before deleting it.

The Sedgewick LLRB deletion algorithm uses two key helper operations: `moveRedLeft` and `moveRedRight`, which ensure that as we descend toward the key to delete, the current node is always red. This guarantees that the eventual deletion does not break black balance.

6.2 Delete Minimum

The simplest deletion case is deleting the minimum key (the leftmost node). This forms the basis for general deletion (where we replace the node to delete with its in-order successor, then delete the successor using deleteMin on the right subtree).

The invariant maintained while descending is: the current node or its left child is red. This ensures that when we reach the minimum (a node with no left child), it is red, and we can simply remove it.

```
function deleteMin():
    if root is null:
        return                    // empty tree
    // If both children of root are black, set root to red
    // (to maintain the invariant as we descend)
    if not isRed(root.left) and not isRed(root.right):
        root.color = RED
    root = deleteMinHelper(root)
    if root is not null:
        root.color = BLACK

function deleteMinHelper(h):
    if h.left is null:
        return null               // h is the minimum; delete it

    // If h.left and h.left.left are both black, we need to
    // borrow from the right to make h.left (or a child) red.
    if not isRed(h.left) and not isRed(h.left.left):
        h = moveRedLeft(h)

    h.left = deleteMinHelper(h.left)
    return balance(h)             // fix up on the way back
```

6.3 Move Red Left

The `moveRedLeft` operation is used when we are descending to the left and the left child and left-left grandchild are both black. We need to "borrow" a red link from the right side.

```
function moveRedLeft(h):
    // Precondition: h is red, h.left and h.left.left are both black.
    // Postcondition: h.left or one of its children is red.
    flipColors(h)                 // make h black, both children red
    if isRed(h.right.left):       // right child's left child is red
        h.right = rotateRight(h.right)  // rotate to move red to the right
        h = rotateLeft(h)               // rotate left to balance
        flipColors(h)                   // restore colors
    return h
```

Visually, the simplest case (when h.right.left is not red):

```
Before moveRedLeft(h):            After flipColors(h):

     h(RED)                           h(BLACK)
    / \                              / \
  a(B) b(B)                        a(RED) b(RED)
  / \                               / \
(B) (B)                           (B) (B)

  Now a is red. Invariant restored for descending left.
```

When h.right.left IS red, the additional rotations redistribute the red link more carefully to maintain LLRB invariants.

6.4 Move Red Right

The mirror operation, used when descending to the right:

```
function moveRedRight(h):
    // Precondition: h is red, h.right and h.right.left are both black.
    // Postcondition: h.right or one of its children is red.
    flipColors(h)
    if isRed(h.left.left):        // left child's left child is red
        h = rotateRight(h)
        flipColors(h)
    return h
```

6.5 Balance (Fix-Up)

After deleting a node, we need to fix up the tree on the way back up. The `balance` function applies the same three cases as insertion:

```
function balance(h):
    if isRed(h.right) and not isRed(h.left):
        h = rotateLeft(h)
    if isRed(h.left) and isRed(h.left.left):
        h = rotateRight(h)
    if isRed(h.left) and isRed(h.right):
        flipColors(h)
    h.size = 1 + size(h.left) + size(h.right)
    return h
```

This is identical to the fix-up code in `insertHelper`. The balance function restores all LLRB invariants after any local modification.

6.6 General Deletion

General deletion combines the strategies of deleteMin and deleteMax. To delete an arbitrary key:

1. **Search** for the key, maintaining the invariant that the current node or one of its children is red.
2. If the key is found in an internal node, replace it with its in-order successor (the minimum of the right subtree), then delete that successor.
3. If the key is found at a leaf, simply remove it (it must be red, by our invariant).

```
function delete(key):
    if not contains(key):
        return                     // key not in tree
    if not isRed(root.left) and not isRed(root.right):
        root.color = RED
    root = deleteHelper(root, key)
    if root is not null:
        root.color = BLACK

function deleteHelper(h, key):
    if key < h.key:
        // Going left
        if not isRed(h.left) and not isRed(h.left.left):
            h = moveRedLeft(h)
        h.left = deleteHelper(h.left, key)
    else:
        // key >= h.key: might go right or delete here
        if isRed(h.left):
            h = rotateRight(h)    // lean right to push red right
        if key == h.key and h.right is null:
            return null            // found at bottom: delete
        if not isRed(h.right) and not isRed(h.right.left):
            h = moveRedRight(h)
        if key == h.key:
            // Replace with successor
            successor = minNode(h.right)
            h.key = successor.key
            h.value = successor.value
            h.right = deleteMinHelper(h.right)
        else:
            h.right = deleteHelper(h.right, key)
    return balance(h)
```

6.7 Deletion Trace: Delete Minimum from a Sample Tree

Let us trace deleting the minimum key from the following valid LLRB tree:

```
      H(B)
     / \
   C(R)  S(B)
  / \    /
 A(B) E(B) R(R)
```

Verification of validity: Black heights -- H(B)->C(R)->A(B)->null: H, A = 2. H(B)->C(R)->E(B)->null: H, E = 2. H(B)->S(B)->R(R)->null: H, S = 2. All equal. Red links: C is red left of H (ok), R is red left of S (ok). No consecutive reds. Valid.

The minimum key is A.

Step 1: isRed(root.left)? C is red. isRed(root.right)? S is black. Not both black, so do NOT set root red. Call deleteMinHelper(H).

Step 2: deleteMinHelper(H). H.left = C, not null. isRed(H.left)? C is red. So the condition `not isRed(H.left)` is false, meaning we do NOT call moveRedLeft. Recurse: H.left = deleteMinHelper(C).

Step 3: deleteMinHelper(C). C.left = A, not null. isRed(C.left)? A is black. isRed(C.left.left)? A.left is null (black). Both black! But we also need C to be red for moveRedLeft's precondition. C is red (confirmed from tree). Call moveRedLeft(C).

moveRedLeft(C): flipColors(C). C becomes black, A becomes red, E becomes red.

```
    C(B)
   / \
 A(R) E(R)
```

Check isRed(C.right.left)? C.right = E, E.left is null (black). No special case.

Continue: C.left = deleteMinHelper(A).

Step 4: deleteMinHelper(A). A.left is null. Return null. (A is deleted.)

Step 5: Back at C. C.left = null. Call balance(C).

```
  C(B)
    \
    E(R)
```

Balance(C): isRed(C.right) and not isRed(C.left)? E is red, C.left is null (black). Yes! rotateLeft(C).

```
  E(B)
  /
C(R)
```

E inherits C's color (black). C becomes red. No further balance issues.

Step 6: Back at H. H.left = E(B).

```
       H(?)
      / \
    E(B)  S(B)
   /      /
  C(R)  R(R)
```

Call balance(H). H's color: H was originally black, and nothing changed it. isRed(H.right)? S is black. isRed(H.left)? E is black. isRed(H.left) and isRed(H.right)? No. Done.

Step 7: root = H. root is not null, set root.color = BLACK (already black).

```
       H(B)
      / \
    E(B)  S(B)
   /      /
  C(R)  R(R)
```

Verification: Black heights: H->E->C(R)->null: H, E = 2. H->E->[null right]: H, E = 2. H->S->R(R)->null: H, S = 2. H->S->[null right]: H, S = 2. All equal. Valid LLRB.

The minimum (A) has been successfully deleted.

6.8 Complexity of Deletion

PROPOSITION 6.1 (Deletion Cost). Deletion from an LLRB tree with N keys requires at most 2 lg N comparisons, plus at most a constant number of rotations and color flips per level, for a total of O(lg N) work.

Proof. The descent to find the key (or its successor) follows a root-to-leaf path of length at most 2 lg(N+1). At each level, we do at most one moveRedLeft or moveRedRight (each involving a constant number of rotations and flips) on the way down, and one balance call (constant work) on the way back up. Total: O(lg N).

Section 7: Geometric Properties and Height Analysis (~20 minutes)

Estimated time: 20 minutes

7.1 Height Bound

PROPOSITION 7.1. The height of a left-leaning red-black BST with N keys is at most 2 lg(N+1).

Proof. Consider any root-to-null path. Let B be the number of black links on this path (the *black height*). Since the tree has perfect black balance, B is the same for every root-to-null path.

A path can have at most B red links, because red links cannot be consecutive (no two consecutive red links on any path, since a red right link is impossible and two consecutive left reds are forbidden). Therefore, the total path length is at most 2B.

Now consider the black links alone. If we "collapse" all red links (merging each red node with its parent), we obtain a 2-3 tree. A 2-3 tree of black height B has at least 2^B - 1 keys (when every node is a 2-node) and at most 3^B - 1 keys (when every node is a 3-node). Therefore N >= 2^B - 1, giving B <= lg(N+1).

The height of the red-black tree is at most 2B <= 2 lg(N+1).

PROPOSITION 7.2. The black height of a left-leaning red-black BST with N keys is between ceil(lg(N+1)) and floor(log_3(N+1)) + 1.

Proof. This follows directly from the 2-3 tree correspondence. A 2-3 tree of height h (which equals the black height B) has between 2^h - 1 and 3^h - 1 keys. So 2^B - 1 <= N <= 3^B - 1, giving the bounds.

7.2 Comparison with AVL Trees

AVL trees maintain the invariant that the heights of the left and right subtrees of every node differ by at most 1. The height of an AVL tree with N keys is at most approximately 1.44 lg(N+1), which is tighter than the 2 lg(N+1) bound for red-black trees.

However, AVL trees require more rotations during insertion and deletion (up to O(lg N) rotations per operation, compared to at most 2 rotations for insertion in a red-black tree). The extra rotations increase the constant factor in practice, and red-black trees are generally preferred in standard library implementations because:

1. **Simpler implementation.** The LLRB insertion code is remarkably concise.
2. **Fewer rotations per insert.** At most 2 rotations per insertion (versus up to lg N for AVL).
3. **Good enough height bound.** The factor of 2 in the height bound rarely matters in practice; most red-black trees have height close to lg N, not 2 lg N.

Property	Red-Black BST	AVL Tree
Height bound	<= 2 lg(N+1)	<= 1.44 lg(N+1)
Rotations per insert	<= 2	<= O(lg N)
Rotations per delete	<= 3	<= O(lg N)
Practical height	~lg N	~lg N
Implementation complexity	Moderate	Moderate to High
Standard library use	Java TreeMap, C++ map	Some databases

7.3 Comparison with B-Trees

B-trees generalize 2-3 trees by allowing up to M keys per node (for a parameter M, typically in the hundreds or thousands). They are the primary data structure for database indexing and file system metadata.

A B-tree of order M has height at most log_{M/2}(N), which for M = 1000 and N = 10^9 is about 3. The extreme branching factor makes B-trees ideal for disk-based storage, where each node read requires a slow disk access.

Red-black BSTs are a special case of B-trees with M = 4 (since they correspond to 2-3-4 trees, or in the LLRB variant, 2-3 trees). The choice between them depends on the storage medium:

- **In-memory data:** Red-black BSTs (simple, cache-friendly per node).
- **Disk-based data:** B-trees (minimize disk reads with high branching factor).

7.4 Perfect Black Balance

One of the most elegant properties of red-black BSTs is **perfect black balance**: every root-to-null path has the same number of black links. This is directly inherited from the perfect balance of the corresponding 2-3 tree.

DEFINITION 7.1 -- BLACK HEIGHT. The *black height* of a red-black BST is the number of black links on any root-to-null path (they are all equal).

This property can be verified algorithmically:

```
function blackHeight(node):
    if node is null:
        return 0
    leftBH = blackHeight(node.left)
    rightBH = blackHeight(node.right)
    if leftBH != rightBH:
        error "Black balance violated!"
    return leftBH + (1 if node.color == BLACK else 0)
```

The black height also tells us the height of the corresponding 2-3 tree, which is always between floor(lg(N+1)) and ceil(log_3(N+1)).

7.5 Average Case Performance

In practice, red-black BSTs perform significantly better than the worst-case bounds suggest. Empirical studies show:

- The average height of a red-black tree built from N random insertions is approximately lg N + 0.25 (very close to the information-theoretic minimum of lg N).
- The average number of rotations per insertion is less than 1.
- The average number of color flips per insertion is about 0.3 to 0.5.

This near-optimal average-case behavior, combined with the guaranteed worst-case bound, makes red-black BSTs an excellent all-around choice for ordered symbol tables.

Section 8: Complete Implementation (~25 minutes)

Estimated time: 25 minutes

8.1 Full LLRB Tree Pseudocode

Below is the complete implementation of a left-leaning red-black BST supporting all ordered symbol table operations. The code is language-agnostic pseudocode that can be translated to any programming language.

```
constants:
    RED   = true
    BLACK = false

class Node:
    key       // Comparable key
    value     // Associated value
    left      // Left child (Node or null)
    right     // Right child (Node or null)
    color     // RED or BLACK
    size      // Number of nodes in subtree

    constructor(key, value, color, size):
        this.key = key
        this.value = value
        this.color = color
        this.size = size
        this.left = null
        this.right = null

class LLRBTree:
    root = null

    //------------------------------------------------------------
    // Helper functions
    //------------------------------------------------------------

    function isRed(node):
        if node is null: return false
        return node.color == RED

    function size(node):
        if node is null: return 0
        return node.size

    function size():
        return size(root)

    function isEmpty():
```

```
    return root is null

//------------------------------------------------------------
// Rotations and color flip
//------------------------------------------------------------

function rotateLeft(h):
    x = h.right
    h.right = x.left
    x.left = h
    x.color = h.color
    h.color = RED
    x.size = h.size
    h.size = 1 + size(h.left) + size(h.right)
    return x

function rotateRight(h):
    x = h.left
    h.left = x.right
    x.right = h
    x.color = h.color
    h.color = RED
    x.size = h.size
    h.size = 1 + size(h.left) + size(h.right)
    return x

function flipColors(h):
    h.color = not h.color
    h.left.color = not h.left.color
    h.right.color = not h.right.color

//------------------------------------------------------------
// Search
//------------------------------------------------------------

function get(key):
    return getHelper(root, key)

function getHelper(node, key):
    if node is null: return null
    if key < node.key:
        return getHelper(node.left, key)
    else if key > node.key:
        return getHelper(node.right, key)
    else:
        return node.value

function contains(key):
    return get(key) is not null

//------------------------------------------------------------
// Insertion
//------------------------------------------------------------

function put(key, value):
    root = putHelper(root, key, value)
    root.color = BLACK

function putHelper(h, key, value):
    if h is null:
        return new Node(key, value, RED, 1)

    if key < h.key:
        h.left = putHelper(h.left, key, value)
    else if key > h.key:
        h.right = putHelper(h.right, key, value)
    else:
        h.value = value

    // Fix-up
    if isRed(h.right) and not isRed(h.left):
        h = rotateLeft(h)
    if isRed(h.left) and isRed(h.left.left):
        h = rotateRight(h)
    if isRed(h.left) and isRed(h.right):
        flipColors(h)

    h.size = 1 + size(h.left) + size(h.right)
    return h
```

```
//------------------------------------------------------------
// Min, Max, Floor, Ceiling
//------------------------------------------------------------

function min():
    if isEmpty(): error "Empty tree"
    return minNode(root).key

function minNode(node):
    if node.left is null: return node
    return minNode(node.left)

function max():
    if isEmpty(): error "Empty tree"
    return maxNode(root).key

function maxNode(node):
    if node.right is null: return node
    return maxNode(node.right)

function floor(key):
    node = floorHelper(root, key)
    if node is null: return null
    return node.key

function floorHelper(node, key):
    if node is null: return null
    if key == node.key: return node
    if key < node.key:
        return floorHelper(node.left, key)
    // key > node.key: floor might be in right subtree
    t = floorHelper(node.right, key)
    if t is not null: return t
    return node

function ceiling(key):
    node = ceilingHelper(root, key)
    if node is null: return null
    return node.key
```

```
function ceilingHelper(node, key):
    if node is null: return null
    if key == node.key: return node
    if key > node.key:
        return ceilingHelper(node.right, key)
    t = ceilingHelper(node.left, key)
    if t is not null: return t
    return node

//------------------------------------------------------------
// Rank and Select
//------------------------------------------------------------

function rank(key):
    return rankHelper(root, key)

function rankHelper(node, key):
    if node is null: return 0
    if key < node.key:
        return rankHelper(node.left, key)
    else if key > node.key:
        return 1 + size(node.left) + rankHelper(node.right, key)
    else:
        return size(node.left)

function select(k):
    // Return the key of rank k (0-indexed)
    if k < 0 or k >= size(): error "Invalid rank"
    return selectHelper(root, k).key

function selectHelper(node, k):
    leftSize = size(node.left)
    if k < leftSize:
        return selectHelper(node.left, k)
    else if k > leftSize:
        return selectHelper(node.right, k - leftSize - 1)
    else:
        return node

//------------------------------------------------------------
```

```
// Deletion
//-----------------------------------------------------------

function deleteMin():
    if isEmpty(): error "Empty tree"
    if not isRed(root.left) and not isRed(root.right):
        root.color = RED
    root = deleteMinHelper(root)
    if not isEmpty():
        root.color = BLACK

function deleteMinHelper(h):
    if h.left is null:
        return null
    if not isRed(h.left) and not isRed(h.left.left):
        h = moveRedLeft(h)
    h.left = deleteMinHelper(h.left)
    return balance(h)

function deleteMax():
    if isEmpty(): error "Empty tree"
    if not isRed(root.left) and not isRed(root.right):
        root.color = RED
    root = deleteMaxHelper(root)
    if not isEmpty():
        root.color = BLACK

function deleteMaxHelper(h):
    if isRed(h.left):
        h = rotateRight(h)
    if h.right is null:
        return null
    if not isRed(h.right) and not isRed(h.right.left):
        h = moveRedRight(h)
    h.right = deleteMaxHelper(h.right)
    return balance(h)

function delete(key):
    if not contains(key): return
    if not isRed(root.left) and not isRed(root.right):
        root.color = RED
    root = deleteHelper(root, key)
    if not isEmpty():
        root.color = BLACK

function deleteHelper(h, key):
    if key < h.key:
        if not isRed(h.left) and not isRed(h.left.left):
            h = moveRedLeft(h)
        h.left = deleteHelper(h.left, key)
    else:
        if isRed(h.left):
            h = rotateRight(h)
        if key == h.key and h.right is null:
            return null
        if not isRed(h.right) and not isRed(h.right.left):
            h = moveRedRight(h)
        if key == h.key:
            x = minNode(h.right)
            h.key = x.key
            h.value = x.value
            h.right = deleteMinHelper(h.right)
        else:
            h.right = deleteHelper(h.right, key)
    return balance(h)

//------------------------------------------------------------
// Move-red helpers
//------------------------------------------------------------

function moveRedLeft(h):
    flipColors(h)
    if isRed(h.right.left):
        h.right = rotateRight(h.right)
        h = rotateLeft(h)
        flipColors(h)
    return h

function moveRedRight(h):
    flipColors(h)
    if isRed(h.left.left):
        h = rotateRight(h)
        flipColors(h)
    return h

//------------------------------------------------------------
// Balance (fix-up)
//------------------------------------------------------------

function balance(h):
    if isRed(h.right) and not isRed(h.left):
        h = rotateLeft(h)
    if isRed(h.left) and isRed(h.left.left):
        h = rotateRight(h)
    if isRed(h.left) and isRed(h.right):
        flipColors(h)
    h.size = 1 + size(h.left) + size(h.right)
    return h

//------------------------------------------------------------
// Ordered iteration
//------------------------------------------------------------

function keys():
    return keys(min(), max())

function keys(lo, hi):
    queue = new Queue()
    keysHelper(root, queue, lo, hi)
    return queue

function keysHelper(node, queue, lo, hi):
    if node is null: return
    if lo < node.key:
        keysHelper(node.left, queue, lo, hi)
    if lo <= node.key and node.key <= hi:
        queue.enqueue(node.key)
    if node.key < hi:
        keysHelper(node.right, queue, lo, hi)
//-----------------------------------------------------------
// Tree height (for testing/verification)
//-----------------------------------------------------------

function height():
    return heightHelper(root)

function heightHelper(node):
    if node is null: return -1
    return 1 + max(heightHelper(node.left), heightHelper(node.right))

function blackHeight():
    return blackHeightHelper(root)

function blackHeightHelper(node):
    if node is null: return 0
    leftBH = blackHeightHelper(node.left)
    rightBH = blackHeightHelper(node.right)
    assert leftBH == rightBH, "Black balance violated"
    return leftBH + (1 if node.color == BLACK else 0)
```

8.2 Correctness Verification

To verify that the implementation is correct, we can add a method that checks all LLRB invariants:

```
function isValidLLRB():
    if not isBST(root, null, null): return false
    if not isBalanced(): return false
    if not is23(root): return false
    if isRed(root): return false
    return true

function isBST(node, minKey, maxKey):
    if node is null: return true
    if minKey is not null and node.key <= minKey: return false
    if maxKey is not null and node.key >= maxKey: return false
    return isBST(node.left, minKey, node.key)
       and isBST(node.right, node.key, maxKey)

function is23(node):
    // No right-leaning red links; no two consecutive left reds
    if node is null: return true
    if isRed(node.right): return false
    if node != root and isRed(node) and isRed(node.left): return false
    return is23(node.left) and is23(node.right)

function isBalanced():
    // Check perfect black balance
    blackCount = 0
    node = root
    while node is not null:
        if not isRed(node): blackCount = blackCount + 1
        node = node.left
    return isBalancedHelper(root, blackCount)

function isBalancedHelper(node, blackCount):
    if node is null: return blackCount == 0
    if not isRed(node): blackCount = blackCount - 1
    return isBalancedHelper(node.left, blackCount)
       and isBalancedHelper(node.right, blackCount)
```

8.3 Performance Characteristics

The following table summarizes the performance of the LLRB tree implementation:

Operation	Worst Case	Amortized
Search	2 lg N	1.00 lg N*
Insert	2 lg N	1.00 lg N*
Delete	2 lg N	1.00 lg N*
Min/Max	2 lg N	--
Floor/Ceiling	2 lg N	--
Rank	2 lg N	--
Select	2 lg N	--
Range count	2 lg N	--
Range search	2 lg N + R	--

(*Amortized costs are empirical averages for random inputs. R is the number of keys in the range.)

All operations are guaranteed O(lg N) in the worst case. The constant factor of 2 in the height bound means that in the very worst case, an LLRB tree might be twice as deep as a perfectly balanced binary tree, but this extreme rarely occurs in practice.

Section 9: Applications and Practical Considerations (~15 minutes)

Estimated time: 15 minutes

9.1 Standard Library Implementations

Red-black trees (or close variants) are the backbone of ordered collections in most major programming languages:

- **Java:** `java.util.TreeMap` and `java.util.TreeSet` use a red-black tree. The implementation is based on Cormen et al. (CLRS), not the left-leaning variant, but the performance characteristics are the same.
- **C++:** `std::map`, `std::set`, `std::multimap`, and `std::multiset` in the Standard Template Library (STL) are typically implemented as red-black trees.
- **C#:** `SortedDictionary<K,V>` and `SortedSet<T>` use red-black trees.
- **Python:** The `sortedcontainers` library uses a different approach (sorted lists with B-tree-like structure), but the `bintrees` library provides red-black tree implementations.

9.2 When to Use Red-Black BSTs

Red-black BSTs are the right choice when you need:

1. **Guaranteed O(lg N) worst-case** for all operations.
2. **Ordered operations** like floor, ceiling, rank, select, and range queries.
3. **In-order traversal** of keys.

If you only need search and insert (no ordering), a **hash table** is usually faster (O(1) amortized). If you need to store data on disk, a **B-tree** is more appropriate (fewer disk accesses due to high branching factor).

9.3 Red-Black BSTs vs. Hash Tables

This is one of the most common design decisions in practice:

Feature	Red-Black BST	Hash Table
Search (average)	lg N	1
Insert (average)	lg N	1
Delete (average)	lg N	1
Search (worst)	2 lg N	N
Ordered operations	Yes	No
Min/Max	O(lg N)	O(N)
Range queries	O(lg N + R)	O(N)
Keys must be	Comparable	Hashable
Memory overhead	3 pointers + color per node	Array + linked lists/probing

Rule of thumb: Use a hash table for fast lookup by key. Use a red-black BST when you need ordered operations or worst-case guarantees.

9.4 The Importance of Balance

The performance difference between balanced and unbalanced trees is dramatic in practice. Consider a symbol table with N = 1,000,000 keys:

Tree Type	Expected Height	Worst-Case Height
Random BST	~28	1,000,000
Red-Black BST	~20	~40
AVL Tree	~20	~29
2-3 Tree	~13-20	~20

A search in a degenerate BST (height 1,000,000) would take a million comparisons -- essentially a linear scan. The same search in a red-black BST would take at most 40 comparisons. This is the difference between a sub-microsecond operation and one that takes milliseconds -- a factor of 1000 in performance.

For interactive applications (databases, web servers, real-time systems), this difference between guaranteed and probabilistic performance can be the difference between acceptable and unacceptable latency.

Section 10: Practice Problems

Problem 1: Insertion Trace

Problem: Starting from an empty LLRB tree, insert the keys D, B, F, A, C, E, G in that order. Show the tree after each insertion, including node colors and any rotations/flips performed.

Solution:

Insert D: New red node, set root to black.

```
D(B)
```

Insert B: B < D, insert as left child. Red left child is valid.

```
  D(B)
 /
B(R)
```

Insert F: F > D, insert as right child of D. F is red.

```
  D(B)
 / \
B(R) F(R)
```

Fix-up at D: isRed(D.right) and not isRed(D.left)? F is red but B is also red, so Case 1 does not fire. Case 2: isRed(D.left) and isRed(D.left.left)? B is red, B.left is null. No. Case 3: isRed(D.left) and isRed(D.right)? Both red! Flip colors.

```
After flipColors(D):


    D(R)
   / \
  B(B) F(B)
```

Set root to black:

```
    D(B)
   / \
  B(B) F(B)
```

Insert A: A < D, go left. A < B, insert as left child of B. A is red.

```
      D(B)
     / \
    B(B) F(B)
   /
  A(R)
```

No fix-ups needed (A is a valid red left child of black B).

Insert C: C < D, go left. C > B, insert as right child of B. C is red.

```
      D(B)
     / \
    B(B) F(B)
   / \
  A(R) C(R)
```

Fix-up at B: Case 3: isRed(B.left) and isRed(B.right)? Both red! Flip colors.

```
After flipColors(B):

      D(B)
     / \
    B(R) F(B)
   / \
  A(B) C(B)
```

Fix-up at D: Case 1: isRed(D.right) and not isRed(D.left)? F is black. No. Case 2: isRed(D.left) and isRed(D.left.left)? B is red, B.left = A is black. No. Case 3: B red, F black. No. Done.

```
      D(B)
     / \
    B(R) F(B)
   / \
  A(B) C(B)
```

Insert E: E > D, go right. E < F, insert as left child of F. E is red.

```
       D(B)
      / \
    B(R) F(B)
   / \   /
  A(B) C(B) E(R)
```

No fix-ups at F needed (E is a valid red left child of black F).

Insert G: G > D, go right. G > F, insert as right child of F. G is red.

```
       D(B)
      / \
    B(R)  F(B)
   / \   / \
  A(B) C(B) E(R) G(R)
```

Fix-up at F: Case 3: isRed(F.left) and isRed(F.right)? Both red! Flip colors.

```
After flipColors(F):

       D(B)
      / \
    B(R)  F(R)
   / \   / \
  A(B) C(B) E(B) G(B)
```

Fix-up at D: Case 1: isRed(D.right) and not isRed(D.left)? F is red and B is red, so condition fails. Case 2: isRed(D.left) and isRed(D.left.left)? B is red, B.left = A is black. No. Case 3: isRed(D.left) and isRed(D.right)? Both red! Flip colors.

```
After flipColors(D):

       D(R)
      / \
    B(B)  F(B)
   / \   / \
  A(B) C(B) E(B) G(B)
```

Set root to black:

```
       D(B)
      / \
    B(B)  F(B)
   / \   / \
  A(B) C(B) E(B) G(B)
```

This is a perfectly balanced tree with no red nodes. Inserting keys in the order D, B, F, A, C, E, G (a level-order insertion of a balanced tree) produces a tree that is a perfect binary tree of height 2 (black height 3).

Verification: In-order gives A, B, C, D, E, F, G. All nodes black, so black height = 3 on every path. Height = 2 <= 2 lg(8) = 6. Valid.

Problem 2: 2-3 Tree Correspondence

Problem: Draw the 2-3 tree that corresponds to the following LLRB tree:

```
          M(B)
         / \
       E(B)  R(B)
      / \    / \
     C(B) H(B) P(B) X(B)
     /
    A(R)
```

Solution:

To convert, "flatten" each red link so the red child joins its parent's node as a 3-node:

1. A is a red left child of C, so A and C form a 3-node [A C].
2. All other nodes are black, so they remain as 2-nodes.

First, verify the LLRB is valid. Black heights:
- M -> E -> C -> A -> null: M(B), E(B), C(B) = 3 blacks (A is red).
- M -> E -> C -> [C.right null]: M(B), E(B), C(B) = 3 blacks.
- M -> E -> H -> null: M(B), E(B), H(B) = 3 blacks.
- M -> R -> P -> null: M(B), R(B), P(B) = 3 blacks.
- M -> R -> X -> null: M(B), R(B), X(B) = 3 blacks.

All paths have black height 3. Valid.

The corresponding 2-3 tree:

```
              [M]
            /     \
          [E]     [R]
         /   \   /   \
      [A C] [H] [P]  [X]
```

All leaf paths have length 2 from the root. Perfectly balanced. The 2-3 tree has height 2, matching the black height minus 1 (counting edges rather than nodes).

Problem 3: Height Bound Proof

Problem: Prove that any LLRB tree with N = 15 keys has height at most 6.

Solution:

By Proposition 7.1, the height of an LLRB tree with N keys is at most 2 lg(N+1).

For N = 15: height <= 2 lg(16) = 2 * 4 = 8.

But we can derive a tighter bound. The height is at most 2B where B is the black height. We need B <= lg(N+1) = lg(16) = 4. But can B actually be 4?

A 2-3 tree of height B has at least 2^B - 1 keys. For B = 4: at least 2^4 - 1 = 15 keys. So with exactly N = 15, the black height can be at most 4, but only if every node is a 2-node (a perfect binary tree of 15 nodes has height 3, meaning 4 levels, so B = the number of levels from root to leaf = 3 edges). Let us be more careful.

A perfect binary tree with 15 nodes has height 3 (root at level 0, leaves at level 3). In the LLRB representation with all black nodes, the black height B = number of black links on a root-to-null path. From root to leaf is 3 edges, plus 1 edge from leaf to null = 4 black links? No -- we count black links from root to null. The root itself contributes one black link (the link from root to its child). Actually, by convention, the black height counts the number of black nodes (not edges) on any path from the root to null, inclusive of the root.

With 15 keys, all black nodes in a perfect binary tree of height 3: the black height is 4 (root + 3 levels below). But the height of the red-black tree is 3 (edges from root to deepest leaf).

Since all nodes are black, the maximum path length is 3, and 2B would be 2*4 = 8, but the actual height is only 3. The bound 2 lg(N+1) = 8 is not tight here.

For a tight bound: the maximum height occurs when we have the maximum number of red nodes on the longest path. With B = 3 black links on a path (possible when some nodes are 3-nodes, reducing the total count), the maximum path length is 2*3 = 6.

Can we achieve B = 3 with 15 keys? A 2-3 tree of height 3 with B = 3 has at least 2^3 - 1 = 7 keys and at most 3^3 - 1 = 26 keys. Since 7 <= 15 <= 26, yes. So B = 3 is achievable, and the height is at most 2 * 3 = **6**.

Therefore, any LLRB tree with 15 keys has height at most 6.

Problem 4: Rotation Practice

Problem: Given the following subtree, apply rotateLeft(E) and show the result. Then apply rotateRight to the result and verify you get back the original.

```
    E(B)
   / \
  A(B) S(R)
      / \
    H(B) X(B)
```

Solution:

rotateLeft(E):
- x = E.right = S
- E.right = x.left = H
- x.left = E
- x.color = E.color = BLACK
- E.color = RED

```
After rotateLeft(E):

    S(B)
   / \
  E(R) X(B)
 / \
A(B) H(B)
```

Verify BST order: A < E < H < S < X. Correct.

rotateRight(S):
- x = S.left = E
- S.left = x.right = H
- x.right = S

- x.color = S.color = BLACK
- S.color = RED

```
After rotateRight(S):

    E(B)
   / \
  A(B) S(R)
      / \
    H(B) X(B)
```

This is identical to the original. Rotations are inverses of each other.

Problem 5: Counting Red Nodes

Problem: What is the minimum and maximum number of red nodes in an LLRB tree with N = 7 keys?

Solution:

Minimum red nodes: 0. This occurs when the corresponding 2-3 tree has all 2-nodes. A 2-3 tree with 7 keys as all 2-nodes requires a perfect binary tree: 2^3 - 1 = 7. So a perfect binary tree of height 2 (3 levels) with 7 nodes works. The LLRB tree has all black nodes.

```
        D(B)
       / \
     B(B)  F(B)
    / \   / \
  A(B) C(B) E(B) G(B)
```

Red nodes: **0**.

Maximum red nodes: 3. To maximize red nodes, maximize 3-nodes in the 2-3 tree. Each 3-node contributes exactly one red node. A 2-3 tree of height 1 (root plus one level of leaves) with 7 keys: 3-node root (2 keys) with three children. To get 7 keys total: 2 + 5 = 7 among children, so we need 5 keys in 3 children: two 3-node children (2 keys each = 4) and one 2-node child (1 key). That gives 3 three-nodes (root + 2 children), for **3 red nodes**.

Example: 2-3 tree root [D G], children [A B], [E], [H]:

```
        [D  G]
       / |    \
    [A B] [E]  [H]
```

Keys: A, B, D, E, G, H = 6 keys. That is only 6. We need 7.

Try: root [D G], children [A B], [E F], [H]:

```
        [D  G]
       / |    \
    [A B] [E F]  [H]
```

Keys: A, B, D, E, F, G, H = 7 keys. 3 three-nodes (root, left child, middle child). So 3 red nodes.

LLRB encoding:

```
          G(B)
         / \
       D(R)  H(B)
      / \
    B(B) F(B)
   /    /
  A(R) E(R)
```

Red nodes: D, A, E = **3**.

Problem 6: Delete Minimum Trace

Problem: Starting from the tree built in Problem 1 (after inserting D, B, F, A, C, E, G), delete the minimum three times and show the tree after each deletion.

Solution:

Starting tree (all black, perfect):

```
         D(B)
        / \
      B(B)  F(B)
     / \   / \
    A(B) C(B) E(B) G(B)
```

Delete minimum (A):

Step 1: isRed(root.left)? B is black. isRed(root.right)? F is black. Both black. Set root.color = RED.

```
         D(R)
        / \
      B(B)  F(B)
     / \   / \
    A(B) C(B) E(B) G(B)
```

Step 2: deleteMinHelper(D). D.left = B, not null. isRed(D.left)? B is black. isRed(D.left.left)? B.left = A is black. Both black. Call moveRedLeft(D).

moveRedLeft(D): flipColors(D): D becomes black, B becomes red, F becomes red.

```
         D(B)
        / \
      B(R)  F(R)
     / \   / \
    A(B) C(B) E(B) G(B)
```

Check isRed(D.right.left)? D.right = F, F.left = E is black. No special case.

Recurse: D.left = deleteMinHelper(B).

Step 3: deleteMinHelper(B). B.left = A, not null. isRed(B.left)? A is black. isRed(B.left.left)? null. Both black. But B must be red for moveRedLeft precondition. B IS red (from flipColors). Call moveRedLeft(B).

moveRedLeft(B): flipColors(B): B becomes black, A becomes red, C becomes red.

```
    B(B)
   / \
  A(R) C(R)
```

Check isRed(B.right.left)? B.right = C, C.left is null. No.

Recurse: B.left = deleteMinHelper(A). A.left is null. Return null. Delete A.

Step 4: balance(B). B.left = null, B.right = C(R).

```
  B(B)
    \
    C(R)
```

isRed(B.right) and not isRed(B.left)? Yes! rotateLeft(B).

```
  C(B)
  /
 B(R)
```

Done with balance.

Step 5: Back at D. D.left = C(B).

```
       D(B)
      / \
    C(B)  F(R)
   /     / \
  B(R) E(B) G(B)
```

balance(D): isRed(D.right) and not isRed(D.left)? F is red, C is black. Yes! rotateLeft(D).

```
          F(B)
         / \
       D(R)  G(B)
      / \
    C(B) E(B)
    /
   B(R)
```

Check at F: isRed(F.left) and isRed(F.left.left)? D is red, D.left = C is black. No. isRed(F.left) and isRed(F.right)? D red, G black. No. Done.

Set root to black (already black):

```
          F(B)
         / \
       D(R)  G(B)
      / \
    C(B) E(B)
    /
   B(R)
```

Tree after deleting A: 6 keys B, C, D, E, F, G. Black heights: F->D(R)->C->B(R)->null: F, C = 2. F->D(R)->E->null: F, E = 2. F->G->null: F, G = 2. Valid.

Delete minimum (B):

Root: F(B). isRed(F.left)? D is red. Not both children black. Do not set root red.

deleteMinHelper(F): F.left = D. isRed(F.left)? D is red. Do NOT call moveRedLeft. Recurse on D.

deleteMinHelper(D): D.left = C. isRed(D.left)? C is black. isRed(D.left.left)? C.left = B is red. Since C.left.left... wait: isRed(D.left) is false. The condition is: not isRed(D.left) and not isRed(D.left.left). isRed(D.left) = false (C is black), isRed(D.left.left) = isRed(C.left) = isRed(B) = true. So NOT false AND NOT true = true AND false = false. Do NOT call moveRedLeft. Recurse on C.

deleteMinHelper(C): C.left = B. isRed(C.left)? B is red. Do NOT call moveRedLeft. Recurse on B.

deleteMinHelper(B): B.left is null. Return null. Delete B.

balance(C): C.left = null, C.right = null. No issues. Return C(B).

balance(D): D.left = C(B), D.right = E(B). No reds. Return D.

balance(F): F.left = D(R), F.right = G(B). isRed(F.right) and not isRed(F.left)? No. isRed(F.left) and isRed(F.left.left)? D is red, D.left = C is black. No. Done.

Set root black (already black):

```
          F(B)
         / \
       D(R)  G(B)
      / \
    C(B) E(B)
```

Tree after deleting B: 5 keys C, D, E, F, G.

Delete minimum (C):

Root: F(B). isRed(F.left)? D is red. Not both black. Do not set root red.

deleteMinHelper(F): isRed(F.left)? D is red. Recurse on D.

deleteMinHelper(D): D.left = C. isRed(D.left)? C is black. isRed(D.left.left)? C.left is null (black). Both black. D is red (confirmed). Call moveRedLeft(D).

moveRedLeft(D): flipColors(D): D -> black, C -> red, E -> red. Check isRed(D.right.left)? D.right = E, E.left is null. No.

```
     D(B)
    / \
   C(R) E(R)
```

Recurse: D.left = deleteMinHelper(C). C.left is null. Return null. Delete C.

balance(D): D.left = null, D.right = E(R). isRed(D.right) and not isRed(D.left)? Yes! rotateLeft(D).

```
  E(B)
  /
D(R)
```

Back at F: F.left = E(B).

```
    F(B)
   / \
  E(B) G(B)
  /
D(R)
```

balance(F): No issues (E is black left, G is black right). Done.

Set root black:

```
    F(B)
   / \
  E(B) G(B)
  /
D(R)
```

Tree after deleting C: 4 keys D, E, F, G. Black heights: F->E->D(R)->null: F, E = 2. F->E->[null]: F, E = 2. F->G->null: F, G = 2. Valid.

Problem 7: Why Left-Leaning?

Problem: Explain why the left-leaning restriction reduces implementation complexity compared to a general red-black BST. How many insertion cases does a general red-black tree have versus an LLRB tree?

Solution:

In a **general red-black BST** (as described by Guibas and Sedgewick in 1978, or in CLRS), red links can lean in either direction. This means a 3-node [J K] can be represented two ways:

1. K is black, J is red left child (left-leaning).

2. J is black, K is red right child (right-leaning).

This ambiguity doubles the number of cases in insertion and deletion. The CLRS red-black tree insertion algorithm has **6 cases** (3 cases when the parent is a left child, mirrored 3 cases when the parent is a right child). The deletion algorithm has **6 more cases**, for a total of about 12 distinct transformation cases.

In a **left-leaning red-black BST**, the left-leaning restriction eliminates the ambiguity: every 3-node has exactly one representation. This reduces insertion to **3 fix-up steps** (rotate left, rotate right, flip colors), applied unconditionally in sequence after each recursive insertion call. The deletion algorithm is similarly simplified.

The key simplification is that the LLRB fix-up code does not need `if-else` branches to determine the "case" -- it simply checks three conditions and applies transformations as needed. The conditions are not mutually exclusive; multiple may apply in sequence (e.g., after a rotate right, the flip colors condition may become true). This sequential application of simple transformations replaces the complex case analysis of general red-black trees.

Additionally, the LLRB tree has a bijective correspondence with 2-3 trees (not 2-3-4 trees), which further simplifies the analysis and makes the invariants easier to reason about.

Problem 8: Empirical Comparison

Problem: If you insert N = 10,000 random keys into (a) an unbalanced BST, (b) an LLRB tree, and (c) a hash table with separate chaining, estimate the expected height/maximum chain length and the total number of comparisons for all insertions.

Solution:

(a) Unbalanced BST:
- Expected height: ~4.311 * ln(10000) = ~4.311 * 9.21 = ~39.7, so about 40.
- Average internal path length per insertion: ~1.39 lg N = ~1.39 * 13.29 = ~18.5.
- Total comparisons for N insertions: ~10000 * 18.5 = ~185,000 comparisons.
- Worst case height (sorted input): 10,000.

(b) LLRB tree:
- Height: at most 2 lg(10001) = ~26.6, so at most 27. In practice, approximately lg(10000) = ~13.3, so about 14.

- Total comparisons: approximately N * lg N = 10000 * 13.3 = ~133,000. With the constant factor of rotations and flips, wall-clock time is similar to the BST, but the worst-case guarantee is far superior.

(c) Hash table (separate chaining):

- With load factor alpha = N/M = 1 (table size = N): average chain length is 1. Expected maximum chain length is O(lg N / lg lg N) ~ 5-6.
- Total operations for N insertions: each insertion hashes (O(1)) and appends. Comparisons for duplicate check: ~N * alpha/2 = ~5,000.
- Much faster for pure lookup, but no ordering.

Metric	BST	LLRB	Hash Table
Expected height / max chain	~40	~14	~5
Worst-case height / max chain	10,000	27	10,000*
Total comparisons (all inserts)	~185,000	~133,000	~15,000
Supports ordered ops?	Yes	Yes	No

(*The hash table worst case assumes all keys hash to the same bucket, which is extremely unlikely with a good hash function but possible with adversarial input.)

Summary

Key Concepts

Concept	Description
2-3 Tree	Balanced search tree with 2-nodes and 3-nodes; always perfectly balanced
Red-Black BST	BST with colored links encoding a 2-3 tree; red links represent 3-nodes
Left-Leaning (LLRB)	Red links lean left only; simplifies to 3 fix-up cases
Left Rotation	Fixes right-leaning red link by rotating right child up
Right Rotation	Fixes two consecutive left reds by rotating left child up
Color Flip	Splits a temporary 4-node (both children red) by recoloring
Insert Fix-Up	Three steps: rotateLeft, rotateRight, flipColors (in order, bottom-up)
moveRedLeft/Right	Ensures current node is red during deletion descent
Black Height	Number of black links on any root-to-null path (constant for all paths)
Height Bound	At most 2 lg(N+1); guarantees O(lg N) for all operations

Performance Comparison

Operation	Unordered List	Ordered Array	BST (avg)	BST (worst)	Red-Black BST	AVL Tree	Hash Table (avg)
Search	N	lg N	1.39 lg N	N	2 lg N	1.44 lg N	1
Insert	N	N	1.39 lg N	N	2 lg N	1.44 lg N	1
Delete	N	N	sqrt(N)*	N	2 lg N	1.44 lg N	1
Min/Max	N	1	lg N**	N	2 lg N	1.44 lg N	N
Floor/Ceiling	N	lg N	1.39 lg N	N	2 lg N	1.44 lg N	N
Rank/Select	N	1 / lg N	1.39 lg N	N	2 lg N	1.44 lg N	N
Ordered iteration	N lg N	N	N	N	N	N	N lg N

(*BST deletion with Hibbard's method degrades to sqrt(N) height over time.)*
*(*BST min is the depth of the leftmost node, which is O(lg N) on average but O(N) worst case.)

Key Takeaways

1. **2-3 trees** provide the conceptual foundation: perfect balance through node splitting at the root.
2. **Red-black BSTs** are the practical realization: 2-3 trees encoded as binary trees with colored links.
3. The **left-leaning** restriction reduces insertion to three simple fix-up steps.
4. The height bound of **2 lg(N+1)** guarantees worst-case logarithmic performance for all operations.
5. Red-black BSTs are the standard implementation behind ordered collections in Java, C++, and many other languages.

6. For applications requiring only search/insert (no ordering), hash tables are faster. For applications requiring ordered operations or worst-case guarantees, red-black BSTs are the right choice.

Looking ahead: In Lecture 8, we will study **hash tables** in depth -- the other major approach to the symbol table problem. Hash tables achieve O(1) average-case performance for search and insert by trading away ordered operations. We will study hash functions, collision resolution (separate chaining and linear probing), and the theoretical foundations of hashing.

References

1. **Sedgewick, R. & Wayne, K.** *Algorithms*, 4th Edition. Addison-Wesley, 2011. Chapter 3.3 (Balanced Search Trees). The primary reference for left-leaning red-black BSTs.

2. **Sedgewick, R.** "Left-Leaning Red-Black Trees." Department of Computer Science, Princeton University, 2008. Available at: https://www.cs.princeton.edu/~rs/talks/LLRB/LLRB.pdf. The original paper introducing the LLRB simplification.

3. **Guibas, L.J. & Sedgewick, R.** "A Dichromatic Framework for Balanced Trees." *Proceedings of the 19th Annual IEEE Symposium on Foundations of Computer Science (FOCS)*, pp. 8-21, 1978. The seminal paper introducing red-black trees.

4. **Cormen, T.H., Leiserson, C.E., Rivest, R.L., & Stein, C.** *Introduction to Algorithms*, 4th Edition. MIT Press, 2022. Chapter 13 (Red-Black Trees). The standard textbook treatment with the general (non-left-leaning) variant.

5. **Bayer, R.** "Symmetric Binary B-Trees: Data Structure and Maintenance Algorithms." *Acta Informatica*, 1(4):290-306, 1972. An early formulation of balanced binary trees equivalent to red-black trees.

6. **Adelson-Velsky, G.M. & Landis, E.M.** "An Algorithm for the Organization of Information." *Proceedings of the USSR Academy of Sciences*, 146:263-266, 1962. The original AVL tree paper.

7. **Bayer, R. & McCreight, E.** "Organization and Maintenance of Large Ordered Indexes." *Acta Informatica*, 1(3):173-189, 1972. The original B-tree paper.

8. **Knuth, D.E.** *The Art of Computer Programming, Volume 3: Sorting and Searching*, 2nd Edition. Addison-Wesley, 1998. Section 6.2.3 (Balanced Trees). Comprehensive historical and analytical treatment.

9. **Sedgewick, R.** "Algorithms in C++, Parts 1-4: Fundamentals, Data Structures, Sorting, Searching," 3rd Edition. Addison-Wesley, 1998. Chapter 13 (Balanced Trees). Earlier treatment with red-black tree implementations.

10. **Booksite:** https://algs4.cs.princeton.edu/33balanced/ -- Sedgewick & Wayne's companion website with Java implementations, visualizations, and exercises for balanced search trees.

Lecture 7 of 12 -- *Data Structures & Algorithms* -- *Based on Sedgewick's Algorithms, 4th Edition*