Red-Black Trees

A binary search tree becomes less efficient when the tree becomes unbalanced. For instance, if we store a set of numbers in increasing order, we get a tree as shown in Figure 1 that is virtually a linked list. To overcome this weakness of binary search trees, we can use 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`. Each node therefore contains at least the fields *color*, *key*, *left*, *right*, and parent pointer p.

1 Properties of Red-black Trees

To avoid the situation illustrated in Figure 1, red-black trees adhere to the following properties in addition to properties of a binary search tree.

- Every node is either `RED` or `BLACK`.
- Every leaf is NIL and is `BLACK`.
- If a node is `RED`, then both its children are `BLACK`.
- Every simple path from a node to one of its descendant leaf nodes contains the same number of `BLACK` nodes.

Maintaining these properties, a red-black tree with n internal nodes ensures that its height is at most $2\log(n+1)$. Thus, a red-black tree may be unbalanced but will avoid becoming a linked-list that is longer than $2\log(n+1)+1$.

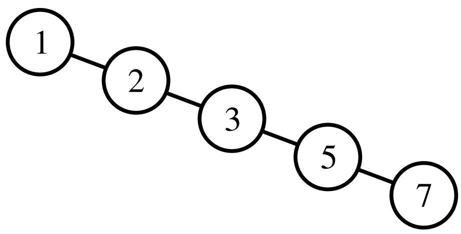

Figure 1: A binary search tree that is no better than a linked-list!

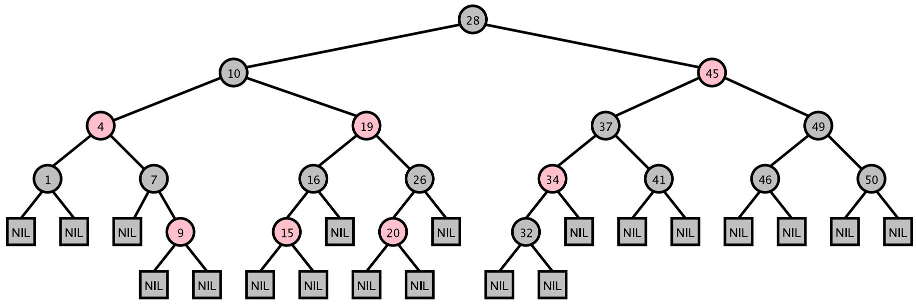

Figure 2: Example Red Black Tree.

Definition: The **black-height** of a node x refers to the number of `BLACK` nodes on any path from, but not including x, to a leaf. The black-height of the tree is the black-height of the root node.

2 Insertion and Deletion

As red-black trees are essentially binary search trees, querying algorithms such as TREE-SEARCH and TREE-MINIMUM can be used on red-black trees. However, due to the red-black tree properties, insertion and deletion are different from TREE-INSERT and TREE-DELETE. Now we may have to change the colours of some nodes in the tree as well as pointer structures. Changing pointer structures is the most important as this allows us to avoid the "excessive linked-list" situation.

2.1 Rotation

We change the pointer structure through rotation, which is a local operation in a search tree that perserves the binary search tree properties. The algorithm for left rotation is shown below along with an illustration in Figure 3. Right rotation is the mirror reflection of left rotation.

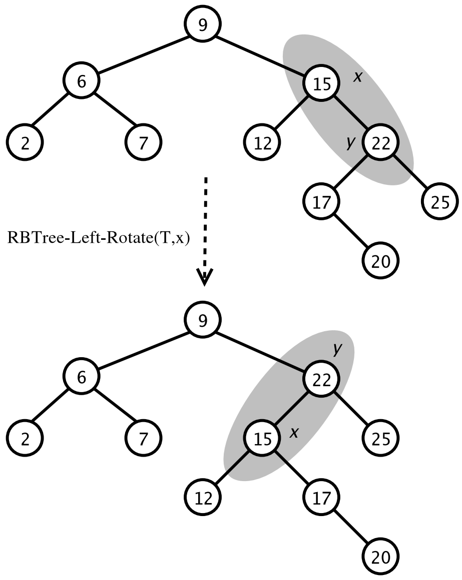

Figure 3: RBTree-Left-Rotate in action. The rotation results in: (1) y takes x's original position, (2) x becomes y's left child, and (3) y's original left child becomes x's right child.

```
RBTree-Left-Rotate(T, x)
y ← right[x]
right[x] ← left[y]
p[left[y]] ← x
p[y] ← p[x]
if p[x] = NIL
   then root[T] ← y
   else if x = left[p[x]]
           then left[p[x]] ← y
           else right[p[x]] ← y
left[y] ← x
p[x] ← y
```

2.2 Insert

We use the Tree-Insert procedure to insert a node x into T as if it were an ordinary binary search tree, and then color x `RED`. If x is the new root node, then we bypass the while-loop and colour it `BLACK`. Otherwise, if x's parent is `BLACK`, then there is nothing to do as adding a `RED` node does not violate any of the red-black tree properties. However, if x's parent is `RED`, then the property that a `RED` node has two `BLACK` nodes is violated (because x is `RED`). In this case, we enter the while-loop.

```
RBTree-Insert(T, x)
Tree-Insert(T, x)  /* See Binary Search Trees */
color[x] ← RED
while x ≠ root[T] and color[p[x]] =RED
      do if p[x] = left[p[p[x]]]  /* Is the parent of x a left child? */
            then y ← right[p[p[x]]]
                 if color[y] =RED
                   then color[p[x]] ← BLACK
                        color[y] ← BLACK
                        color[p[p[x]]] ← RED
                        x ← p[p[x]]
                   else if x = right[p[x]]
                          then x ← p[x]
                        RBTree-Left-Rotate(T, x)
                        color[p[x]] ← BLACK
                        color[p[p[x]]] ← RED
                        RBTree-Right-Rotate(T, p[p[x]])
            else Mirror opposite of "then" clause
 color[root[T]] ← BLACK
```

2.3 Delete

```
RBTree-Delete(T, z)
if left[z] = NIL or right[z] = NIL
   then y ← z
   else  y ← Tree-Successor(z)
if left[y] ≠ NIL
   then x ← left[y]
   else  x ← right[y]
p[x] ← p[y]
if p[y] = NIL
   then root[t] ← x
    else  if y = left[p[y]]
            then left[p[y]] ← x
            else  right[p[y]] ← x
 if y ≠ z
    then key[z] ← key[y]
 if color[y] =BLACK
    then RBTree-Delete-Fixup(T, x)
 return y
```

```
RBTree-Delete-Fixup(T, x)
while x ≠ root[T] and color[x] =BLACK
      do if x = left[p[x]]
            then w ← right[p[x]]
                 if color[w] =RED
                    then color[w] ← BLACK
                         color[p[x]] ← RED
                         RBTree-Left-Rotate(T, p[x])
                         w ← right[p[x]]
               if color[left[w]] =BLACK and color[right[w]] =BLACK
                  then color[w] ← RED
                       x ← p[x]
                  else  if color[right[w]] =BLACK
                           then color[left[w]] ← BLACK
                                color[w] ← RED
                                RBTree-Right-Rotate(T, w)
                                w ← right[p[x]]
                        color[w] ← color[p[x]]
                        color[p[x]] ← BLACK
                        color[right[w]] ← BLACK
                        RBTree-Left-Rotate(T, p[x])
                        x ← root[T]
            else  Mirror opposite of "then" clause
 color[x] ← BLACK
```