University of Zurich[UZH]

Department of Informatics
Database Technology Group
Prof. Dr. M. Böhlen

Data Structures and Algorithms
Assignment 5

Apr 18, 2016

Red-Black Trees [60 points]

Task 1. [**10 points**] The values **20 50 15 ; 10 60 90 ; 40 30 ;** are inserted in the given order into an empty `red-black tree`. Draw the red-black tree at the positions marked by a semicolon.

Task 2. [**20 points**] This task is a preparatory step to implement a **red-black tree**. A **red-black node** is of the following type:

```
struct rb_node {
  int key, color;
  struct rb_node *left, *right, *parent;
};
```

A **red-black tree** is of the following type:

```
struct rb_tree {
  struct rb_node *root;
  struct rb_node *nil;
};
```

In datatype `rb_tree`, `root` points to the root of the tree. Sentinel `nil` is a convenient node that deals with boundary conditions in red-black tree code. For a red-black tree `T`, the sentinel `T.nil` is an object with the same attributes as an ordinary node in the tree. Its color attribute is `black`, its `parent, left, right` are `T.nil`, and its `key` can take on any arbitrary values. We use the sentinel so that we can treat a `NIL` child of a node `x` as an ordinary node whose parent is `x`. We use one sentinel `T.nil` to represent all NIL nodes of a red-black tree `T` (all leaves and the root's parent). Refer to Fig. 1 for illustration.

Along with the above datatypes create two constants, *red* and *black* equal to 0 and 1 respectively, and the following functions:

- *struct rb_tree* rb_initialize()* that creates a red black tree `T` with a *root* and a *NIL node* (`left = right = parent = T.nil` and `color = black`).

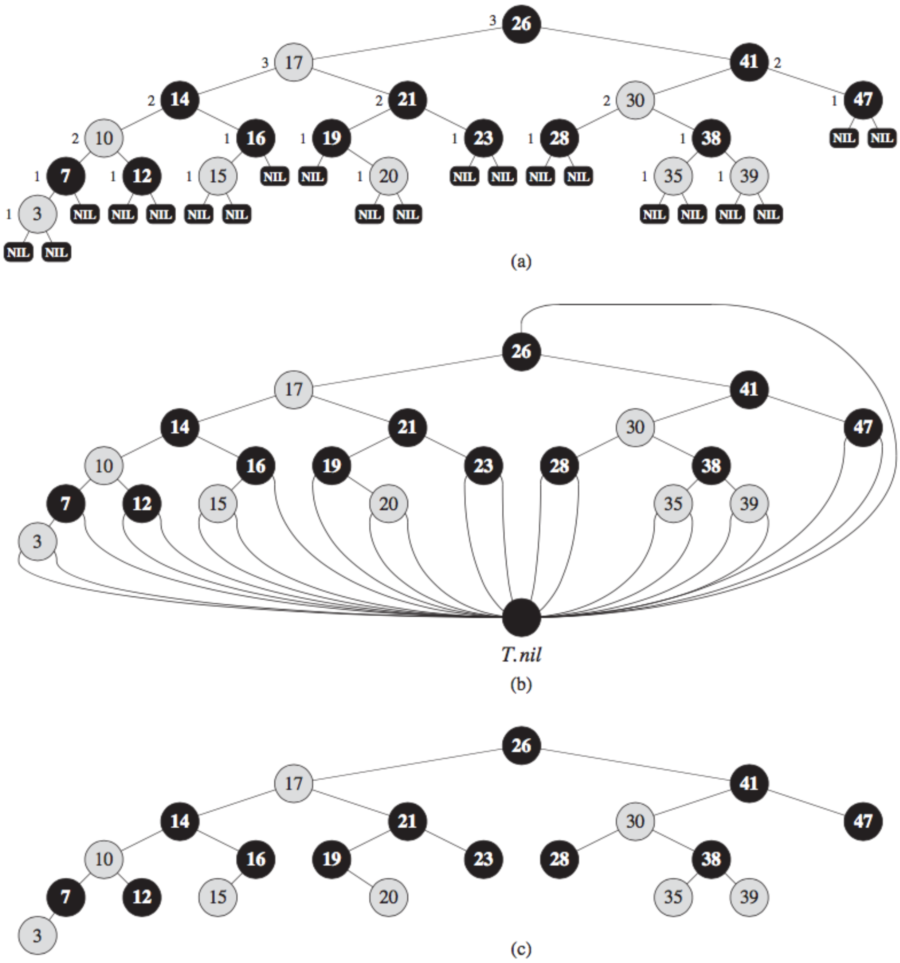


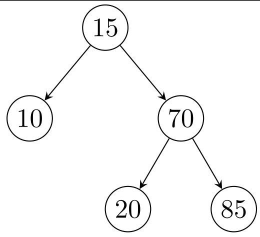
Figure 1: A red-black tree with black nodes darkened and red nodes shaded. *(a)* Every leaf, shown as a NIL, is black. *(b)* The same red-black tree but with each `NIL` replaced by the single sentinel `T.nil`, that is always black. The root's parent is also the sentinel. *(c)* The same red-black tree but with leaves and the root's parent omitted entirely.

- *void bst_insert(struct rb_tree* tree, struct rb_node *nodeToInsert)* that inserts the node `nodeToInsert` into `tree` using the binary search tree insertion algorithm.
- *void rb_print(struct rb_tree *tree)* which prints a given `tree` in a horizontal way. A node `v` with depth `a` and order `b` in in-order traversal of `tree` should be printed at row `b` and column `a`. Root node has depth 0.

Output Form

```
   10
15
      20
   70
      85
```

Alternative Output Form



- *struct rb_node* rb_search(struct rb_tree* tree, int q)* that returns the node with key equals to `q` if it exists in `tree`, otherwise it returns `T.nil`.
- *void rb_leftRotate(struct rb_tree* tree, struct rb_node* x)* that does left rotation on node `x` in `tree`.
- *void rb_rightRotate(struct rb_tree* tree, struct rb_node* x)* that does right rotation on node `x` in `tree`.

Test your implementation by performing the following operations:

- Initialize a red-back tree `T`;
- Insert 5, 90, 20 into `T`.
- Print the tree.
- Right rotate node 90.
- Left rotate node 5.
- Print the tree.
- Insert 60, 30 into `T`.
- Print the tree.
- Right rotate node 90.
- Print the tree.

Note: For this task the color attribute of nodes is not relevant and can be ignored.

Task 3 (Optional). [20 points] Using the implementation from Task 2, implement the following functions for a `red-black tree`:

- *struct rb_node* rb_insert_fixup(struct rb_tree* tree, struct rb_node* n)* that *fixes* node `n` in `tree` after insertion to restore the red-black properties. Make sure your function covers all the cases mentioned in the lecture and their mirror cases.
- *struct rb_node* rb_insert(struct rb_tree* tree, struct rb_node* n)* that uses *bst_insert* to insert a new node with key value k into *tree* and then uses *rb_insert_fixup* to restore the red-black properties.

Test your implementation by performing the following operations:

- Initialize a red-back tree `T`;
- Insert 5, 90, 20 into `T`.
- Print the tree.
- Insert 60, 30 into `T`.
- Print the tree.
- Insert 50, 40 into `T`.
- Print the tree.

Task 4 (Optional). [10 points] This task is about comparing the insertion efficiency in binary search trees and red black trees. Insert values from 1 to 500'000, in increasing order, in two different trees using your `bst_insert` and `rb_insert` functions, from Task 2 and Task 3, and report the elapsed time as follows:

	time (sec.)
BST	
red black tree	

Submission

For this exercise, you need to submit a zipped folder *a<exercise number>_<family name>_<matriculation number>.zip* where `family name` and `matriculation number` correspond to your personal data. This folder should include:

a) the C-files you created for the tasks where an implementation was needed. Each C-file should be named as *task<task number>.c*

b) a pdf named *a<exercise number>.pdf* with the solutions for the rest of the tasks.

Deadline: **Sunday, May 1st at 23:59**.