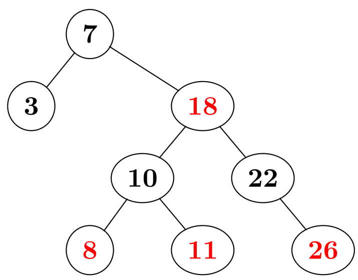
Informatics II
Exercise 9 / **Solution**

April 29, 2020

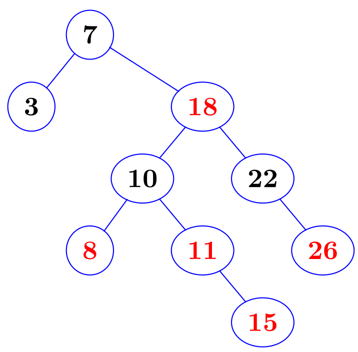
Red Black Trees

Task 1. Consider the red-black tree shown in following figure. Show the step-wise procedure and changes in tree when 15 is inserted in the tree (You have to show the state of tree after color change and rotation transformations).

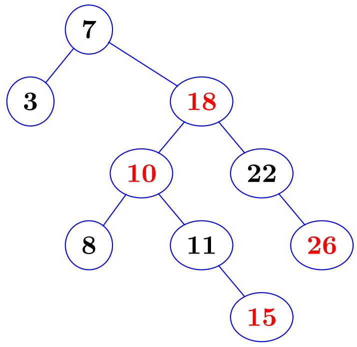
Initial tree



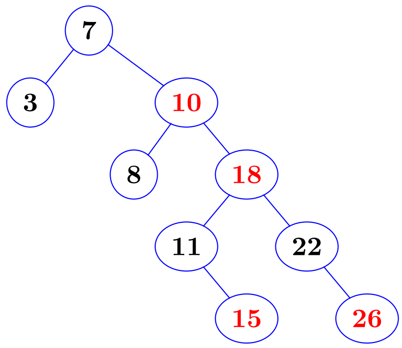
Insert 15



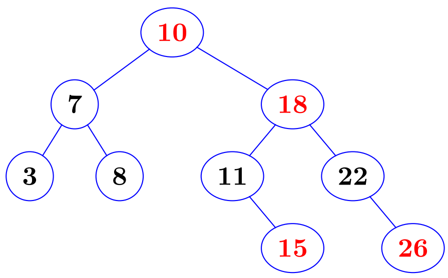
Recolor



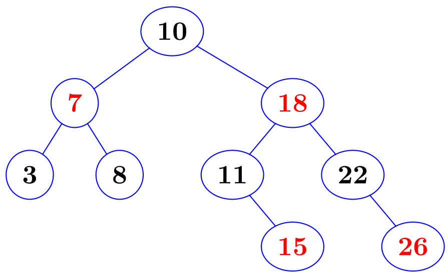
Right-Rotate(18)



Left-Rotate(7)

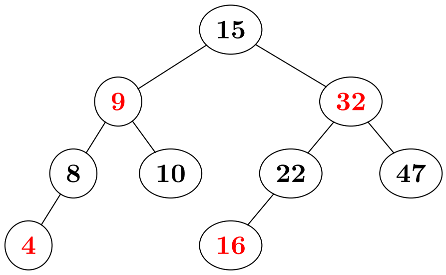


Recolor

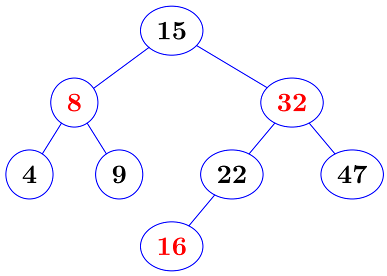


Task 2. Consider the red-black tree in Figure 1. The nodes with key 15, 8, 10, 22 and 47 are black, whereas the nodes with keys 9, 32, 4 and 16 are red. Perform the following operations on it and draw the resulting tree after each of them: Delete 10, Insert 5, Delete 9, Delete 15.

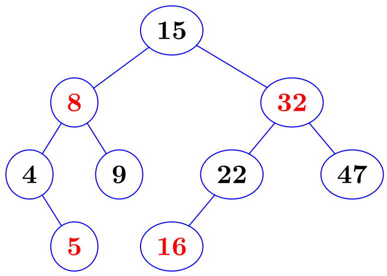
Initial tree



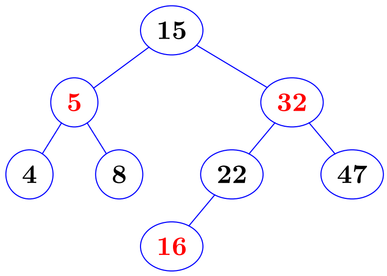
Delete 10



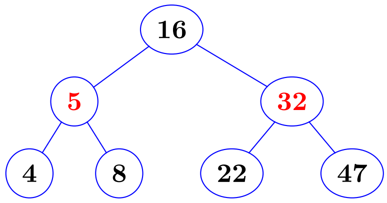
Insert 5



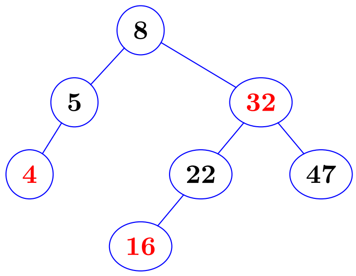
Delete 9



Delete 15: 16 becomes root



Delete 15: 8 becomes root



Task 3. You are given a nearly complete implementation of red-black trees in the file `task1_framework.c`. 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 {
  int bh;
  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:

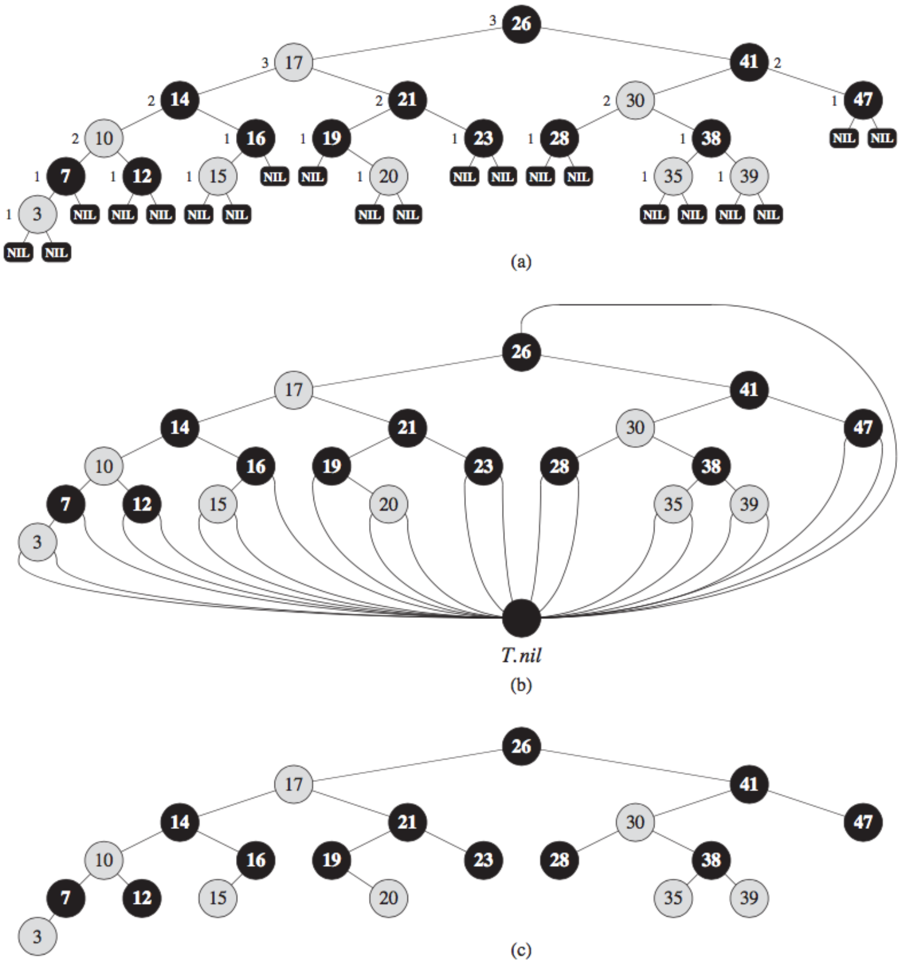


Figure 1: A red-black tree with black nodes darkened and red nodes shaded. *(a)* Every leaf, shown as a NIL, is black. *(b)* The same red-black tree but with each `NIL` replaced by the single sentinel `T.nil`, that is always black. The root's parent is also the sentinel. *(c)* The same red-black tree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

- *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`).

```
struct rb_tree* rb_initialize() {
  struct rb_tree* tree;
  struct rb_node* node;

  tree = (struct rb_tree*) malloc(sizeof(struct rb_tree));

  tree->nil = (struct rb_node*) malloc(sizeof(struct rb_node));
  tree->nil->parent = tree->nil;
  tree->nil->left = tree->nil;
  tree->nil->right = tree->nil;
  tree->nil->color = black;
  tree->nil->key = -2;

  tree->root = tree->nil;
  tree->bh = 0;

  return tree;
}
```

- *void rb_leftRotate(struct rb_tree* tree, struct rb_node* x)* that does left rotation on node `x` in `tree`.

```
void rb_leftRotate(struct rb_tree* T, struct rb_node* x) {
  struct rb_node* y;
  if(x->right == T->nil) return;
  y = x->right;
  x->right = y->left;
  y->parent = x->parent;
  if (y->left != T->nil) {
    y->left->parent = x;
  }
  if(x->parent == T->nil){
    T->root = y;
  }else{
   if (x == x->parent->left) {
    x->parent->left = y;
  }
  else {
    x->parent->right = y;
  }
  }
  y->left = x;
  x->parent = y;
}
```

- *void rb_rightRotate(struct rb_tree* tree, struct rb_node* x)* that does right rotation on node `x` in `tree`.

```
void rb_rightRotate(struct rb_tree* T, struct rb_node* x) {
  struct rb_node* y;
  if(x->left == T->nil) return;
  y = x->left;
```

```
  x->left = y->right;
  y->parent = x->parent;

  if(y->right!= T->nil){
    y->right->parent =x;
  }
  if (x->parent == T->nil) {
    T->root = y;
  }else{
    if (x == x->parent->right) {
        x->parent->right = y;
    }
    else {
        x->parent->left = y;
    }
  }
  y->right = x;
  x->parent = y;
}
```

- *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_insertFixup(struct rb_tree* T, struct rb_node* z) {
  struct rb_node* y;
  while (z->parent->color == red) {
    if (z->parent == z->parent->parent->left) { /* non-mirrored cases */
      y = z->parent->parent->right;
      if (y->color == red) { /* case 1 */
        z->parent->color = black;
        y->color = black;
        z->parent->parent->color = red;
        z = z->parent->parent;
      } else {
        if (z == z->parent->right) { /* case 2 */
          z = z->parent;
          rb_leftRotate(T, z);
        }
        z->parent->color = black; /* case 3 */
        z->parent->parent->color = red;
        rb_rightRotate(T, z->parent->parent);
      }
    } else { /* mirrored cases */
      y = z->parent->parent->left;
      if (y->color == red) { /* case 1m */
        z->parent->color = black;
        y->color = black;
        z->parent->parent->color = red;
        z = z->parent->parent;
      } else {
        if (z == z->parent->left) { /* case 2m */
          z = z->parent;
```

```
            rb_rightRotate(T, z);
          }
          z->parent->color = black; /* case 3m */
          z->parent->parent->color = red;
          rb_leftRotate(T, z->parent->parent);
        }
      }
    }
    if (T->root->color == red) {
      T->bh += 1;
    }
    T->root->color = black;
}
```

- *void rb_insert(struct rb_tree* T, int key)* that inserts a new node with key value `k` into `tree` and then uses *rb_insert_fixup* to restore the red-black properties.

```
void rb_insert(struct rb_tree* T, int key) {
    struct rb_node *oneDelayed = T->nil;
    struct rb_node *insertPlace = T->root;
    struct rb_node *nodeToInsert =
      (struct rb_node*) malloc(sizeof(struct rb_node));
    nodeToInsert->key = key;
    nodeToInsert->color = red;

    nodeToInsert->left= T->nil;
    nodeToInsert->right= T->nil;
    nodeToInsert->parent= T->nil;
    while (insertPlace != T->nil) {
      oneDelayed = insertPlace;
      if (nodeToInsert->key < insertPlace->key) {
        insertPlace = insertPlace->left;
      }
      else {
        insertPlace = insertPlace->right;
      }
    }

    if (oneDelayed == T->nil) {
      T->root = nodeToInsert;
    }
    else if (oneDelayed->key < nodeToInsert->key) {
      oneDelayed->right = nodeToInsert;
      nodeToInsert->parent = oneDelayed;
    }
    else {
      oneDelayed->left = nodeToInsert;
      nodeToInsert->parent = oneDelayed;
    }
    rb_insertFixup(T, nodeToInsert);
}
```

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 20.
- Left rotate node 5.
- Print the tree.
- Insert 60, 30 into `T`.
- Print the tree.
- Right rotate node 90.
- Print the tree.

```
struct rb_tree *T;
T = rb_initialize();

rb_insert(T, 5);
rb_insert(T, 90);
rb_insert(T, 20);
rb_print(T);
rb_rightRotate(T, rb_search(T, 20));
rb_print(T);
rb_leftRotate(T, rb_search(T, 5));
rb_print(T);
rb_insert(T, 60);
rb_insert(T, 30);
rb_print(T);
rb_rightRotate(T, rb_search(T, 90));
rb_print(T);
```