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!](image-url)
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 preserves 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.

\[
\text{RBTree-Left-Rotate}(T, x)
\]

1. \( y \leftarrow \text{right}[x] \)
2. \( \text{right}[x] \leftarrow \text{left}[y] \)
3. \( p[\text{left}[y]] \leftarrow x \)
4. \( p[y] \leftarrow p[x] \)
5. \( \text{if } p[x] = \text{NIL} \)
   \( \text{then } \text{root}[T] \leftarrow y \)
6. \( \text{else if } x = \text{left}[p[x]] \)
   \( \text{then } \text{left}[p[x]] \leftarrow y \)
7. \( \text{else } \text{right}[p[x]] \leftarrow y \)
8. \( \text{left}[y] \leftarrow x \)
9. \( p[x] \leftarrow y \)

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.

$$\text{RBTree-Insert}(T, x)$$
1. $\text{Tree-Insert}(T, x)$ /* See Binary Search Trees */
2. $\text{color}[x] \leftarrow \text{RED}$
3. while $x \neq \text{root}[T]$ and $\text{color}[p[x]] = \text{RED}$
4. do if $p[x] = \text{left}[p[p[x]]]$ /* Is the parent of $x$ a left child? */
5. then $y \leftarrow \text{right}[p[p[x]]]$
6. if $\text{color}[y] = \text{RED}$
7. then $\text{color}[p[x]] \leftarrow \text{BLACK}$
8. $\text{color}[y] \leftarrow \text{BLACK}$
9. $\text{color}[p[p[x]]] \leftarrow \text{RED}$
10. $x \leftarrow p[p[x]]$
11. else if $x = \text{right}[p[x]]$
12. then $x \leftarrow p[x]$
13. $\text{RBTree-Left-Rotate}(T, x)$
14. $\text{color}[p[x]] \leftarrow \text{BLACK}$
15. $\text{color}[p[p[x]]] \leftarrow \text{RED}$
16. $\text{RBTree-Right-Rotate}(T, p[p[x]])$
17. else Mirror opposite of “then” clause
18. $\text{color}[\text{root}[T]] \leftarrow \text{BLACK}$
2.3 Delete

RBTree-Delete(T, z)
1  if left[z] = NIL or right[z] = NIL
2    then y ← z
3  else y ← Tree-Successor(z)
4  if left[y] ≠ NIL
5    then x ← left[y]
6    else x ← right[y]
7  p[x] ← p[y]
8  if p[y] = NIL
9    then root[T] ← x
10   else if y = left[p[y]]
11      then left[p[y]] ← x
12      else right[p[y]] ← x
13  if y ≠ z
14    then key[z] ← key[y]
15  if color[y] = BLACK
16    then RBTree-Delete-Fixup(T, x)
17  return y

RBTree-Delete-Fixup(T, x)
1  while x ≠ root[T] and color[x] = BLACK
2    do if x = left[p[x]]
3        then w ← right[p[x]]
4          if color[w] = RED
5            then color[w] ← BLACK
6            color[p[x]] ← RED
7            RBTree-Left-Rotate(T, p[x])
8          w ← right[p[x]]
9          if color[left[w]] = BLACK and color[right[w]] = BLACK
10             then color[w] ← RED
11             x ← p[x]
12          else if color[right[w]] = BLACK
13            then color[left[w]] ← BLACK
14            color[w] ← RED
15            RBTree-Right-Rotate(T, w)
16            w ← right[p[x]]
17            color[w] ← color[p[x]]
18            color[p[x]] ← BLACK
19            color[right[w]] ← BLACK
20            RBTree-Left-Rotate(T, p[x])
21            x ← root[T]
22        else Mirror opposite of “then” clause
23  color[x] ← BLACK