Red Black Trees
**Observations**

There are many variations of balanced binary trees. The prominent among them are Red Black trees, B-trees (again of many kinds), weak AVL trees etc.; they all share the same property that (1) height of the tree is $O(\log_2 n)$, where $n$ is the number of nodes and (2) the worst-case time for each of the operations of search, insert, delete is $O(\log_2 n)$.

Limitations of Balanced Search Trees: Balanced search trees require storing an extra piece of information per node. Their worst-case, average-case, and best-case performance are essentially identical. We do not win when easy inputs occur – would be nice if the second access to the same piece of data was cheaper than the first. The 90-10 rule – empirical studies suggest that in practice 90% of the accesses are to 10% of the data items; it would be nice to get easy wins for the 90% case.

A **red-black** tree is a kind of self-balancing binary search tree where each node has an extra bit, and that bit is often called the color (red or black). These colors are used to ensure that the tree remains balanced during insertions and deletions. Although the balance of the tree is not perfect, it is good enough to reduce the searching time and maintain it around $O(\log n)$ time, where $n$ is the total number of elements in the tree.
Why Red-Black Trees?

Most of the BST operations (e.g., search, max, min, insert, delete.. etc) take $O(h)$ time where $h$ is the height of the BST. The cost of these operations may become $O(n)$ for a skewed Binary tree. If we make sure that the height of the tree remains $O(\log n)$ after every insertion and deletion, then we can guarantee an upper bound of $O(\log n)$ for all these operations. The height of a Red-Black tree is always $O(\log n)$ where $n$ is the number of nodes in the tree.

Most of the BST operations (e.g., search, max, min, insert, delete.. etc) take $O(h)$ time where $h$ is the height of the BST. The cost of these operations may become $O(n)$ for a skewed Binary tree. If we make sure that the height of the tree remains $O(\log n)$ after every insertion and deletion, then we can guarantee an upper bound of $O(\log n)$ for all these operations. The height of a Red-Black tree is always $O(\log n)$ where $n$ is the number of nodes in the tree.

Comparison with AVL Tree: The AVL trees are more balanced compared to Red-Black Trees, but they may cause more rotations during insertion and deletion.

- If the application involves frequent insertions and deletions, then Red-Black trees should be preferred.
- And if the insertions and deletions are less frequent and search is a more frequent operation, then AVL tree should be preferred over Red-Black Tree.
Rules of Red Black Trees

A red-black tree is a binary search tree with one extra bit of storage per node: its color, which can be either RED or BLACK.

By constraining the node colors on any simple path from the root to a leaf, red-black trees ensure that no such path is more than twice that of any other, so that the tree is approximately balanced.

Each node of the tree contains the attributes color, key, left, right, and p [parent]. If a child or the parent of a node does not exist, the corresponding pointer attribute of the node contains the value NIL. We regard these NILs as being pointers to leaves (external nodes) of the binary search tree and the normal, key-bearing nodes as being internal nodes of the tree. A red-black tree satisfies the following red-black properties:

1. Every node is either red or black.
2. The root is black.
3. Every leaf (NIL) is black.
4. If a node is red, then both its children are black.
5. For each node, all simple paths from the node to descendant leaves contain the same number of black nodes.
As a matter of convenience in dealing with boundary conditions in red-black tree code, we use a single sentinel to represent NIL. For a red-black tree $T$, the sentinel $T.nil$ is an object with the same attributes as an ordinary node in the tree. Its color attribute is BLACK, and its other attributes—$p$, left, right, and key—can take on arbitrary values. As the second picture shows [see next page], all pointers to NIL are replaced by pointers to the sentinel $T.nil$.

We use the sentinel so that we can treat a NIL child of a node $x$ as an ordinary node whose parent is $x$. [Although we instead could add a distinct sentinel node for each NIL in the tree, so that the parent of each NIL is well defined, that approach would waste space.]

- We use the one sentinel $T.nil$ to represent all the NILs—all leaves and the root’s parent.
- The values of the attributes $p$, left, right, and key of the sentinel are immaterial, although we may set them during a procedure for our convenience.
- We generally confine our interest to the internal nodes of a red-black tree, since they hold the key values. We omit the leaves when we draw red-black trees,
Every leaf, shown as a NIL, is black. Each non-NIL node is marked with its black-height; NILS have black-height 0.

The same red-black tree but with each NIL replaced by the single sentinel T:nil, which is always black, and with black-heights omitted.
The same red-black tree but with leaves and the root’s parent omitted entirely.
Theorem: A red-black tree with n internal nodes has height at most 2 \( \log (n + 1) \).

Proof: We need to show that the subtree rooted at any node \( x \) contains at least \( 2^{bh(x)} - 1 \) internal nodes. We prove this claim by induction on the height of \( x \).

If the height of \( x \) is 0, then \( x \) must be a leaf (T.nil), and the subtree rooted at \( x \) indeed contains at least \( 2^{bh(x)} - 1 = 2^0 - 1 = 0 \) internal nodes. For the inductive step, consider a node \( x \) that has positive height and is an internal node with two children. Each child has a black-height of either \( bh(x) \) or \( bh(x) - 1 \), depending on whether its color is red or black. Since the height of a child of \( x \) is less than the height of \( x \) itself, we can apply the inductive hypothesis to conclude that each child has at least \( 2^{bh(x)} - 1 \) internal nodes. Thus, the subtree rooted at \( x \) contains at least \( (2^{bh(x)} - 1) + (2^{bh(x)} - 1) + 1 = (2^{bh(x)} - 1) \) internal nodes, which proves the claim.

To complete the proof of the theorem, let \( h \) be the height of the tree. According to property 4, at least half the nodes on any simple path from the root to a leaf, not including the root, must be black. Consequently, the black-height of the root must be at least \( h/2 \); thus, \( n \geq (2^{bh(x)} - 1) \). Simplifying, we get \( \log (n + 1) \geq h/2 \) or \( h \leq 2 \log(n+1) \).
Now, we can implement the dynamic-set operations SEARCH, MINIMUM, MAXIMUM, SUCCESSOR, and PREDECESSOR in $O(lg n)$ time on red-black trees, since each can run in $O(h)$ time on a binary search tree of height $h$ and any red-black tree on $n$ nodes is a binary search tree with height $O(lg n)$.

The search-tree operations TREE-INSERT and TREE-DELETE, when run on a red black tree with $n$ keys, take $O(lg n)$ time. Because they modify the tree, the result may violate the red-black properties.

To restore these properties, we must change the colors of some of the nodes in the tree and also change the pointer structure.

We change the pointer structure through rotation, which is a local operation in a search tree that preserves the binary-search-tree property. There are two kinds of rotations: left rotations and right rotations.

When we do a left rotation on a node $x$, we assume that its right child $y$ is not $T.nil$; $x$ may be any node in the tree whose right child is not $T.nil$. The left rotation “pivots” around the link from $x$ to $y$. It makes $y$ the new root of the subtree, with $x$ as $y$’s left child and $y$’s left child as $x$’s right child.

The pseudocode for LEFT-ROTATE assumes that $x.right \neq T.nil$ and that the root’s parent is $T.nil$. 
The operation LEFT-ROTATE(T, x) transforms the configuration of the two nodes on the right into the configuration on the left by changing a constant number of pointers.

The inverse operation RIGHT-ROTATE(T; y/) transforms the configuration on the left into the configuration on the right. The letters $\alpha$, $\beta$, $\gamma$ represent arbitrary subtrees. A rotation operation preserves the binary-search-tree property: the keys in $\alpha$ precede $x$.key, which precedes the keys in $\beta$, which precede $y$.key, which precedes the keys in $\gamma$.

Note that in both trees, an in-order traversal yields $\alpha x \beta y \gamma$; Why?
The operation LEFT-ROTATE(T, x) transforms the configuration of the two nodes on the right into the configuration on the left by changing a constant number of pointers.

```plaintext
LEFT-ROTATE(T, x)
1  y = x.right  // set y
2  x.right = y.left  // turn y's left subtree into x's right subtree
3  if y.left ≠ T.nil
4   y.left.p = x
5  y.p = x.p  // link x's parent to y
6  if x.p == T.nil
7    T.root = y
8  elseif x == x.p.left
9    x.p.left = y
10  else x.p.right = y
11  y.left = x  // put x on y's left
12  x.p = y
```
The operation \textsc{LEFT-ROTATE}(T, x) transforms the configuration of the two nodes on the right into the configuration on the left by changing a constant number of pointers.
We can insert a node into an n-node red-black tree in $O(\lg n)$ time.

To do so, we use a slightly modified version of the TREE-INSERT (we did earlier) to insert node $z$ into the tree $T$ as if it were an ordinary binary search tree, and then we color $z$ red.

To guarantee that the red-black properties are preserved, we then call an auxiliary procedure RB-INSERT-FIXUP to recolor nodes and perform rotations.

The call RB-INSERT ($T, z$) inserts node $z$, whose key is assumed to have already been filled in, into the red-black tree $T$. 
The procedures TREE-INSERT and RB-INSERT differ in four ways. First, all instances of NIL in TREE-INSERT are replaced by T.nil. Second, we set z.left and z.right to T.nil in lines 14–15 of RB-INSERT, in order to maintain the proper tree structure. Third, we color z red in line 16. Fourth, because coloring z red may cause a violation of one of the red-black properties, we call RB-INSERT-FIXUP(T, z) in line 17 of RB-INSERT to restore the red-black properties.
Left Rotation

The left_rotate operation may be encoded:

```c
left_rotate( Tree T, node x ) {
    node y;
    y = x->right;
    /* Turn y's left sub-tree into x's right sub-tree */
    x->right = y->left;
    if ( y->left != NULL )
        y->left->parent = x;
    /* y's new parent was x's parent */
    y->parent = x->parent;
    /* Set the parent to point to y instead of x */
    /* First see whether we're at the root */
    if ( x->parent == NULL ) T->root = y;
    else
        if ( x == (x->parent)->left )
            /* x was on the left of its parent */
            x->parent->left = y;
        else
            /* x must have been on the right */
            x->parent->right = y;
    /* Finally, put x on y's left */
    y->left = x;
    x->parent = y;
}
```

Node Structure:

```c
struct t_red_black_node {
    enum { red, black } color;
    void *item;
    struct t_red_black_node *left,
    *right,
    *parent;
}
```
**Insertion**

```
rb_insert( Tree T, node x ) {
    /* Insert in the tree in the usual way */
    tree_insert( T, x );
    /* Now restore the red-black property */
    x->color = red;
    while ( (x != T->root) && (x->parent->color == red) ) {
        if ( x->parent == x->parent->parent->left ) {
            /* If x's parent is a left, y is x's right 'uncle' */
            y = x->parent->parent->right;
            if ( y->color == red ) {
                /* case 1 - change the colours */
                x->parent->color = black;
                y->color = black;
                x->parent->parent->color = red;
                /* Move x up the tree */
                x = x->parent->parent;                  }
            else { /* y is a black node */
                if ( x == x->parent->right ) {
                    /* and x is to the right */
                    /* case 2 - move x up and rotate */
                    x = x->parent;
                    left_rotate( T, x );             }
            }
        } else { /* repeat the "if" part with right
                  and left exchanged */
            /
        }
    }
    /* case 3 */
    x->parent->color = black;
    x->parent->parent->color = red;
    right_rotate( T, x->parent->parent );
}
/* color the root black */
T->root->color = black;
```