Red-Black Tree

- This is a binary tree where height is balanced and thus it provides efficient search.
- We need to maintain the balance during insertions and deletions.

Properties of RB tree.

A binary tree where each edge is colored either red or black. One may store a bit in one of the nodes (lower one) to indicate the color.

Red-black property.

1. Every edge to a leaf is black

2. No (downward) path has 2 consecutive red edges.

3. Every path from a node \( m \) to a leaf has the same # of black edges, the black-height \( bh(m) \).
Example.

\[ bh(P) = 3 \]
\[ h(P) = 6 \]

Longest path is at most twice as long as the shortest path from the root to a leaf.

Claim. A red-black tree with \( n \) internal nodes has height at most \( 2 \log(n+1) \).
Proof. The tree has \( n+1 \) leaves. Now contract red edges, that is, identify their respective 2 nodes. The example red-black tree becomes

```
Every leaf has depth \( bh(p) \).
Every remaining interior node has at least 2 children.
Thus, \( 2^{bh(p)} \leq n+1 \)
\( \lor \ bh(p) \leq \log(n+1) \)
\( \lor \ h(p) \leq 2bh(p) \leq 2\log(n+1) \).
```

The main tool used to balance the tree is rotation.
Rotations

A rotation is a local restructuring operation designed to improve the balance.

Important: A rotation does not change the inorder sequence.

Procedure

Left-Rotate (p, x: Node) [p: root assume x.r ≠ nil]

\[ \begin{align*}
    y &:= x.r; \\
    x.r &:= y.l; \\
    \text{if } y.l \neq \text{nil} \text{ then } y.l.p &:= x \text{ endif;} \\
    y.p &:= x.p; \\
    \text{if } x.p \neq \text{nil} \text{ then } p &:= y \text{ else if } x = x.p.l \text{ then } \text{else if } x.p.r = \text{nil} \text{ then } x.p.l &:= y \text{ endif} \\
    \text{else } x.p.r &:= y \text{ endif} \\
    \text{endif} \\
    y.l &:= x; \\
    x.p &:= y.
\end{align*} \]
There is also a composite type of rotation:

1. Single left rotate X
2. Single right rotate Z

An example. We look at the operations for an example first to get an idea.

Sequence: 10, 7, 13, 4, 2, 5, 6.

Add 10, 7, 13, 4
Insertion.

First we add the new key \( x \) by replacing a proper leaf as for binary search tree. Color the incoming edge (from parent) red. Then, adjust color and structure at \( y := x.p \)

Invariant.

1. If \( y \) has a red incoming edge and a red outgoing edge then this is the only violation of the red-black tree property.

2. If \( y \) has a red incoming edge then it has exactly one red outgoing edge; otherwise there are one or two outgoing edges.

Case 1. Incoming edge of \( y \) is black: Done

Case 2. Incoming edge of \( y \) is red. Set \( M := y.p \)

Case 2.1 Both outgoing edges of \( M \) are red; promote \( M \);

if \( M.p = \text{nil} \) then \( y := M.p \)
endif and recurse for \( y \).
Case 2.2 \( \gamma \) is left child of \( \mu \), and left outgoing edge of \( \gamma \) is red.

- Single rotate \( \mu \) to right.
- Done.

(There is a symmetric right-right case)

Case 2.3 \( \gamma \) is left child of \( \mu \); and right outgoing edge of \( \gamma \) is red.

- Double rotate \( \mu \) to right
- Done.

(There is a symmetric right-left case)

Observe that invariants are maintained and then at most 2 rotations needed.
Deletions. First find the node \( x \) that stores the item that needs to be deleted.

By substituting with successor or predecessor we can assume that \( x \) has 2 leaves as children.

Caveat: you may need to perform several successor or predecessor operation. For example, if \( x \) has a right child, do successor and repeat if that successor(\( x \)) has a right child.... and so on. Successor(\( x \)) cannot have left child. If \( x \) had a left child but no right child, do a predecessor.....

Since \( x \) has 2 leaves, we can replace \( x \) by a leaf \( \lambda \). If the incoming edge of \( x \) is red than that of \( \lambda \) should be black. If the incoming edge of \( x \) is black then we have a problem and need to restructure. Make the incoming edge as "double black". Start restructuring with \( \gamma := \lambda \).
Invariant. If the incoming edge of \( r \) is black then we have a valid red-black tree; otherwise, the incoming double-black edge is the only violation of the red-black property.

**Case 1.** Incoming edge of \( r \) is black. Done.

**Case 2.** Incoming edge of \( r \) is double-black. \( \mu := r \cdot b; \ r \) is sibling of \( r \).

**Case 2.1** Edge from \( \mu \) to \( r \) is black.

(In this case \( r \) is not a leaf): otherwise, \( \mu \) will not have same black-height on both sides.

**Case 2.1.1** Both outgoing edges of \( r \) are black

Demote \( \mu \); recurse for \( r := \mu \).
Case 2.1.2 \( \gamma \) is left child of \( m \) and right outgoing edge of \( \gamma \) is red. 
Single rotate \( m \) to left. Done.

\[
\begin{align*}
\text{Case 2.1.3} & \quad \gamma \text{ is left child of } m \text{ and left outgoing edge of } \gamma \text{ is red, the other one is black. Double rotate } m \text{ to left. Done.} \\
\end{align*}
\]

Case 2.2 Edge from \( m \) to \( \gamma \) is red, assume \( \gamma \) is left child of \( m \). Single rotate \( m \) to left, recurse for \( \gamma \). (Next step case 2.1, terminates)
Summary:

A red-black tree supports operations search, minimum, maximum, successor, predecessor, insertion, deletion in time $O(\log n)$ each. A single insertion or deletion requires at most 3 rotations (only during promote or demote recursions happen that do not involve any rotations).