Red-Black Trees

(read Section 13 on Red-Black Trees in Cormen, Leiserson, Rivest, Stein)

Binary search trees are an elegant implementation of the dictionary data type, which requires support for

```plaintext
item search(item),
void insert(item),
void delete(item),
```

and possible other miscellaneous operations, such as finding the minimum, etc. Their main disadvantage is the worst case time $\Omega(n)$ for a single operation. The reasons are insertions and deletions that tend to get the tree unbalanced. It is possible to counteract this tendency with occasional local restructuring operations and to guarantee logarithmic time per operation.

2-3-4 Trees and Red-Black Trees. A special type of balanced tree is the 2-3-4 tree. Every internal node in such a tree stores one, two, or three items and has two, three, or four children, and every leaf has the same depth. As shown in Figure 21, the items in the internal nodes separate the items stored in the subtrees and thus facilitate fast searching. In the smallest 2-3-4 tree of height $h$ every internal node has exactly two children, so we have $2^h$ leaves and $2^h - 1$ internal nodes. In the largest 2-3-4 tree of height $h$ every internal node has four children, so we have $4^h$ leaves and $(4^h - 1)/3$ internal nodes. We can store a 2-3-4 tree in a binary tree by expanding a node with $i > 1$ items and $i + 1$ children into $i$ nodes each with one item, as shown in Figure 21.

![Figure 21: A 2-3-4 tree of height 2. All items are stored in internal nodes.](image)

Figure 22. Suppose we color each edge of a binary search tree either red or black. The color is conveniently stored in the lower node of the edge. Such an edge-colored tree is a red-black tree if

1. every edge to a leaf is black,
2. no path from the root to a leaf has two consecutive red edges,
3. all paths from the root to the leaf level have the same number of black edges.

![Figure 22: Transforming a 2-3-4 tree into a binary tree. Bold edges are called ‘red’ and the others are called ‘black’.](image)

Figure 23: A red-black tree corresponding to the 2-3-4 tree in Figure 21.

The number of black edges on a path from the root $\rho$ to a leaf is the black height of $\rho$, denoted as $bh(\rho)$. Note that
transforming a 2-3-4 tree into a binary tree gives a red-black tree. The result of transforming the tree in Figure 21 is shown in Figure 23.

**HEIGHT LEMMA.** A red-black tree with \( n \) internal nodes has height at most \( 2 \log_2(n+1) \).  

**PROOF.** The number of leaves is \( n+1 \). Contract each red edge to get a 2-3-4 tree with \( n+1 \) leaves. Its height is \( h \leq \log_2(n+1) \). We have \( bh(\varnothing) = h \), and by rule (2) the height of the red-black tree is at most \( 2bh(\varnothing) \leq 2 \log_2(n+1) \).

**Rotations.** Restructuring a red-black tree can be done with only one operation (and its symmetric version). This operation is called a rotation and it moves a subtree from one side to another, as shown in Figure 24. The inorder sequence of the left tree in Figure 24 is 

\[ \ldots, \text{inorder}(A), \nu, \text{inorder}(B), \mu, \text{inorder}(C), \ldots, \]

and this is also the inorder sequence of the right tree. In other words, a rotation maintains the inorder sequence. Function \( \text{ZIG} \) below implements the right rotation:

```c
Node * Zig(Node * μ)
assert μ ≠ NULL and ν = μ → ℓ ≠ NULL;
μ → ℓ = ν → r; ν → r = μ; return ν.
```

Function \( \text{ZAG} \) is symmetric to \( \text{ZIG} \) and performs a left rotation. Occasionally it is necessary to perform two rotations in sequence, and it is convenient to combine them into a single operation referred to as a double rotation, as shown in Figure 25. We use a function \( \text{ZIGZAG} \) to implement a double right rotation and the symmetric function \( \text{ZAGZIG} \) to implement a double left rotation.

```c
Node * ZIGZAG(Node * μ)
μ → ℓ = ZAG(μ → ℓ); return Zig(μ).
```

**Insertion.** Before studying the details of the restructuring algorithms for red-black trees we look at the trees that arise in a short insertion sequence, as shown in Figure 26. After adding 10, 7, 13, 4 we have two red edges in sequence and repair this by promoting 10 (A). After adding 2 we repair the two red edges in sequence by a single rotation of 7 (B). After adding 5 we promote 4 (C), and after adding 6 we do a double rotation of 7 (D).

An item \( x \) is added by substituting a new internal node for a leaf at the appropriate position. To satisfy condition (3) of the red-black tree definition, color the incoming edge of the new node red, as shown in Figure 27. Start the adjustment of color and structure at the parent \( ν \) of the new node. We state the properties maintained by the insertion algorithm in the following two invariants that apply to a
node \( \nu \) traced by the algorithm.

(I) The only possible violation of the red-black tree is the violation of property (2) by the node \( \nu \). Furthermore, if \( \nu \) has a red incoming edge then it has exactly one red outgoing edge, otherwise, it has one or two red outgoing edges.

Observe that (I) holds right after adding \( x \). We continue with the analysis of all cases that may arise. The local adjustment operations depend on the neighborhood of \( \nu \).

**Case 1** The incoming edge of \( \nu \) is black. Done.

**Case 2** The incoming edge of \( \nu \) is red. Let \( \mu \) be the parent of \( \nu \) and assume \( \nu \) is left child of \( \mu \).

**Case 2.1** Both outgoing edges of \( \mu \) are red, as shown in Figure 28. Promote \( \mu \). Let \( \nu \) be the parent of \( \mu \) and recurse.

![Figure 28: Promotion of \( \mu \). The colors of the left and right outgoing edges of \( \nu \) might also be swapped.](image)

**Case 2.2** The edge from \( \mu \) to \( \nu \) is red and the left outgoing edge of \( \nu \) is red, as shown in Figure 29 to the left. Right rotate \( \mu \). Done.

![Figure 29: Right rotation of \( \mu \) to the left and double right rotation of \( \mu \) to the right.](image)

**Case 2.3** The edge from \( \mu \) to \( \nu \) is red and the right outgoing edge of \( \nu \) is red, as shown in Figure 29 to the right. Double right rotate \( \mu \). Done.

Case 2 has a symmetric case where left and right are exchanged. An insertion may cause logarithmically many promotions but at most two rotations.

**Deletion.** First find the node \( \pi \) that is to be removed. If necessary we substitute the inorder successor for \( \pi \) or the inorder successor of the inorder successor so we can assume that both children of \( \pi \) are leaves. If \( \pi \) is the last node we substitute symmetrically. Replace \( \pi \) by a leaf \( \nu \), as shown in Figure 30. If the incoming edge of \( \pi \) is red then mark the incoming edge of \( \nu \) black. Otherwise, remember the incoming edge of \( \nu \) as ‘double-black’, which counts as two black edges and is drawn as a dashed line.

To understand the deletion algorithm we need again an invariant, which holds right after removing \( \pi \).

(D) The only possible violation of the red-black tree properties is a double-black incoming edge of \( \nu \).

We now present the analysis of all the possible cases. The adjustment operation is chosen depending on the local neighborhood of \( \nu \).

**Case 1** The incoming edge of \( \nu \) is black. Done.

**Case 2** The incoming edge of \( \nu \) is double-black. Let \( \mu \) be the parent and \( \kappa \) the sibling of \( \nu \). Assume \( \nu \) is left child of \( \mu \) and note that \( \kappa \) is internal.

**Case 2.1** The edge from \( \mu \) to \( \kappa \) is black.

**Case 2.1.1** Both outgoing edges of \( \kappa \) are black, as shown in Figure 31. Demote \( \mu \). Recurse for \( \nu = \mu \).

![Figure 31: Demotion of \( \mu \).](image)
Case 2.1.2 The right outgoing edge of $\kappa$ is red, as shown in Figure 32 to the left. Change the color of that edge to black and left rotate $\mu$. Done.

![Figure 32: Left rotation of $\mu$ to the left and double left rotation of $\mu$ to the right.](image)

Case 2.1.3 The right outgoing edge of $\kappa$ is black, as shown in Figure 32 to the right. Change the color of the left outgoing edge to black and double left rotate $\mu$. Done.

Case 2.2 The edge from $\mu$ to $\kappa$ is red, as shown in Figure 33. Left rotate $\mu$. Recurse for $\nu$.

![Figure 33: Left rotation of $\mu$.](image)

Case 2 has a symmetric case where $\nu$ is the right child of $\mu$. Case 2.2 seems problematic because it recurses without moving $\nu$ any closer to the root. However, the configuration excludes the possibility of Case 2.2 occurring again. If we enter Case 2.1.2 or Case 2.1.3 then the termination is immediate. If we enter Case 2.1.1 then the termination follows because the incoming edge of $\mu$ is red. The deletion may cause logarithmically many demotions but at most three rotations.

### Summary
The red-black tree is an implementation of the dictionary data type and supports the operations search, insert, delete in logarithmic time each. An insertion or deletion requires at most the equivalent of three single rotations. The red-black tree also supports finding the minimum, maximum and the inorder successor, predecessor of a given node in logarithmic time each.