Laboratory Module 6

Red-Black Trees

Purpose:

- ***understand the notion of red-black trees***
- ***to build, in C, a red-black tree***

1 Red-Black Trees

1.1 General Presentation

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 way nodes can be colored on any path from the root to a leaf, red-black trees ensure that no such path is more than twice as long as any other, so that the tree is approximately balanced.

Red-black trees are a popular alternative to the AVL tree, due to the fact that a single top-down pass can be used during the insertion and deletion routines. This approach contrasts with an AVL tree, in which a pass down the tree is used to establish the insertion point and a second pass up the tree is used to update heights and possibly rebalance. As a result, a careful nonrecursive implementation of the red-black tree is simpler and faster than an AVL tree implementation. As on AVL trees, operations on red-black trees take logarithmic worst-case time.

A binary search tree is a red-black tree if it satisfies the following red-black properties:

1. Every node is either red or black
2. The root is black
3. Every leaf (NULL) is black
4. If a node is red, then both its children are black
5. For each node, all paths from the node to descendant leaves contain the same number of black nodes.

Each node of the tree contain the fields: color, key, left, right, and parent. A red-black tree's node structure would be:

```
struct RedBlackNode{
   int key;
   enum { red, black } color;
   RedBlackNOde *left, *right, *parent;
};
```

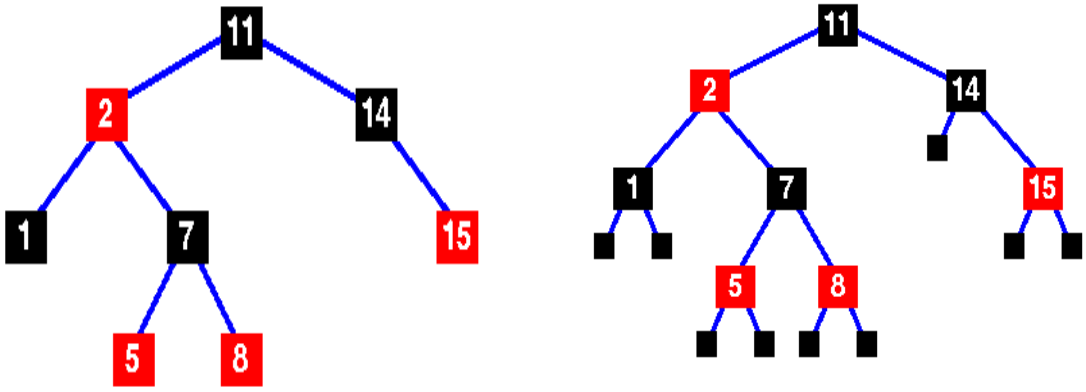
If a child or the parent of a node does not exist, the corresponding pointer field of the node contains the value NULL. We shall regard these NULL's as being pointers to external nodes (leaves) of the binary tree.

The number of black nodes on any path from, but not including, a node x to a leaf is called the black-height of a node, denoted *bh(x)*. We can prove the following lemma:

Lemma: A red-black tree with n internal nodes has height at most 2log(n+1).

This demonstrates why the red-black tree is a good search tree: it can always be searched in O(log n) time. As with heaps, additions and deletions from red-black trees destroy the red-black property, so we need to restore it. To do this we need to look at some operations on red-black trees.

In figure 1 there is also a basic red-black tree with the sentinel nodes added. Implementations of the red-black tree algorithms will usually include the sentinel nodes as a convenient means of flagging that you have reached a leaf node. They are the NULL black nodes of property 3.



a) b)

Figure 1. a) Sample Red-Black tree b) Red-black tree with sentinel nodes

1.2 Operations on Red-Black Trees

1.2.1 Rotations

A rotation is a local operation in a search tree that preserves in-order traversal key ordering. We change the pointer structure trough rotation, which is a local operation in a search tree that preserves the binary-search-tree property. The figure illustrated below shows the 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 NULL; x may be any node in the tree whose right child is not NULL. 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.

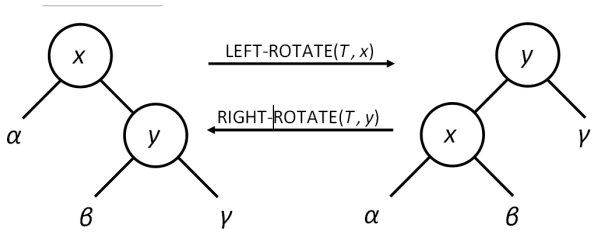


Figure 2. Rotations in a Red-Black tree

The pseudocode for LEFT-ROTATE and LEFT-ROTATE assumes that right assumes that x->right≠ NULL and that the root's parent is NULL.

LEFT-ROTATE(T, x)

```
y ← x->right
x->right← y->left
y->left->p ← x
y->p ← x->p
if      x->p = Null
        then    T->root ← y
        else if x = x->p->left
                then    x->p->left ← y
                else    x->p->right ← y
y->left ← x
x->p ← y
```

RIGHT-ROTATE(T, x)

```
y ← x->left
x->left← y->right
y->right->p ← x
y->p ← x->p
if      x->p = Null
        then    T->root ← y
        else if x = x->p->right
                then    x->p->right ← y
                else    x->p->left ← y
y->right ← x
x->p ← y
```

1.2.2 Insertion

Insertion begins by adding the node much as binary search tree insertion does and by coloring it red. Whereas in the binary search tree, we always add a leaf, in the red-black tree leaves contain no information, so instead we add a red interior node, with two black leaves, in place of an existing black leaf.

Insertion of a node into an n-node red-black tree can be accomplished in O(lg n) tine. We use a slightly modified version of the insertion procedure 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.

```
RB-INSERT(T, z)
        y ← null
        x ← T->root
        while x ≠ null
                do y ← x
                        if z->key < x->key
                                then x ← x->left
                                else x ← x->right
        z->p ← y
        if y = null
                then T->root ← z
                else if z->key < y->key
                        then y->left ← z
                        else y->right ← z
        z->left ← null
        z->right ← null
        z->color ← RED
        RB-INSERT-FIXUP(T, z)
```

There are some differences between the insertion procedure in binary search trees and RB-INSERT. The color of *z* node is set firstly to red. Because coloring z red may cause a violation of one of the red-black properties, we call RB-INSERT-FIXUP(T, z) in last line of RB-INSERT to restore the red-black properties.

```
RB-INSERT-FIXUP(T, z)
while  z->p->color  = RED
        do if z->p =  z->p->p->left
                then y ← z->p->p->right
                        if y->color = RED
                                then    z->p->color ← BLACK                     Case 1
                                        y->color ← BLACK                        Case 1
                                        z->p->p->color ← RED                    Case 1
                                        z ← z->p->p                             Case 1
                                else if z =  z->p->right
                                        then    z ← z->p                        Case 2
                                                LEFT-ROTATE(T, z)               Case 2
                                                z->p->color ← BLACK             Case 3
                                                z->p->p->color ← RED            Case 3
                                                RIGHT-ROTATE(T, z->p->p)        Case 3
                        else (same as then clause with "right" and "left" exchanged)
T->root->color ← BLACK
```

To understand how RB-INSERT-FIXUP works, we shall break our examination of the code into three major steps. First, we shall determine what violations of the red-black properties are introduced in RB-INSERT when the node z is inserted and colored red. Second, we shall examine the overall goal of the

while loop. Finally, we shall explore each of the three cases into which the while loop is broken and see how they accomplish the goal.
The figures below show how RB-INSERT-FIXUP operates on a sample red-black tree.

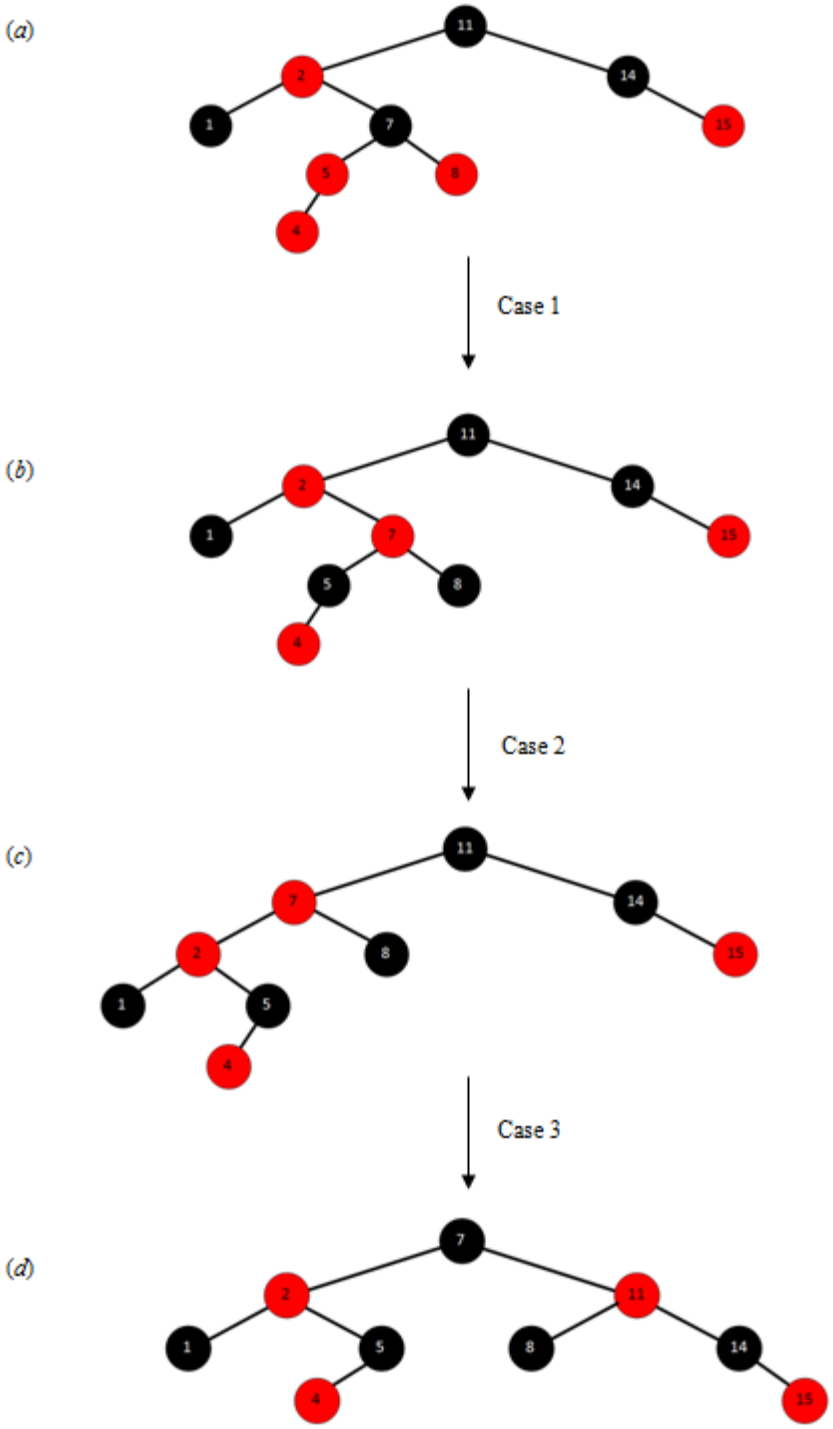


Figure 3. Cases of rotations in a Red-Black tree

Since z (node with key 4 in Figure 3.a) and its parent z->p are both red (node with key 5 in Figure 3.a), a violation of property 4 occurs. Since z's uncle y (node with key 8 in Figure 3.a) is red, case 1 in the code can be applied. Nodes are recolored and the pointer z is moved up the tree, resulting in the tree shown in Figure 3.b.

Once again, z (node with key 7 in Figure 3.b) and its parent (node with key 2 in Figure 3.b) are both red, but z's uncle y (node with key 14 in Figure 3.b) is black. Since z is the right child of z->p, case 2 can be applied. A left rotation is performed, and the tree that results is shown in Figure 3c. Now z is the left child of its parent, and case 3 can be applied. A right rotation yields the tree in Figure 3d, which is a legal red-black tree.

The red-black properties that can be violated upon the call to RB-INSERT-FIXUP need to be analysed. Property 1 certainly continues to hold, as does property 3, since both children of the newly inserted red node are the sentinel NULL. Property 5, which says that the number of black nodes is the same on every path from a given node, is satisfied as well, because node z replaces the (black) sentinel, and node z is red with sentinel children. Thus, the only properties that might be violated are property 2, which requires the root to be black, and property 4, which says that a red node cannot have a red child. Both possible violations are due to z being colored red. Property 2 is violated if z has been inserted.

At the start of each iteration of the loop,

a. Node z is red.
b. If z->p is the root, then z->p is black.
c. If there is a violation of the red-black properties, there is at most one violation, and it is of either property 2 or property 4. If there is a violation of property 2, it occurs because z is the root and is red. If there is a violation of property 4, it occurs because both z and z->p are red.

Part (c), which deals with violations of red-black properties, is more central to showing that RB-INSERT-FIXUP restores the red-black properties than parts (a) and (b), which we use along the way to understand situations in the code. Because we will be focusing on node z and nodes near it in the tree, it is helpful to know from part (a) that z is red. We shall use part (b) to show that the node z->p->p exists then we reference it later.

Recall that we need to show that a loop invariant is true prior to the first iteration of the loop, that each iteration maintains the loop invariant, and that the loop invariant gives us a useful property at loop termination.

We start with the initialization and termination arguments. Then, as we examine how the body of the loop works in more detail, we shall argue that the loop maintains the invariant upon each iteration of the loop: the pointer z moves up the tree, or some rotations are performed and the loop terminates.

Initialization: Prior to the first iteration of the loop, we started with a red-black tree with no violations, and we added a red node z. We show that each part of the invariant holds at the time that RB-INSERT-FIXUP is called:

a. When RB-INSERT-FIXUP is called, z is the red node that was added.
b. If z->p is the root, then z->p started out black and did not change prior to the call of RB-INSERT-FIXUP.
c. We have already seen that properties 1,3, and 5 hold when RB-INSERT-FIXUP is called.

If there is a violation of property 2, then the red root must be the newly added node z, which is the only internal node in the tree. Because the parent and both children of z are the sentinel, which is black, there is not also a violation of property 4. Thus, this violation of property 2 is the only violation of red-black properties in the entire tree.

If there is a violation of property 4, then because the children of node z are black sentinels and the tree had no other violations prior to z being added, the violations must be because both z and z->p are red. Moreover, there are not other violations of red-black properties.

Termination: When the loop terminates, it does so because z->p is black. (If z is the root, then z->p is the sentinel NULL, which is black.) Thus, there is no violation of property 4 at loop termination. By the loop invariant, the only property that might fail to hold is property 2. Last line restores this property, too, so, that when RB-INSERT-FIXUP terminates, all the red-black properties hold.

Maintenance: There are actually six cases to consider in the while loop, but three of them are symmetric to the other three, depending on whether z's parent z->p is a left child or a right child of z's grandparent z->p->p, which is determined in line 2. We have given the code only for the situation in which z->p is a left child. The node z->p->p exists, since by part (b) of the loop invariant, if z->p is the root, then

z->p is black. Since we enter a loop iteration only if z->p is red, we know that z->p cannot be the root. Hence, z->p->p exists.

Case 1 is distinguished from cases 2 and 3 by the color of z's parent's sibling, or "uncle". Line 3 makes y point to z's uncle z->p->p->right, and a test is made in line 4. If y is red, then case 1 is executed. Otherwise, control passes to cases 2 and 3. In all three cases, z's grandparent z->p->p is black, since its parent z->p is red, and property 4 is violated only between z and z->p.

Case 1:z's uncle is red

The below figure shows the situation for case 1. Case 1 is executed when both z->p and y are red. Since z->p->p is black, we can color both z->p and y black, thereby fixing the problem of z and z->p both being red, and color z->p->p red, thereby maintaining property 5. We then repeat the while loop with z->p->p as the new node z. The pointer z moves up two levels in the tree.

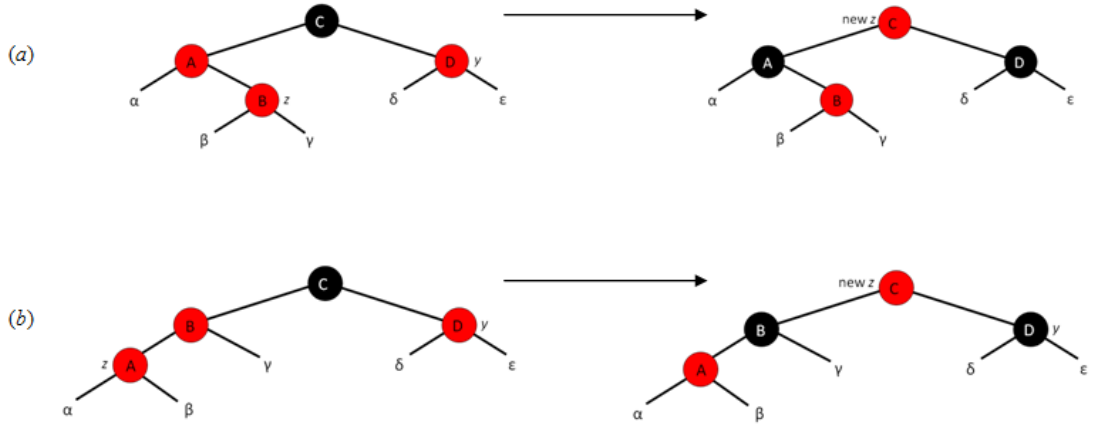


Figure 4. Cases 1 of rotations in a Red-Black tree

In case 1 of the procedure RB-INSERT, Property 4 is violated, since z and its parent z->p are both red. The same action is taken whether (Figure 4.a) z is a right child or (Figure 4.b) z is a left child. Each of the subtrees α,β,γ,δ and ε has a black root, and each has the same black-height. The code for case 1 changes the colors of some nodes, preserving property 5: all downward paths from a node to a leaf have the same number of blacks. The while loop continues with node z's grandparent z->p->p as the new z. Any violation of property 4 can now occur between the new z, which is red, and its parent, if it is red as well.

Now we show that case 1 maintains the loop invariant at the start of the next iteration. We use z to denote node z in the current iteration, and z' z->p->p to denote the node z at the test in line 1 upon the next iteration

a. Because this iteration colors z->p->p red, node z' is red at the start of the next iteration.
b. The node z'->p is z->p->p->p in this iteration, and the color of this node does not change. If this node is the root, it was black prior to this iteration, and it remains black at the start of the next iteration.
c. We have already argued that case 1 maintains property 5, and it clearly does not introduce a violation of properties 1 or 3.

If node z' is the root at the start of the next iteration, then case 1 corrected the lone violation of property 4 in this iteration. Since z' is red and it is the root, property 2 becomes the only one that is violated, and this violation is due to z'.

If node z' is not the root at the start of the next iteration, then case 1 has not created a violation of property 2.Case 1 corrected the lone violation of property 4 that existed at the start of this iteration. It then made z' red and left z'->p alone. If z'->p was black, there is no violation of property 4. If z'->p was red, coloring z' red created one violation of property 4 between z' and z'->p.

Case 2: z's uncle y is black and z is a right child
Case 3: z's uncle y is black and z is a left child

In cases 2 and 3, the color of z's uncle y is black. The two cases are distinguished by whether z is a right or left child of z->p. Lines 10 -11 constitute case 2, which is shown below together with case 3. In case 2, node z is a right child of its parent. We immediately use a left rotation to transform the situation into case 3 (lines 12-14), in which node z is a left child. Because both z and z->p are red, the rotation affects neither the black-height of nodes nor property 5. Whether we enter case 3 directly or through case 2, z's uncle y is black, since otherwise we would have executed case 1. Additionally, the node z->p->p exists, since we have argued that this node existed at the time that lines 2 and 3 were executed, and after moving z up one level in line 10 and then down one level in line 11, the identity of z->p->p remains unchanged. In case 3, we execute some color changes and a right rotation, which preserve property 5, and then, since we no longer have two red nodes in a row, we are done. The body of the while loop is not executed another time, since z->p is now black.

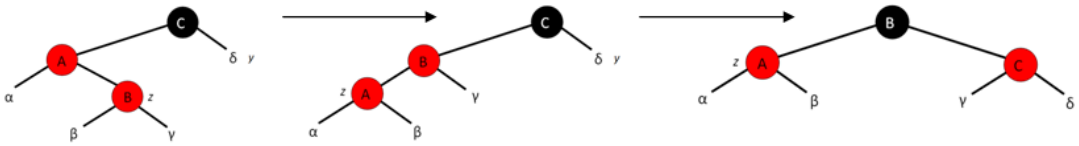


Figure 5. Cases 2 and 3 of rotations in a Red-Black tree

Cases 2 and 3 of the procedure RB-INSERT. As in case 1, property 4 is violated in either case 2 or case 3 because z and its parent z->p are both red. Each of the subtrees α, β, γ, and δ has a black root (α, β and γ from property 4, and δ because otherwise we would be in case 1), and each has the same black-height. Case 2 is transformed into case 3 by a left rotation, which preserves property 5: all downward paths from a node to a leaf have the same number of blacks. Case 3 causes some color changes and a right rotation, which also preserve property 5. The while loop then terminates, because property 4 is satisfied: there are no longer two red nodes in a row.

Now we show that cases 2 and 3 maintain the loop invariant. (As we have just argues, z->p will be black upon the next test in line 1, and the loop body will not execute again.)

a. Case 2 makes z point to z->p, which is red. No further change to z or its color occurs in cases 2 and 3.
b. Case 3 makes z->p black, so that if z->p is the root at the start of the next iteration, it is black.
c. As in case 1, properties 1,3, and 5 are maintained in cases 2 and 3.

Since node z in not the root in cases 2 and 3, we know that there is no violation of property 2. Cases 2 and 3 do not introduce a violation of property 2, since the only node that is made red becomes a child of a black node by the rotation in case 3.

Cases 2 and 3 correct the lone violation of property 4, and they do not introduce another violation.

Having shown that each iteration of the loop maintains the invariant, we have shown that RB-INSERT-FIXUP correctly restores the red-black properties.

1.2.3 Deletion

Like the other basic operations on an n-node red-black tree, deletion of a node take time O(lg n). Deleting a node from a red-black tree is only slightly more complicated than inserting a node.

After splicing out a node the procedure RB-DELETE calls an auxiliary procedure RB-DELETE-FIXUP that changes colors and performs rotations to restore the red-black properties.

A call to RB-DELETE-FIXUP is made in lines 16 – 17 if y is black. If y is red, the red-black properties still hold when y is spliced out, for the following resons:

- No black-heights in the tree have changed,
- No red nodes have been made adjacent,
- Y could not have been the root if it was red, so the root remains black.

```
RB-DELETE(T, z)
    if z->left = null or z->right = null
        then y ← z
        else y ← TREE-SUCCESSOR(z)
    if y->left ≠ null
        then x ← y->left
        else x ← y->right
    x->p ← y->p
    if y->p = null
        then T->root ← x
        else if y = y->p->left
                then y->p->left ← x
                else y->p->right ← x
    if y 3≠ z
        then z->key ← y->key
                copy y's satellite data into z
    if y->color = BLACK
        then RB-DELETE-FIXUP(T, x)
    return y
```

The node x passed to RB-DELETE-FIXUP is one of two nodes: either the node that was y's sole child before y was spliced out if y had a child that was not the sentinel NULL, or it y had no children, x is the sentinel NULL. In the latter case, the unconditional assignment in line 7 guarantees that x's parent is now the node that was previously y's parent, whether x is a key-bearing internal node or the sentinel NULL.

We can now examine how the procedure RB-DELETE-FIXUP restores the red-black properties to the search tree.

```
RB-DELETE-FIXUP(T, x)
while x ≠ T->root and x->color = BLACK
    do if x = x->p->left
        then    w ← x->p->right
                if w->color = RED
                    then    w->color ← BLACK                          Case 1
                            x->p->color ← RED                         Case 1
                            LEFT-ROTATE(T, x->p)                      Case 1
                            w ← x->p->right                           Case 1
                if w->left->color = BLACK and w->right->color = BLACK
                    then    w->color ← RED                            Case 2
                            x  ←  x->p                                Case 2
                    else if w->right->color = BLACK
                            then    w->left->color ← BLACK            Case 3
                                    w->color ← RED                    Case 3
                                    RIGHT-ROTATE(T, w)                Case 3
                                    w ← x->p->right                   Case 3
                            w->color ← x->p->color                    Case 4
                            x->p->color ← BLACK                       Case 4
                            w->right->color ← BLACK                   Case 4
                            LEFT-ROTATE(T, x->p)                      Case 4
                            x ← T->root                               Case 4
        else (same as then clause with "right" and "left" exchanged)
x->color ← BLACK
```

If the spliced-out node y in RB-DELETE is black, three problems may arise. First, if y had been the root and a red child of y becomes a new root, we have violated property 2. Second, if both x and y->p were red, then we have violated property 4. Third, y's removal causes any path that previously contained y to have one fewer black node. Thus, property 5 is now violated by any ancestor of y in the tree. We can

correct this problem by saying that node x has an "extra" black. That is, if we add 1 to the count of black nodes on any path that contains x, then under this interpretation, property 5 holds. When we splice out the black node y, we "push" its blackness onto its child. The problem is that now node x is neither red nor black, thereby violating property 1. Instead, node x is either "doubly black" or "red-and-black" and it contributes either 2 or 1, respectively, to the count of black nodes on paths containing x. The color attribute of x will still be either RED (if x is red-and-black) or BLACK (if x is doubly black). In other words, the extra black on a node is reflected in x's pointing to the node rather than in the color attribute.

Within the while loop, x always points to a nonroot doubly black node. We determine in line 2 whether x is a left child or a right child of its parent x->p. We maintain a pointer w to the sibling of x. Since node x is doubly black, node w cannot be NULL; otherwise, the number of blacks on the path from x->p to the (singly black) leaf w would be smaller than the number on the path from x->p to x.

The four cases in the code are illustrated below. Before examining each case in detail, let's look more generally at how we can verify that the transformation in each of the cases preserves property 5. The key idea is that in each case the number of black nodes (including x's extra black) from (and including) the root of the subtree shown to each of the subtrees α,β, ..., ζ is preserved by the transformation. Thus, if property 5 holds prior to the transformation, it continues to hold afterward. For example, in figure (a) which illustrates case 1, the number of black nodes from the root to any ofγ, δ, and ε, is ζ, both before and after the transformation. In figure (b) the counting must involve the value c of the color attribute of the root of the subtree shown, which can be either RED or BLACK. If we define count(RED) = 0 and count(BLACK) = 1, then the number of black nodes from the root to α is 2 + count(c), both before and after the transformation. In this case, after the transformation on, the new node x has color attribute c, but this node is really either red-and-black (if c = RED) or doubly black (if c = BLACK). The other cases can be verified similarly.

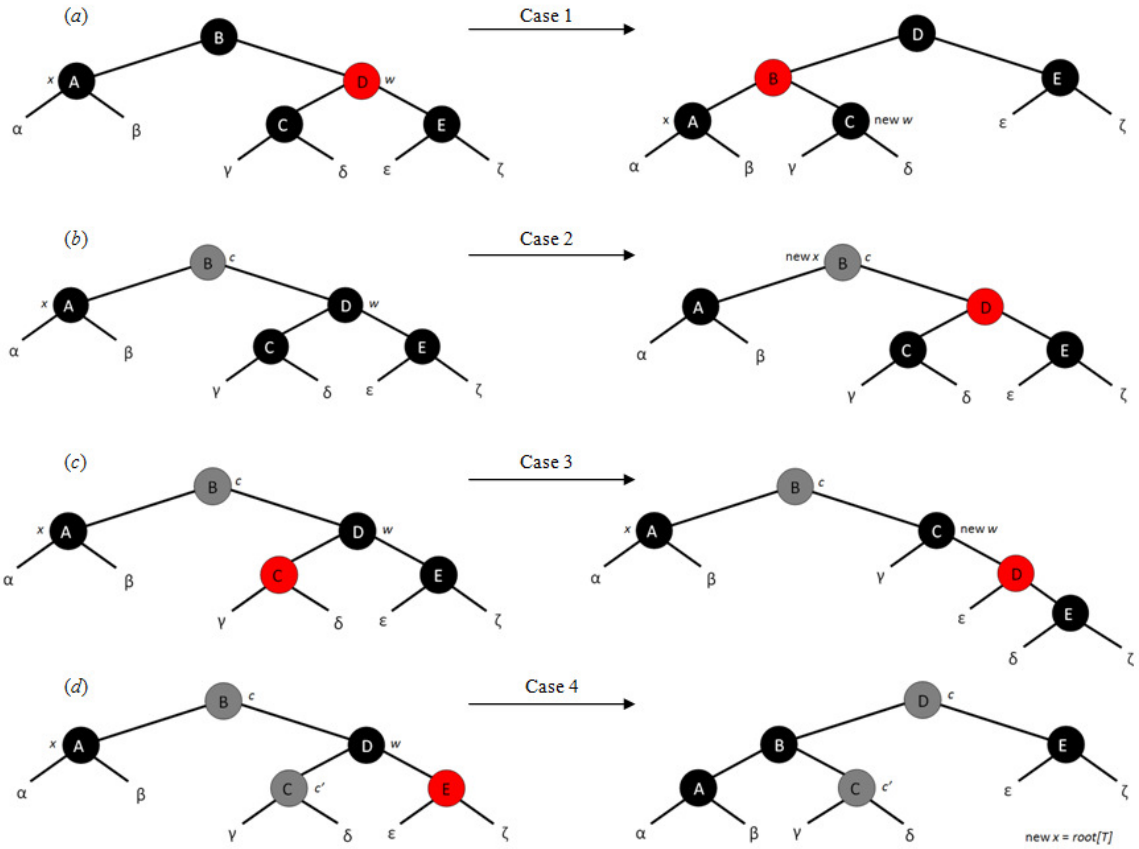


Figure 6. Cases of rotations when deleting a node from a Red-Black tree

The cases in the while loop of the procedure RB-DELETE-FIXUP. Darkened nodes have color attributes BLACK, heavily shaded nodes color attributes RED, and lightly shaded nodes nodes have color attributes represented by c and c', which may be either RED or BLACK. The letters α,β, ..., ζ represent

arbitrary subtrees. In each case, the configuration on the left is transformed into the configuration on the right by changing some colors and/or performing a rotations. Any node pointed by x has an extra black and is either doubly black or red-and-black. The only case that causes the loop to repeat is case 2. (a) Case 1 is transformed to case 2,3 or 4 by changing the colors of nodes B and D and performing a left rotation. (b) In case 2, the extra black represented by the pointer x is moved up the tree by coloring node D red and setting x to point to node B. If we enter case 2 through case 1, the while loop terminates because the new node x is red-and-black, and therefore the value c of its color attribute is RED. (c) Case 3 is transformed to case 4 by exchanging the colors of nodes C and D and performing a right rotation. (d) In Case 4, the extra black represented by x can be removed by changing some colors and performing a left rotation (without violating the red-black properties), and the loop terminates.

Case 1: x's sibling is red

Case 1 (lines 5–8 of RB-DELETE-FIXUP and figure (a) occurs when node w, the sibling of node x, is red. Since w must have black children, we can switch the colors of w and x->p and then perform a left-rotation on x->p without violating any of the red-black properties. The new sibling of x, which is one of w's children prior to the rotation, is now black, and thus we have converted case 1 into case 2, 3, or 4.
Cases 2,3 and 4 occur when node w is black; they are distinguished by the colors of w's children.

Case 2: x's sibling w is black, and both w's children are black.

In case 2 (lines 10–11 of RB-DELETE-FIXUP and figure (b), both of w's children are black. Since w is also black, we take one black off both x and w, leaving x with only one black and leaving w red. To compensate for emoving one black from x and w, we would like to add an extra black to x->p, which was originally either red or black. We do so by repeating the while loop with x->p as the new node x. Observe that if we enter case 2 through case 1, the new node x is red-and-black, since the original x->p was red. Hence, the value c of the color attribute of the new node x is RED, and the loop terminates when it tests the loop condition.The new node x is then colored (singly) black in line 23.

Case 3: x's sibling w is black, w's left child is red, and w's right child is black

Case 3 (lines 13–16 and figure (c) occurs when w is black, its left child is red, and its right child is black. We can switch the colors of w and its left child w->left and then perform a right rotation on w without violating any of the red-black properties. The new sibling w of x is now a black node with a red right child, and thus we have transformed case 3 into case 4.

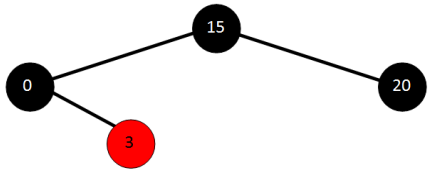
Case 4: x's sibling w is black, and w's right child is red

Case 4 (lines 17–21 and figure (d) occurs when node x's sibling w is black and w's right child is red. By making some color changes and performing a left rotation on x->p, we can remove the extra black on x, making it singly black, without violating any of the red-black properties. Setting x to be the root causes the while loop to terminate when it tests the loop condition.

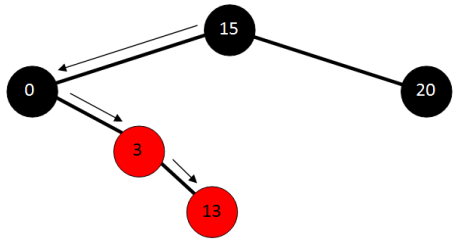
1.2.4 Sample operations in a Red-Black Tree

Insertions:

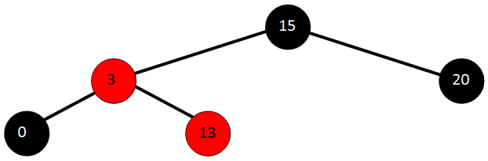
We start from the following tree:



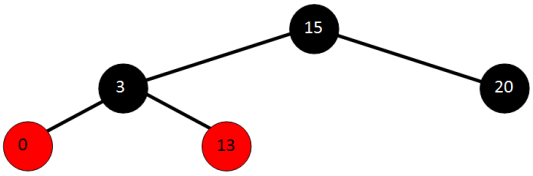
We add number 13 to the tree.



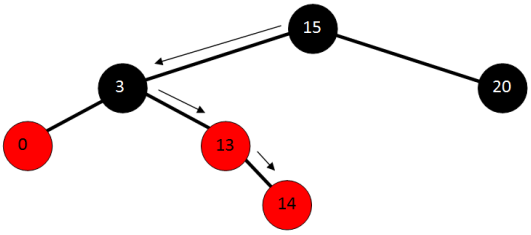
In the next step a simple left rotation takes place. So that number 3 goes up one level and 0 becomes the left child of 3.



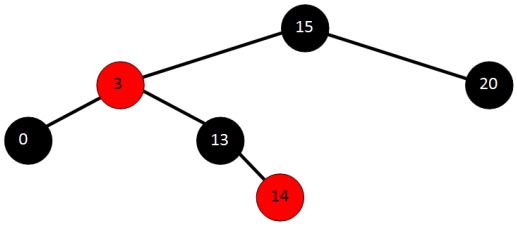
In the next step some color adjustments are made so that the color properties are not violated.



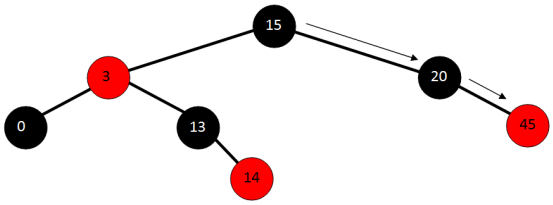
Next we add the number 14 to our tree.



In the next step a color adjustment takes place so that the color properties are not violated.

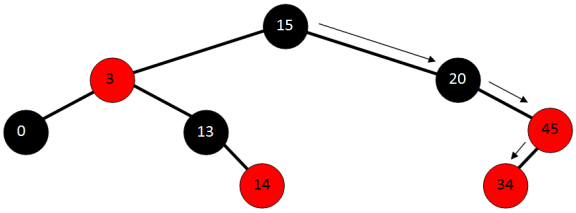


We now add number 45 to the tree.

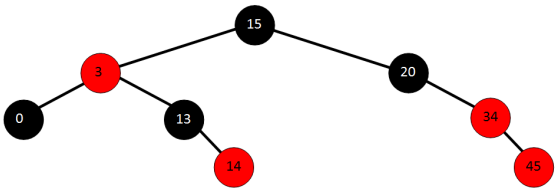


The insertion of this node does not violate any of the red-black tree proprieties, so there are no adjustments to be made.

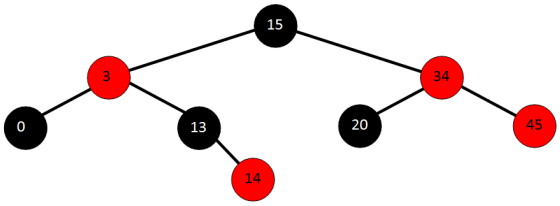
We now add number 34 to the tree.



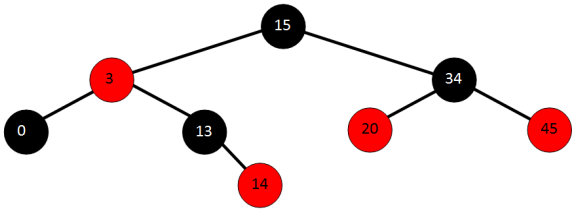
In the next step a right rotation takes place, in which 34 goes up one level and 45 becomes 34's right child.



In the next step a left rotation takes place, which results in 34 going up yet another level and the number 20 becoming 34's left child.

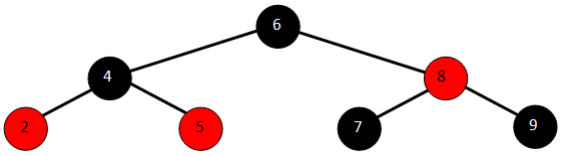


In the next step some color adjustments are made so that the color properties are not violated.

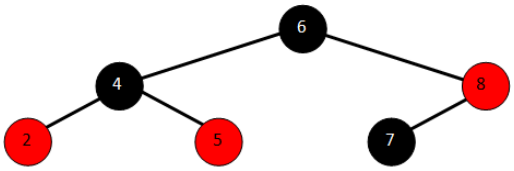


Deletions:

We start from the following tree.

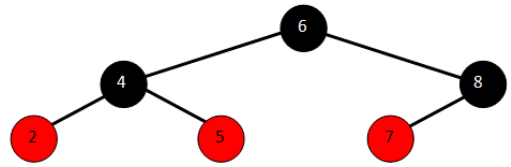


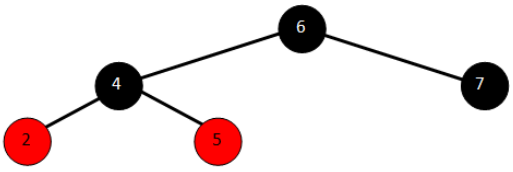
We delete number 9.



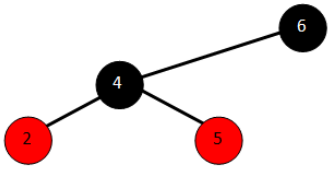
After deleting number 9 the node with key 8 remains with two right side black children that belonged to number 9. (Don't forget every leaf is black, but those leafs are NULL and are not represented in these drawings,. Those leaf nodes are the sentinels, in our case NULL nodes with black color values.)

In order to fix this violation we recolor the tree as follows:
In the next step we delete node number 8. By deleting number 8 no double black node emerges as in the previous example. So the structure of the tree doesn't suffer modification.

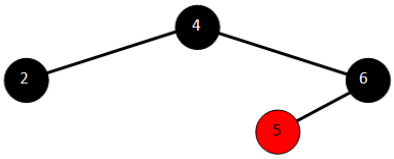




In the next step we delete node number 7. By deleting node number 7 we end up with a tree looking like this:



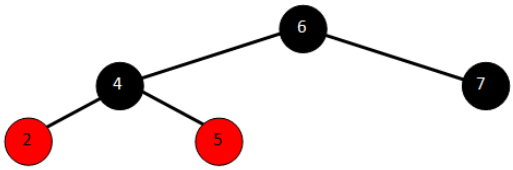
The tree obtained is unbalanced therefore we need no restructure it, and do so by a right rotation of node number 4. Also some re-coloring is needed after that. The final tree looks like this:



2. Assignments

1) Write a program that creates and manages a Red-Black tree. The program must implement the following operations: creation, insertion, deletion and tree display. The program should present a meniu where user may choose from implemented options.

2) Consider the a sample Red-Black tree with 5 nodes:



Perform the following operations on the above-mentioned B-tree:

- Insert key 20;
- Insert key 16;
- Insert key 17;
- Delete key 6;
- Delete key 4;
- Delete key 20;

For each insertion/deletion operation there must be presented the following:

- Initial position of the key into the Red-Black tree
- Analysis regarding the violated properties
- The necessary rotations and collor assignments
- The final shape of the tree