Red-black tree, not so complex as it was thought

Larry LIU Xinyu [*]

September 30, 2017

1 Introduction

1.1 Exploit the binary search tree

We showed the power of using binary search tree as a dictionary to count the occurrence of every word in a book in previous chapter.

One may come to the idea to feed a yellow page book [1] to a binary search tree, and use it to look up the phone number for a contact.

By modifying a bit of the program for word occurrence counting yields the following code.

```
public static void main(String[] args) {
    Map<String, String> dict = new TreeMap<>();
    try {
        Scanner sc = new Scanner(new File("yp.txt"));
        while (sc.hasNext()) {
            String name = sc.next();
            String phone = sc.next();
            dict.put(name, phone);
        }
        sc = new Scanner(System.in);
        while (true) {
            System.out.print("name: ");
            String name = sc.next();
            String phone = dict.get(name);
            System.out.println(phone == null ? "not found" : ("phone: " + phone));
        }
    } catch (FileNotFoundException e) {
        System.out.println(e);
    }
}
```

[*] **Larry LIU Xinyu**
Email: liuxinyu95@gmail.com

[1] A telephone number contact list book

This program works well. However, if you replace the TreeMap provided in the standard library with the binary search tree we developed in previous chapter, the performance will be bad, especially when you search some names such as Zara, Zed, Zulu.

This is because the content of yellow page is typically listed in lexicographic order. Which means the name list is in increase order. If we try to insert a sequence of number 1, 2, 3, ..., n to a binary search tree, we will get a tree like in Figure 1.

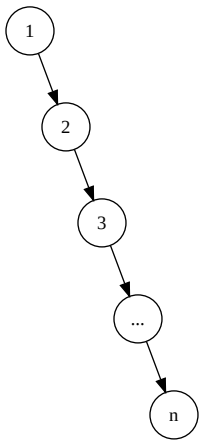


Figure 1: unbalanced tree

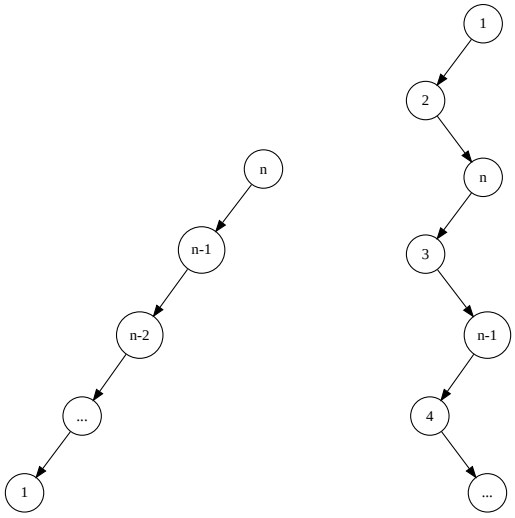
This is extreme unbalanced binary search tree. The looking up performs $O(h)$ for a tree with height h. In balanced case, we benefit from binary search tree by $O(\lg n)$ search time. But in this extreme case, the search time downgraded to $O(n)$. It's no better than a normal link-list.

Exercise 1

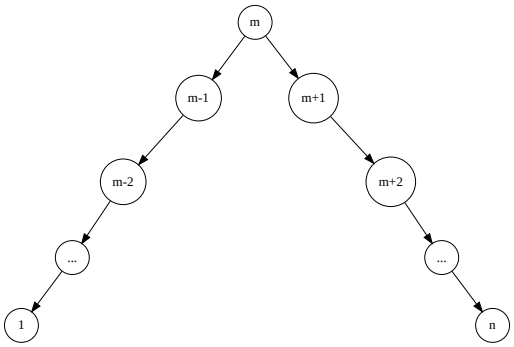
- For a very big yellow page list, one may want to speed up the dictionary building process by two concurrent tasks (threads or processes). One task reads the name-phone pair from the head of the list, while the other one reads from the tail. The building terminates when these two tasks meet at the middle of the list. What will be the binary search tree looks like after building? What if you split the list more than two and use more tasks?
- Can you find any more cases to exploit a binary search tree? Please consider the unbalanced trees shown in figure 2.

1.2 How to ensure the balance of the tree

In order to avoid such case, we can shuffle the input sequence by randomized algorithm, such as described in Section 12.4 in [1]. However, this method doesn't



(a) (b)



(c)

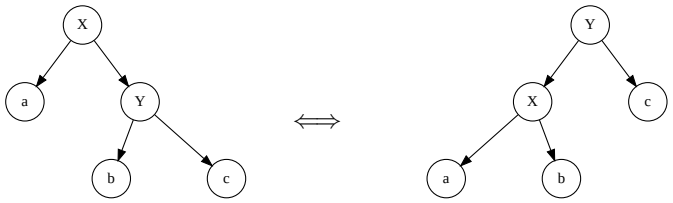
Figure 2: Some unbalanced trees

always work, for example the input is fed from user interactively, and the tree need to be built and updated online.

There are many solutions people found to make binary search tree balanced. Many of them rely on the rotation operations to the binary search tree. Rotation operations change the tree structure while maintain the ordering of the elements. Thus it can be used to improve the balance property of the binary search tree.

In this chapter, we'll first introduce the red-black tree. It is one of the most popular and widely used self-adjusting balanced binary search tree. In next chapter, we'll introduce another intuitive solution, the AVL tree. In later chapter about binary heaps, we'll show another interesting tree called splay tree, which can gradually adjust the the tree to make it more and more balanced.

1.3 Tree rotation



(a) (b)

Figure 3: Tree rotation, 'rotate-left' transforms the tree from left side to right side, and 'rotate-right' does the inverse transformation.

Tree rotation is a set of operations that can transform the tree structure without changing the in-order traverse result. It based on the fact that for a specified ordering, there are multiple binary search trees correspond to it. Figure 3 shows the tree rotation. For a binary search tree on the left side, left rotate transforms it to the tree on the right, and right rotate does the inverse transformation.

Although tree rotation can be realized in procedural way, there exists simple functional definition by using pattern matching. Denote the non-empty tree as $T = (T_l, k, T_r)$, where k is the key, and T_l, T_r are left and right sub-trees.

$$rotate_l(T) = \begin{cases} ((a, X, b), Y, c) & : \quad T = (a, X, (b, Y, c)) \\ T & : \quad otherwise \end{cases} \tag{1}$$

$$rotate_r(T) = \begin{cases} (a, X, (b, Y, c)) & : \quad T = ((a, X, b), Y, c)) \\ T & : \quad otherwise \end{cases} \tag{2}$$

To perform tree rotation imperatively, we need set all fields of the node as the following.

```
1: function LEFT-ROTATE(T, x)
2:     p ← PARENT(x)
3:     y ← RIGHT(x)                                  ▷ Assume y ≠ NIL
4:     a ← LEFT(x)
5:     b ← LEFT(y)
6:     c ← RIGHT(y)
7:     REPLACE(x, y)
8:     SET-CHILDREN(x, a, b)
9:     SET-CHILDREN(y, x, c)
10:    if p = NIL then
11:        T ← y
12:    return T

13: function RIGHT-ROTATE(T, y)
14:    p ← PARENT(y)
15:    x ← LEFT(y)                                   ▷ Assume x ≠ NIL
16:    a ← LEFT(x)
17:    b ← RIGHT(x)
18:    c ← RIGHT(y)
19:    REPLACE(y, x)
20:    SET-CHILDREN(y, b, c)
21:    SET-CHILDREN(x, a, y)
22:    if p = NIL then
23:        T ← x
24:    return T
```

Where procedure REPLACE(x, y), uses y to replace x.

```
1: function REPLACE(x, y)
2:     if PARENT(x) = NIL then
3:         if y ≠ NIL then PARENT(y) ← NIL
4:     else if LEFT(PARENT(x)) = x then
5:         SET-LEFT(PARENT(x), y)
6:     else
7:         SET-RIGHT(PARENT(x), y)
8:     PARENT(x) ← NIL
```

Procedure SET-CHILDREN assigns a pair of sub-trees as the left and right children of a given node.

```
1: function SET-CHILDREN(x, L, R)
2:     SET-LEFT(x, L)
3:     SET-RIGHT(x, R)

4: function SET-LEFT(x, y)
5:     LEFT(x) ← y
6:     if y ≠ NIL then PARENT(y) ← x
```

```
7: function SET-RIGHT(x, y)
8:     RIGHT(x) ← y
9:     if y ≠ NIL then PARENT(y) ← x
```

Compare the imperative operations with the pattern matching functions, we can found the latter focus on the structure change, while the former focus on the rotation process. As the title of this chapter indicated, red-black tree needn't be so complex as it was thought. Many traditional algorithm text books use the classic procedural treatment to the red-black tree. When insert or delete keys, there are multiple cases with a series of node manipulation. On the other hand, in functional settings, the algorithm turns to be intuitive and simple, although there is some performance overhead.

Most of the content in this chapter is based on Chris Okasaki's work in [2].

2 Definition of red-black tree

Red-black tree is a type of self-balancing binary search tree[4]. [2] By using color changing and rotation, red-black tree provides a very simple and straightforward way to keep the tree balanced.

For a binary search tree, we can augment the nodes with a color field, a node can be colored either red or black. We call a binary search tree red-black tree if it satisfies the following 5 properties([1] pp273).

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 paths from the node to descendant leaves contain the same number of black nodes.

Why this 5 properties can ensure the red-black tree is well balanced? Because they have a key characteristic, the longest path from root to a leaf can't be as 2 times longer than the shortest path.

Consider the 4-th property. It means there can't be two adjacent red nodes. so the shortest path only contains black nodes, any path that is longer than the shortest one has interval red nodes. According to property 5, all paths have the same number of black nodes, it finally ensures there can't be any path taht is 2 times longer than others[4]. Figure 4 shows a red-black tree example.

As all NIL nodes are black, people often omit them when draw red-black tree. Figure 5 gives the corresponding tree that hides all the NIL nodes.

[2] Red-black tree is one of the equivalent form of 2-3-4 tree (see chapter B-tree about 2-3-4 tree). That is to say, for any 2-3-4 tree, there is at least one red-black tree has the same data order.

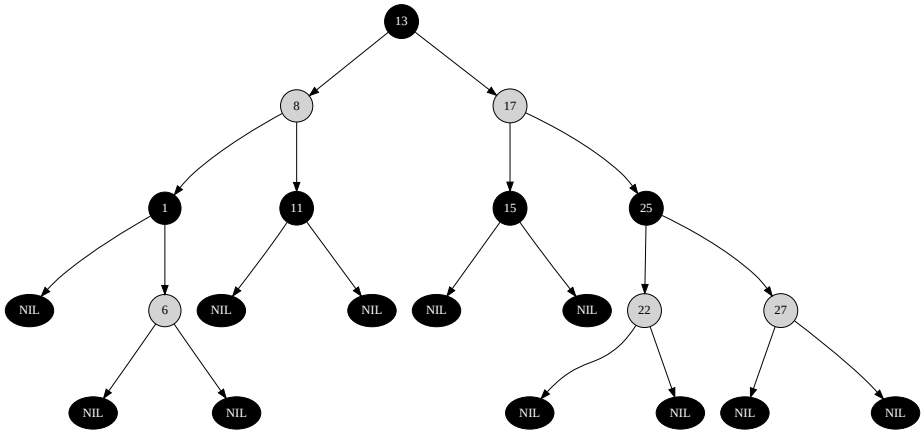


Figure 4: A red-black tree

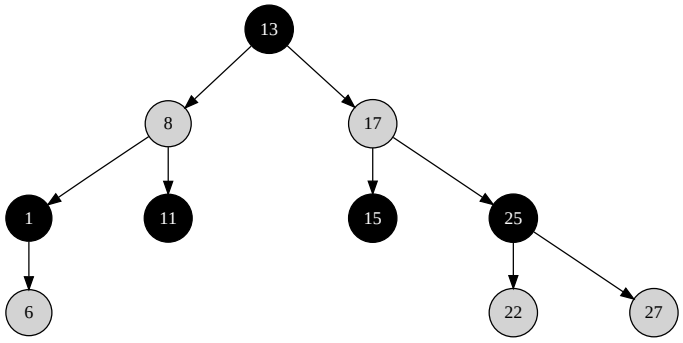


Figure 5: The red-black tree with all NIL nodes hidden.

All read only operations such as search, min/max are as same as in binary search tree. While only the insertion and deletion are special.

As we have shown in word occurrence example, many implementation of set or map container are based on red-black tree. One example is the C++ Standard library (STL)[6].

As mentioned previously, the only change in data layout is that there is color information augmented to binary search tree. This can be represented as a data field in imperative languages such as C++ like below.

```
enum Color {Red, Black};

template <class T>
struct node{
  Color color;
  T key;
  node* left;
  node* right;
  node* parent;
};
```

In functional settings, we can add the color information in constructors, below is the Haskell example of red-black tree definition.

```
data Color = R | B
data RBTree a = Empty
              | Node Color (RBTree a) a (RBTree a)
```

Exercise 2

- Can you prove that a red-black tree with n nodes has height at most $2\lg(n+1)$?

3 Insertion

Inserting a new node as what has been mentioned in binary search tree may cause the tree unbalanced. The red-black properties has to be maintained, so we need do some fixing by transform the tree after insertion.

When we insert a new key, one good practice is to always insert it as a red node. As far as the new inserted node isn't the root of the tree, we can keep all properties except the 4-th one. that it may bring two adjacent red nodes.

Functional and procedural implementation have different fixing methods. One is intuitive but has some overhead, the other is a bit complex but has higher performance. Most text books about algorithm introduce the later one. In this chapter, we focus on the former to show how easily a red-black tree insertion algorithm can be realized. The traditional procedural method will be given only for comparison purpose.

As described by Chris Okasaki, there are total 4 cases which violate property 4. All of them has 2 adjacent red nodes. However, they have a uniformed form after fixing[2] as shown in figure 6.

Note that this transformation will move the redness one level up. During the bottom-up recursive fixing, the last step will make the root node red. According to property 2, root is always black. Thus we need final fixing to revert the root color to black.

Observing that the 4 cases and the fixed result have strong pattern features, the fixing function can be defined by using the similar method we mentioned in tree rotation. Denote the color of a node as $\mathcal{C}$, it has two values: black $\mathcal{B}$, and red$\mathcal{R}$. Thus a non-empty tree can be represented as $T = (\mathcal{C}, T_l, k, T_r)$.

$$balance(T) = \begin{cases} (\mathcal{R}, (\mathcal{B}, A, x, B), y, (\mathcal{B}, C, z, D)) & : \quad match(T) \\ T & : \quad otherwise \end{cases} \tag{3}$$

where function $match()$ tests if a tree is one of the 4 possible patterns as the following.

$$match(T) = \left\{ T = \begin{array}{l} (\mathcal{B}, (\mathcal{R}, (\mathcal{R}, A, x, B), y, C), z, D) \vee \\ (\mathcal{B}, (\mathcal{R}, A, x, (\mathcal{R}, B, y, C), z, D)) \vee \\ (\mathcal{B}, A, x, (\mathcal{R}, B, y, (\mathcal{R}, C, z, D))) \vee \\ (\mathcal{B}, A, x, (\mathcal{R}, (\mathcal{R}, B, y, C), z, D)) \end{array} \right\}$$

With function $balance(T)$ defined, we can modify the previous binary search tree insertion functions to make it work for red-black tree.

$$insert(T, k) = makeBlack(ins(T, k)) \tag{4}$$

where

$$ins(T, k) = \begin{cases} (\mathcal{R}, \phi, k, \phi) & : \quad T = \phi \\ balance((ins(T_l, k), k', T_r)) & : \quad k < k' \\ balance((T_l, k', ins(T_r, k))) & : \quad otherwise \end{cases} \tag{5}$$

If the tree is empty, then a new red node is created, and k is set as the key; otherwise, denote the children and the key as T_l, T_r, and k', we compare k and k' and recursively insert k to one of the children. Function *balance* is called after that, and the root is force to be black finally.

$$makeBlack(T) = (\mathcal{B}, T_l, k, T_r) \tag{6}$$

Summarize the above functions and use language supported pattern matching features, we can come to the following Haskell program.

```
insert t x = makeBlack $ ins t where
    ins Empty = Node R Empty x Empty
    ins (Node color l k r)
        | x < k      = balance color (ins l) k r
        | otherwise = balance color l k (ins r) —[3]
```

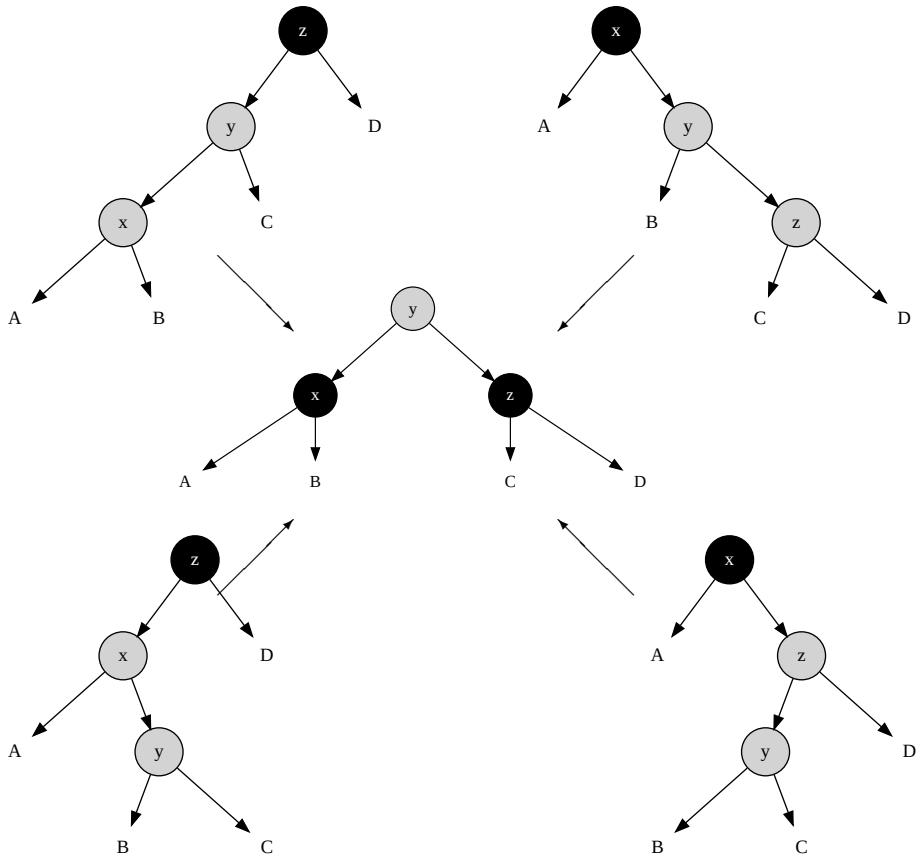


Figure 6: 4 cases for balancing a red-black tree after insertion

```
    makeBlack(Node _ l k r) = Node B l k r

balance B (Node R (Node R a x b) y c) z d =
                Node R (Node B a x b) y (Node B c z d)
balance B (Node R a x (Node R b y c)) z d =
                Node R (Node B a x b) y (Node B c z d)
balance B a x (Node R b y (Node R c z d)) =
                Node R (Node B a x b) y (Node B c z d)
balance B a x (Node R (Node R b y c) z d) =
                Node R (Node B a x b) y (Node B c z d)
balance color l k r = Node color l k r
```

Note that the 'balance' function is changed a bit from the original definition. Instead of passing the tree, we pass the color, the left child, the key and the right child to it. This can save a pair of 'boxing' and 'un-boxing' operations.

This program doesn't handle the case of inserting a duplicated key. However, it is possible to handle it either by overwriting, or skipping. Another option is to augment the data with a linked list[1].

Figure 7 shows two red-black trees built from feeding list 11, 2, 14, 1, 7, 15, 5, 8, 4 and 1, 2, ..., 8. The tree is well balanced even if we input an ordered list.

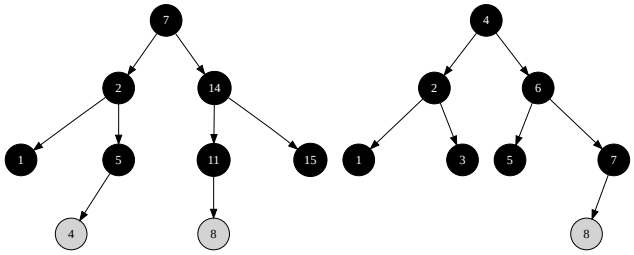


Figure 7: insert results generated from two sequences of keys.

This algorithm shows great simplicity by summarizing the uniform feature from the four different unbalanced cases. It is expressive over the traditional tree rotation approach, that even in programming languages which don't support pattern matching, the algorithm can still be implemented by manually check the pattern. A Scheme/Lisp program is available along with this book can be referenced as an example.

The insertion algorithm takes $O(\lg n)$ time to insert a key to a red-black tree which has n nodes.

Exercise 3

- Write a program in an imperative language, such as C, C++ or python to realize the same algorithm in this section. Note that, because there is no language supported pattern matching, you need to test the 4 different cases manually.

4 Deletion

Remind the deletion section in binary search tree. Deletion is 'imperative only' for red-black tree as well. In typically practice, it often builds the tree just one time, and performs looking up frequently after that. Okasaki explained why he didn't provide red-black tree deletion in his work in [3]. One reason is that deletions are much messier than insertions.

The purpose of this section is just to show that red-black tree deletion is possible in purely functional settings, although it actually rebuilds the tree because trees are read only in terms of purely functional data structure[3]. In real world, it's up to the user (or actually the programmer) to adopt the proper solution. One option is to mark the node be deleted with a flag, and perform a tree rebuilding when the number of deleted nodes exceeds 50%.

Not only in functional settings, even in imperative settings, deletion is more complex than insertion. We face more cases to fix. Deletion may also violate the red black tree properties, so we need fix it after the normal deletion as described in binary search tree.

The deletion algorithm in this book are based on top of a handout in [5]. The problem only happens if you try to delete a black node, because it will violate the last property of red-black tree, which means the number of black node in the path may decreased so that it is not uniformed black-height any more.

When delete a black node, we can resume the last red-black property by introducing a 'doubly-black' concept[1]. It means that the although the node is deleted, the blackness is kept by storing it in the parent node. If the parent node is red, it turns to black, However, if it has been already black, it turns to 'doubly-black'.

In order to express the 'doubly-black node', The definition need some modification accordingly.

```
data Color = R | B | BB -- BB: doubly black for deletion
data RBTree a = Empty | BBEmpty -- doubly black empty
              | Node Color (RBTree a) a (RBTree a)
```

When deleting a node, we first perform the same deleting algorithm in binary search tree mentioned in previous chapter. After that, if the node to be sliced out is black, we need fix the tree to keep the red-black properties. The delete function is defined as the following.

$$delete(T, k) = blackenRoot(del(T, k)) \tag{7}$$

[3]Actually, the common part of the tree is reused. Most functional programming environments support this persistent feature.

where

$$del(T,k) = \begin{cases} \phi & : & T = \phi \\ fixBlack^2((\mathcal{C}, del(T_l,k), k', T_r)) & : & k < k' \\ fixBlack^2((\mathcal{C}, T_l, k', del(T_r,k))) & : & k > k' \\ \begin{cases} mkBlk(T_r) & : & \mathcal{C} = \mathcal{B} \\ T_r & : & otherwise \end{cases} & : & T_l = \phi \\ \begin{cases} mkBlk(T_l) & : & \mathcal{C} = \mathcal{B} \\ T_l & : & otherwise \end{cases} & : & T_r = \phi \\ fixBlack^2((\mathcal{C}, T_l, k'', del(T_r,k''))) & : & otherwise \end{cases} \quad (8)$$

The real deleting happens inside function *del*. For the trivial case, that the tree is empty, the deletion result is ϕ; If the key to be deleted is less than the key of the current node, we recursively perform deletion on its left sub-tree; if it is bigger than the key of the current node, then we recursively delete the key from the right sub-tree; Because it may bring doubly-blackness, so we need fix it.

If the key to be deleted is equal to the key of the current node, we need splice it out. If one of its children is empty, we just replace the node by the other one and reserve the blackness of this node. otherwise we cut and past the minimum element $k'' = min(T_r)$ from the right sub-tree.

Function *delete* just forces the result tree of *del* to have a black root. This is realized by function *blackenRoot*.

$$blackenRoot(T) = \begin{cases} \phi & : & T = \phi \\ (\mathcal{B}, T_l, k, T_r) & : & otherwise \end{cases} \quad (9)$$

The $blackenRoot(T)$ function is almost same as the $makeBlack(T)$ function defined for insertion except for the case of empty tree. This is only valid in deletion, because insertion can't result an empty tree, while deletion may.

Function *mkBlk* is defined to reserved the blackness of a node. If the node to be sliced isn't black, this function won't be applied, otherwise, it turns a red node to black and turns a black node to doubly-black. This function also marks an empty tree ϕ to doubly-black empty Φ.

$$mkBlk(T) = \begin{cases} \Phi & : & T = \phi \\ (\mathcal{B}, T_l, k, T_r) & : & \mathcal{C} = \mathcal{R} \\ (\mathcal{B}^2, T_l, k, T_r) & : & \mathcal{C} = \mathcal{B} \\ T & : & otherwise \end{cases} \quad (10)$$

where $\mathcal{B}^2$ denotes the doubly-black color.

Summarizing the above functions yields the following Haskell program.

```
delete t x = blackenRoot(del t x) where
    del Empty _ = Empty
    del (Node color l k r) x
        | x < k = fixDB color (del l x) k r
        | x > k = fixDB color l k (del r x)
```

```
    — x == k, delete this node
    | isEmpty l = if color==B then makeBlack r else r
    | isEmpty r = if color==B then makeBlack l else l
    | otherwise = fixDB color l k' (del r k') where k'= min r
blackenRoot (Node _ l k r) = Node B l k r
blackenRoot _ = Empty

makeBlack (Node B l k r) = Node BB l k r — doubly black
makeBlack (Node _ l k r) = Node B l k r
makeBlack Empty = BBEmpty
makeBlack t = t
```

The final attack to the red-black tree deletion algorithm is to realize the $fixBlack^2$ function. The purpose of this function is to eliminate the 'doubly-black' colored node by rotation and color changing.

Let's solve the doubly-black empty node first. For any node, If one of its child is doubly-black empty, and the other child is non-empty, we can safely replace the doubly-black empty with a normal empty node.

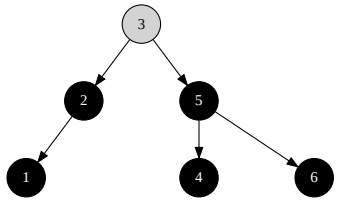
Like figure 8, if we are going to delete the node 4 from the tree (Instead show the whole tree, only part of the tree is shown), the program will use a doubly-black empty node to replace node 4. In the figure, the doubly-black node is shown in black circle with 2 edges. It means that for node 5, it has a doubly-black empty left child and has a right non-empty child (a leaf node with key 6). In such case we can safely change the doubly-black empty to normal empty node. which won't violate any red-black properties.

On the other hand, if a node has a doubly-black empty node and the other child is empty, we have to push the doubly-blackness up one level. For example in figure 9, if we want to delete node 1 from the tree, the program will use a doubly-black empty node to replace 1. Then node 2 has a doubly-black empty node and has an empty right node. In such case we must mark node 2 as doubly-black after change its left child back to empty.

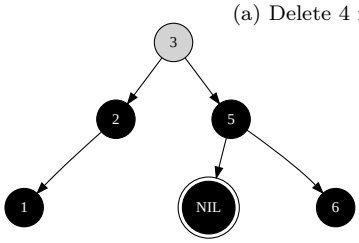
Based on above analysis, in order to fix the doubly-black empty node, we define the function partially like the following.

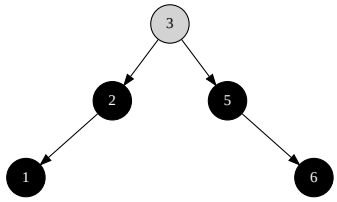
$$fixBlack^2(T) = \begin{cases} (\mathcal{B}^2, \phi, k, \phi) & : & (T_l = \phi \wedge T_r = \Phi) \vee (T_l = \Phi \wedge T_r = \phi) \\ (\mathcal{C}, \phi, k, T_r) & : & T_l = \Phi \wedge T_r \neq \phi \\ (\mathcal{C}, T_l, k, \phi) & : & T_r = \Phi \wedge T_l \neq \phi \\ \ldots & : & \ldots \end{cases} \tag{11}$$

After dealing with doubly-black empty node, we need to fix the case that the sibling of the doubly-black node is black and it has one red child. In this situation, we can fix the doubly-blackness with one rotation. Actually there are 4 different sub-cases, all of them can be transformed to one uniformed pattern. They are shown in the figure 10.



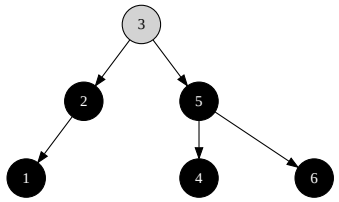
(a) Delete 4 from the tree.



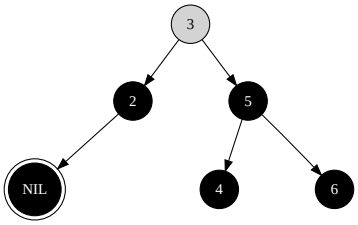


(b) After 4 is sliced off, it is doubly-black empty. (c) We can safely change it to normal NIL.

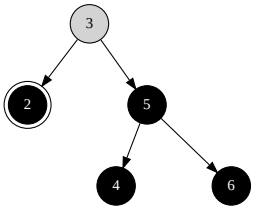
Figure 8: One child is doubly-black empty node, the other child is non-empty.



(a) Delete 1 from the tree.



(b) After 1 is sliced off, it is doubly-black empty.



(c) We must push the doubly-blackness up to node 2.

Figure 9: One child is doubly-black empty node, the other child is empty.

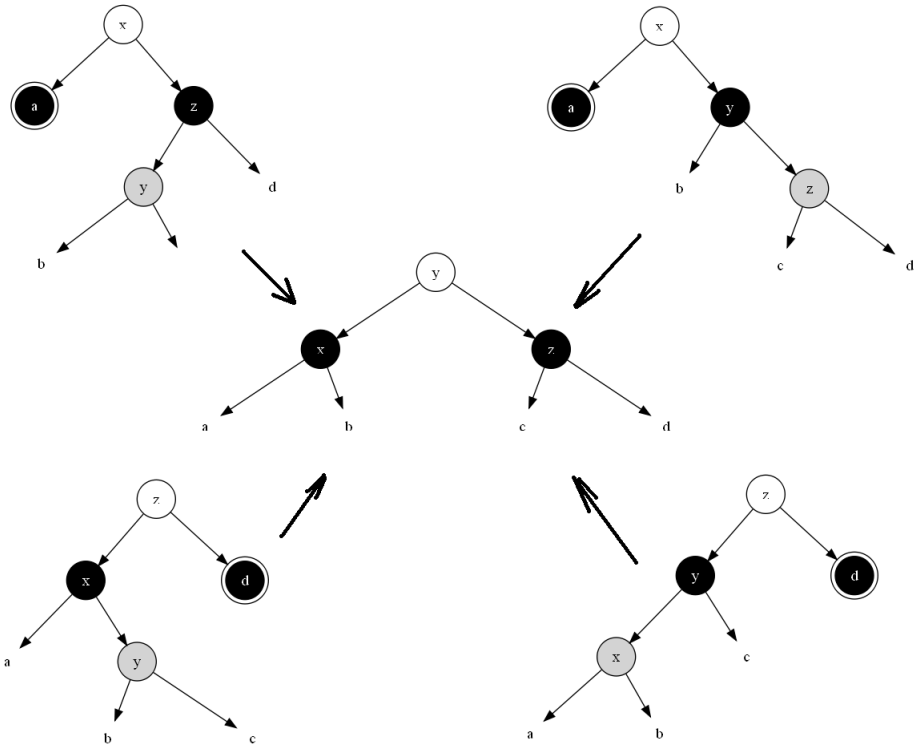


Figure 10: Fix the doubly black by rotation, the sibling of the doubly-black node is black, and it has one red child.

The handling of these 4 sub-cases can be defined on top of formula (11).

$$fixBlack^2(T) = \begin{cases} & \ldots & : & \ldots \\ (\mathcal{C}, (\mathcal{B}, mkBlk(A), x, B), y, (\mathcal{B}, C, z, D)) & : & p1.1 \\ (\mathcal{C}, (\mathcal{B}, A, x, B), y, (\mathcal{B}, C, z, mkBlk(D))) & : & p1.2 \\ & \ldots & : & \ldots \end{cases} \tag{12}$$

where $p1.1$ and $p1.2$ each represent 2 patterns as the following.

$$p1.1 : \left\{ \begin{array}{l} T = (\mathcal{C}, A, x, (\mathcal{B}, (\mathcal{R}, B, y, C), z, D)) \wedge color(A) = \mathcal{B}^2 \\ \vee \\ T = (\mathcal{C}, A, x, (\mathcal{B}, B, y, (\mathcal{R}, C, z, D))) \wedge color(A) = \mathcal{B}^2 \end{array} \right\}$$

$$p1.2 : \left\{ \begin{array}{l} T = (\mathcal{C}, (\mathcal{B}, A, x, (\mathcal{R}, B, y, C)), z, D) \wedge color(D) = \mathcal{B}^2 \\ \vee \\ T = (\mathcal{C}, (\mathcal{B}, (\mathcal{R}, A, x, B), y, C), z, D) \wedge color(D) = \mathcal{B}^2 \end{array} \right\}$$

Besides the above cases, there is another one that not only the sibling of the doubly-black node is black, but also its two children are black. We can change the color of the sibling node to red; resume the doubly-black node to black and propagate the doubly-blackness one level up to the parent node as shown in figure 11. Note that there are two symmetric sub-cases.

We go on adding this fixing after formula (12).

$$fixBlack^2(T) = \begin{cases} & \ldots & : & \ldots \\ mkBlk((\mathcal{C}, mkBlk(A), x, (\mathcal{R}, B, y, C))) & : & p2.1 \\ mkBlk((\mathcal{C}, (\mathcal{R}, A, x, B), y, mkBlk(C))) & : & p2.2 \\ & \ldots & : & \ldots \end{cases} \tag{13}$$

where $p2.1$ and $p2.2$ are two patterns as below.

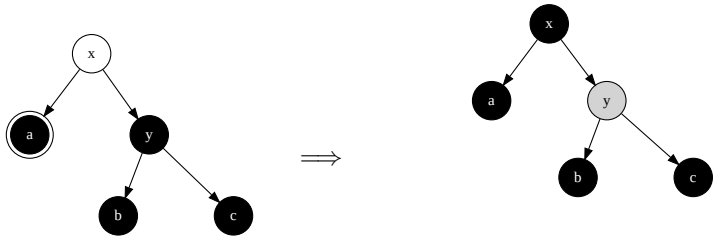
$$p2.1 : \left\{ \begin{array}{l} T = (\mathcal{C}, A, x, (\mathcal{B}, B, y, C)) \wedge \\ color(A) = \mathcal{B}^2 \wedge color(B) = color(C) = \mathcal{B} \end{array} \right\}$$

$$p2.2 : \left\{ \begin{array}{l} T = (\mathcal{C}, (\mathcal{B}, A, x, B), y, C) \wedge \\ color(C) = \mathcal{B}^2 \wedge color(A) = color(B) = \mathcal{B} \end{array} \right\}$$

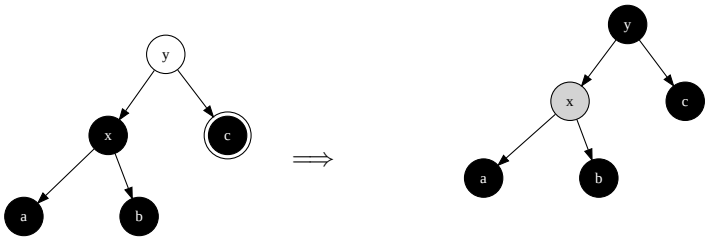
There is a final case left, that the sibling of the doubly-black node is red. We can do a rotation to change this case to pattern $p1.1$ or $p1.2$. Figure 12 shows about it.

We can finish formula (13) with (14).

$$fixBlack^2(T) = \begin{cases} & \ldots & : & \ldots \\ fixBlack^2(\mathcal{B}, fixBlack^2((\mathcal{R}, A, x, B), y, C) & : & p3.1 \\ fixBlack^2(\mathcal{B}, A, x, fixBlack^2((\mathcal{R}, B, y, C)) & : & p3.2 \\ & T & : & otherwise \end{cases} \tag{14}$$



(a) Color of x can be either black or red. (b) If x was red, then it becomes black, otherwise, it becomes doubly-black.



(c) Color of y can be either black or red. (d) If y was red, then it becomes black, otherwise, it becomes doubly-black.

Figure 11: propagate the blackness up.

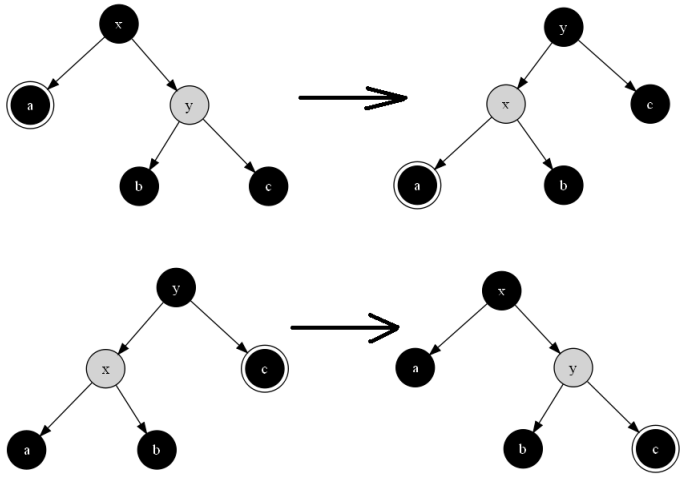


Figure 12: The sibling of the doubly-black node is red.

where $p3.1$ and $p3.2$ are two patterns as the following.

$$p3.1 : \{color(T) = \mathcal{B} \wedge color(T_l) = \mathcal{B}^2 \wedge color(T_r) = \mathcal{R}\}$$

$$p3.2 : \{color(T) = \mathcal{B} \wedge color(T_l) = \mathcal{R} \wedge color(T_r) = \mathcal{B}^2\}$$

Fixing the doubly-black node with all above different cases is a recursive function. There are two termination conditions. One contains pattern $p1.1$ and $p1.2$, The doubly-black node was eliminated. The other cases may continuously propagate the doubly-blackness from bottom to top till the root. Finally the algorithm marks the root node as black anyway. The doubly-blackness will be removed.

Put formula (11), (12), (13), and (14) together, we can write the final Haskell program.

```
fixDB color BBEmpty k Empty = Node BB Empty k Empty
fixDB color BBEmpty k r = Node color Empty k r
fixDB color Empty k BBEmpty = Node BB Empty k Empty
fixDB color l k BBEmpty = Node color l k Empty
-- the sibling is black, and it has one red child
fixDB color a@(Node BB _ _ _) x (Node B (Node R b y c) z d) =
    Node color (Node B (makeBlack a) x b) y (Node B c z d)
fixDB color a@(Node BB _ _ _) x (Node B b y (Node R c z d)) =
    Node color (Node B (makeBlack a) x b) y (Node B c z d)
fixDB color (Node B a x (Node R b y c)) z d@(Node BB _ _ _) =
    Node color (Node B a x b) y (Node B c z (makeBlack d))
fixDB color (Node B (Node R a x b) y c) z d@(Node BB _ _ _) =
    Node color (Node B a x b) y (Node B c z (makeBlack d))
-- the sibling and its 2 children are all black, propagate the blackness up
fixDB color a@(Node BB _ _ _) x (Node B b@(Node B _ _ _) y c@(Node B _ _ _))
  = makeBlack (Node color (makeBlack a) x (Node R b y c))
fixDB color (Node B a@(Node B _ _ _) x b@(Node B _ _ _)) y c@(Node BB _ _ _)
  = makeBlack (Node color (Node R a x b) y (makeBlack c))
-- the sibling is red
fixDB B a@(Node BB _ _ _) x (Node R b y c) = fixDB B (fixDB R a x b) y c
fixDB B (Node R a x b) y c@(Node BB _ _ _) = fixDB B a x (fixDB R b y c)
-- otherwise
fixDB color l k r = Node color l k r
```

The deletion algorithm takes $O(\lg n)$ time to delete a key from a red-black tree with n nodes.

Exercise 4

- As we mentioned in this section, deletion can be implemented by just marking the node as deleted without actually removing it. Once the number of marked nodes exceeds 50%, a tree re-build is performed. Try to implement this method in your favorite programming language.

- Why needn't enclose $mkBlk$ with a call to $fixBlack^2$ explicitly in the definition of $del(T, k)$?

5 Imperative red-black tree algorithm ⋆

We almost finished all the content in this chapter. By induction the patterns, we can implement the red-black tree in a simple way compare to the imperative tree rotation solution. However, we should show the comparator for completeness.

For insertion, the basic idea is to use the similar algorithm as described in binary search tree. And then fix the balance problem by rotation and return the final result.

```
1:  function INSERT(T, k)
2:      root ← T
3:      x ← CREATE-LEAF(k)
4:      COLOR(x) ← RED
5:      p ← NIL
6:      while T ≠ NIL do
7:          p ← T
8:          if k < KEY(T) then
9:              T ← LEFT(T)
10:         else
11:             T ← RIGHT(T)
12:     PARENT(x) ← p
13:     if p = NIL then                          ▷ tree T is empty
14:         return x
15:     else if k < KEY(p) then
16:         LEFT(p) ← x
17:     else
18:         RIGHT(p) ← x
19:     return INSERT-FIX(root, x)
```

The only difference from the binary search tree insertion algorithm is that we set the color of the new node as red, and perform fixing before return. It is easy to translate the pseudo code to real imperative programming language, for instance Python [4].

```
def rb_insert(t, key):
    root = t
    x = Node(key)
    parent = None
    while(t):
        parent = t
        if(key < t.key):
            t = t.left
```

[4] C, and C++ source codes are available along with this book

```
        else:
            t = t.right
    if parent is None: #tree is empty
        root = x
    elif key < parent.key:
        parent.set_left(x)
    else:
        parent.set_right(x)
    return rb_insert_fix(root, x)
```

There are 3 base cases for fixing, and if we take the left-right symmetric into consideration. there are total 6 cases. Among them two cases can be merged together, because they all have uncle node in red color, we can toggle the parent color and uncle color to black and set grand parent color to red. With this merging, the fixing algorithm can be realized as the following.

```
1:  function INSERT-FIX(T, x)
2:      while PARENT(x) ≠ NIL ∧ COLOR(PARENT(x)) = RED do
3:          if COLOR(UNCLE(x)) = RED then          ▷ Case 1, x's uncle is red
4:              COLOR(PARENT(x)) ← BLACK
5:              COLOR(GRAND-PARENT(x)) ← RED
6:              COLOR(UNCLE(x)) ← BLACK
7:              x ← GRAND-PARENT(x)
8:          else                                          ▷ x's uncle is black
9:              if PARENT(x) = LEFT(GRAND-PARENT(x)) then
10:                 if x = RIGHT(PARENT(x)) then  ▷ Case 2, x is a right child
11:                     x ← PARENT(x)
12:                     T ← LEFT-ROTATE(T, x)
                                                  ▷ Case 3, x is a left child
13:                 COLOR(PARENT(x)) ← BLACK
14:                 COLOR(GRAND-PARENT(x)) ← RED
15:                 T ← RIGHT-ROTATE(T, GRAND-PARENT(x))
16:             else
17:                 if x = LEFT(PARENT(x)) then       ▷ Case 2, Symmetric
18:                     x ← PARENT(x)
19:                     T ← RIGHT-ROTATE(T, x)
                                                  ▷ Case 3, Symmetric
20:                 COLOR(PARENT(x)) ← BLACK
21:                 COLOR(GRAND-PARENT(x)) ← RED
22:                 T ← LEFT-ROTATE(T, GRAND-PARENT(x))
23:     COLOR(T) ← BLACK
24:     return T
```

This program takes $O(\lg n)$ time to insert a new key to the red-black tree. Compare this pseudo code and the *balance* function we defined in previous section, we can see the difference. They differ not only in terms of simplicity, but also in logic. Even if we feed the same series of keys to the two algorithms, they may build different red-black trees. There is a bit performance overhead in the pattern matching algorithm. Okasaki discussed about the difference in

detail in his paper[2].

Translate the above algorithm to Python yields the below program.

```
# Fix the red->red violation
def rb_insert_fix(t, x):
    while(x.parent and x.parent.color==RED):
        if x.uncle().color == RED:
            #case 1: ((a:R x:R b) y:B c:R) ==> ((a:R x:B b) y:R c:B)
            set_color([x.parent, x.grandparent(), x.uncle()],
                      [BLACK, RED, BLACK])
            x = x.grandparent()
        else:
            if x.parent == x.grandparent().left:
                if x == x.parent.right:
                    #case 2: ((a x:R b:R) y:B c) ==> case 3
                    x = x.parent
                    t=left_rotate(t, x)
                # case 3: ((a:R x:R b) y:B c) ==> (a:R x:B (b y:R c))
                set_color([x.parent, x.grandparent()], [BLACK, RED])
                t=right_rotate(t, x.grandparent())
            else:
                if x == x.parent.left:
                    #case 2': (a x:B (b:R y:R c)) ==> case 3'
                    x = x.parent
                    t = right_rotate(t, x)
                # case 3': (a x:B (b y:R c:R)) ==> ((a x:R b) y:B c:R)
                set_color([x.parent, x.grandparent()], [BLACK, RED])
                t=left_rotate(t, x.grandparent())
    t.color = BLACK
    return t
```

Figure 13 shows the results of feeding same series of keys to the above python insertion program. Compare them with figure 7, one can tell the difference clearly.

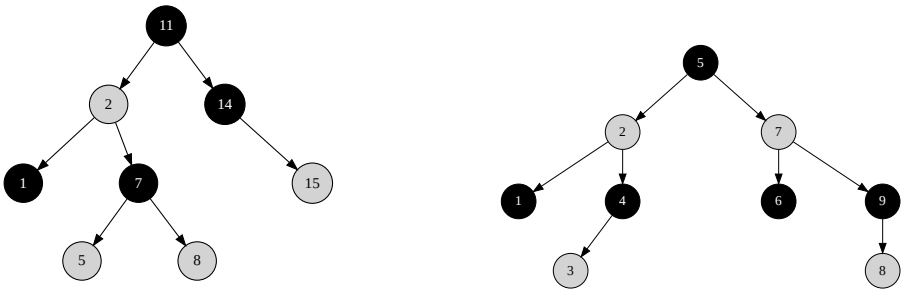
We skip the red-black tree delete algorithm in imperative settings, because it is even more complex than the insertion. The implementation of deleting is left as an exercise of this chapter.

Exercise 5

- Implement the red-black tree deleting algorithm in your favorite imperative programming language. you can refer to [1] for algorithm details.

6 More words

Red-black tree is the most popular implementation of balanced binary search tree. Another one is the AVL tree, which we'll introduce in next chapter. Red-black tree can be a good start point for more data structures. If we extend the



(a) (b)

Figure 13: Red-black trees created by imperative algorithm.

number of children from 2 to k, and keep the balance as well, it leads to B-tree, If we store the data along with edge but not inside node, it leads to Tries. However, the multiple cases handling and the long program tends to make new comers think red-black tree is complex.

Okasaki's work helps making the red-black tree much easily understand. There are many implementation in other programming languages in that manner [7]. It's also inspired me to find the pattern matching solution for Splay tree and AVL tree etc.

References

[1] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest and Clifford Stein. "Introduction to Algorithms, Second Edition". ISBN:0262032937. The MIT Press. 2001

[2] Chris Okasaki. "FUNCTIONAL PEARLS Red-Black Trees in a Functional Setting". J. Functional Programming. 1998

[3] Chris Okasaki. "Ten Years of Purely Functional Data Structures". http://okasaki.blogspot.com/2008/02/ten-years-of-purely-functional-data.html

[4] Wikipedia. "Red-black tree". http://en.wikipedia.org/wiki/Red-black_tree

[5] Lyn Turbak. "Red-Black Trees". cs.wellesley.edu/ cs231/fall01/red-black.pdf Nov. 2, 2001.

[6] SGI STL. http://www.sgi.com/tech/stl/

[7] Pattern matching. http://rosettacode.org/wiki/Pattern_matching