

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 $2 \log(n+1) + 1$.

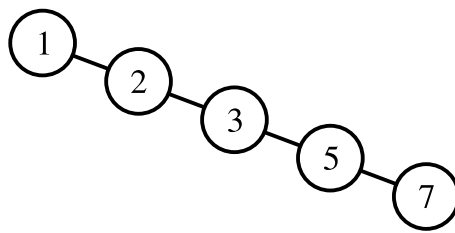
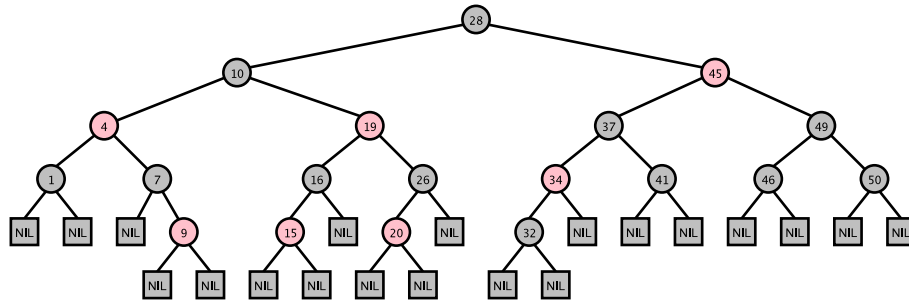


Figure 1: A binary search tree that is no better than a linked-list!



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 **inorder key ordering**. The algorithm for left rotation is shown below along with an illustration in Figure 2. Right rotation is the mirror reflection of left rotation.

```

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  if  $p[x] = \text{NIL}$ 
6      then  $\text{root}[T] \leftarrow y$ 
7  else if  $x = \text{left}[p[x]]$ 
8      then  $\text{left}[p[x]] \leftarrow y$ 
9      else  $\text{right}[p[x]] \leftarrow y$ 
10  $\text{left}[y] \leftarrow x$ 
11  $p[x] \leftarrow y$ 

```

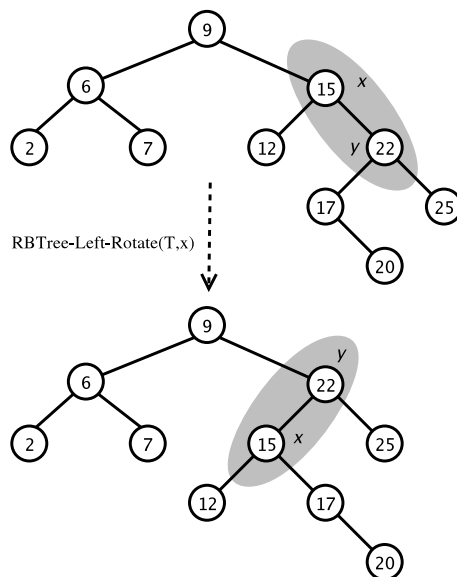


Figure 2: RBTree-LEFT-ROTATE in action. The rotation results in: (1) y takes x 's original position, (2) x becomes y 's left child, and (3) y 's original left child becomes x 's right child.

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( $T, x$ )
1  TREE-INSERT( $T, x$ ) /* See Binary Search Trees */
2  color[ $x$ ] ← RED
3  while  $x \neq \text{root}[T]$  and color[ $p[x]$ ] = 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 color[ $y$ ] = RED
7                  then color[ $p[x]$ ] ← BLACK
8                      color[ $y$ ] ← BLACK
9                      color[ $p[p[x]]$ ] ← RED
10                      $x \leftarrow p[p[x]]$ 

```

```

11         else if  $x = \text{right}[p[x]]$ 
12             then  $x \leftarrow p[x]$ 
13                 RBTREE-LEFT-ROTATE( $T, x$ )
14                  $\text{color}[p[x]] \leftarrow \text{BLACK}$ 
15                  $\text{color}[p[p[x]]] \leftarrow \text{RED}$ 
16                 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  $\text{left}[z] = \text{NIL}$  or  $\text{right}[z] = \text{NIL}$ 
2      then  $y \leftarrow z$ 
3      else  $y \leftarrow \text{TREE-SUCCESSOR}(z)$ 
4  if  $\text{left}[y] \neq \text{NIL}$ 
5      then  $x \leftarrow \text{left}[y]$ 
6      else  $x \leftarrow \text{right}[y]$ 
7   $p[x] \leftarrow p[y]$ 
8  if  $p[y] = \text{NIL}$ 
9      then  $\text{root}[T] \leftarrow x$ 
10     else if  $y = \text{left}[p[y]]$ 
11         then  $\text{left}[p[y]] \leftarrow x$ 
12         else  $\text{right}[p[y]] \leftarrow x$ 
13  if  $y \neq z$ 
14      then  $\text{key}[z] \leftarrow \text{key}[y]$ 
15  if  $\text{color}[y] = \text{BLACK}$ 
16      then RBTREE-DELETE-FIXUP( $T, x$ )
17  return  $y$ 

```

```

RBTREE-DELETE-FIXUP( $T, x$ )
1  while  $x \neq \text{root}[T]$  and  $\text{color}[x] = \text{BLACK}$ 
2      do if  $x = \text{left}[p[x]]$ 
3          then  $w \leftarrow \text{right}[p[x]]$ 
4              if  $\text{color}[w] = \text{RED}$ 
5                  then  $\text{color}[w] \leftarrow \text{BLACK}$ 
6                      $\text{color}[p[x]] \leftarrow \text{RED}$ 
7                     RBTREE-LEFT-ROTATE( $T, p[x]$ )
8                      $w \leftarrow \text{right}[p[x]]$ 
9                 if  $\text{color}[\text{left}[w]] = \text{BLACK}$  and  $\text{color}[\text{right}[w]] = \text{BLACK}$ 
10                    then  $\text{color}[w] \leftarrow \text{RED}$ 
11                     $x \leftarrow p[x]$ 
12                else if  $\text{color}[\text{right}[w]] = \text{BLACK}$ 

```

```

13         then  $color[left[w]] \leftarrow \text{BLACK}$ 
14              $color[w] \leftarrow \text{RED}$ 
15             RBTREE-RIGHT-ROTATE( $T, w$ )
16              $w \leftarrow right[p[x]]$ 
17              $color[w] \leftarrow color[p[x]]$ 
18              $color[p[x]] \leftarrow \text{BLACK}$ 
19              $color[right[w]] \leftarrow \text{BLACK}$ 
20             RBTREE-LEFT-ROTATE( $T, p[x]$ )
21              $x \leftarrow root[T]$ 
22         else Mirror opposite of “then” clause
23      $color[x] \leftarrow \text{BLACK}$ 

```