Department of Informatics
Database Technology Group
University of Zurich
Prof. Dr. Anton Dignös

Informatics II
Exercise 9

April 26, 2020

Learning Goals:

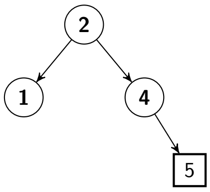
- Practise inserting nodes in a Red Black Tree
- Practise deleting nodes from a Red Black Tree.
- Implement Red Black Trees on C.

Red Black Trees

Consider the red-black tree in Figure 1a where black nodes are denoted with a circle and red nodes are denoted with a square. We use the key to represent a node. For example, 2 represents the node with key 2. We consider four operations:

1. Creating a new node (left and right are set to NULL):
   Example: create node 9: `create(9)`
2. Setting a property (color, key, left, right) of a node:
   Example: set the key of node 9 to 5: `9->key = 5.` Here, 9 represents the node with key 9.
3. Rotating a node:
   Example: right rotate node 5: `RightRotate(5)`
4. Deleting a node:
   Example: delete node 9: `delete(9)`

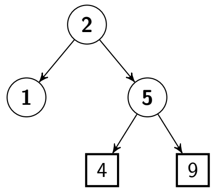
The result of applying the operations in Figure 1b to the red-black tree in Figure 1a yields the red-black tree in Figure 1c.



(a) Before operations

```
create(9)
9->color = red
5->right = 9
5->color = black
4->color = red
LeftRotate(4)
```

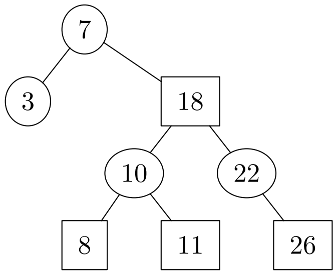
(b) Operations



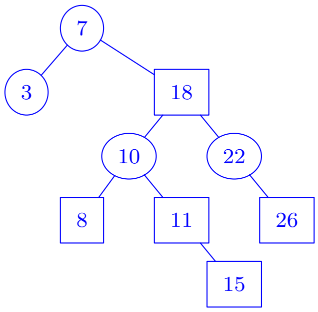
(c) After operations

Figure 1: Tree before and after operations

Task 1. Consider the red-black tree shown below. State the operations that are required to insert 15 into the red-black tree.

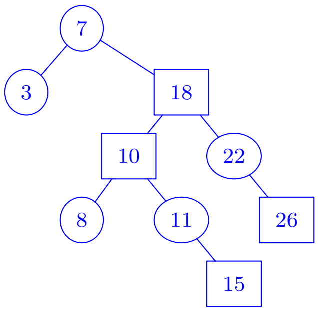


Operations:
```
create 15
11->right = 15
```



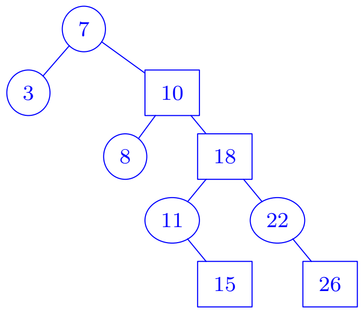
(Note that what follow in the **parentheses** are not operations. Analysis: set t = 15, p = 11, u = 18, g = 10; Case 1.)

Operations:
```
8->color = black
11-color = black
10-> color = red
```



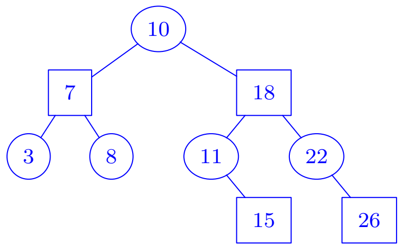
(Analysis: set t = 10, p = 18, u = 3, g = 7.
Inverted case 2, t is a left child.).

Operations:
`RightRotate(18)`

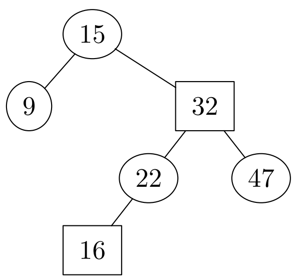


(Analysis: set t = 18, p = 10, u = 3, g = 7.
Inverted case 3, t is a right child.)

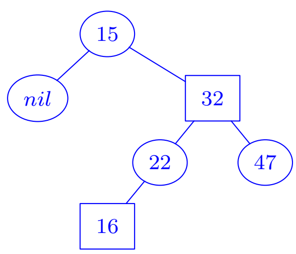
Operations:
```
LeftRotate(7)
10->color = black
7->color = red
```



Task 2. Consider the red-black tree below.

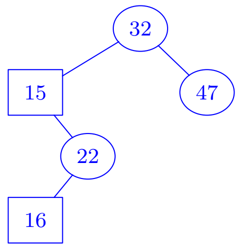


State the operations that are required to delete **9** from the red-black tree.
Operations:
`delete 9`



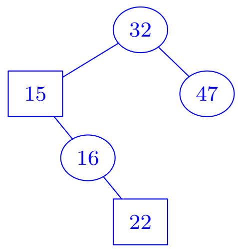
(Note that what follow in the **parentheses** are not operations.
Analysis: set x = nil, p = 15, b = 32, m = 22, n = 47. Case 1.)

Operations:
Leftrotate(15)
`32->color = black`
`15->color = red`



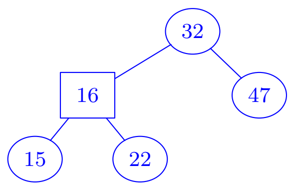
(Analysis: x = nil, set b = 22, m = 16, n = nil. Case 3.)

Operations:
Rightrotate(22)
`16->color = black`
`22->color = red`



(Analysis: x = nil, p = 15, set b = 16, m = nil, n = 22. Case 3.)

Operations:
LeftRotate(15)
`16->color = red(15->color)`
`15->color = black`
`22->color = black`



Task 3. You are asked to complete an implementation of red black trees. 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. 2 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`).

```
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;
```

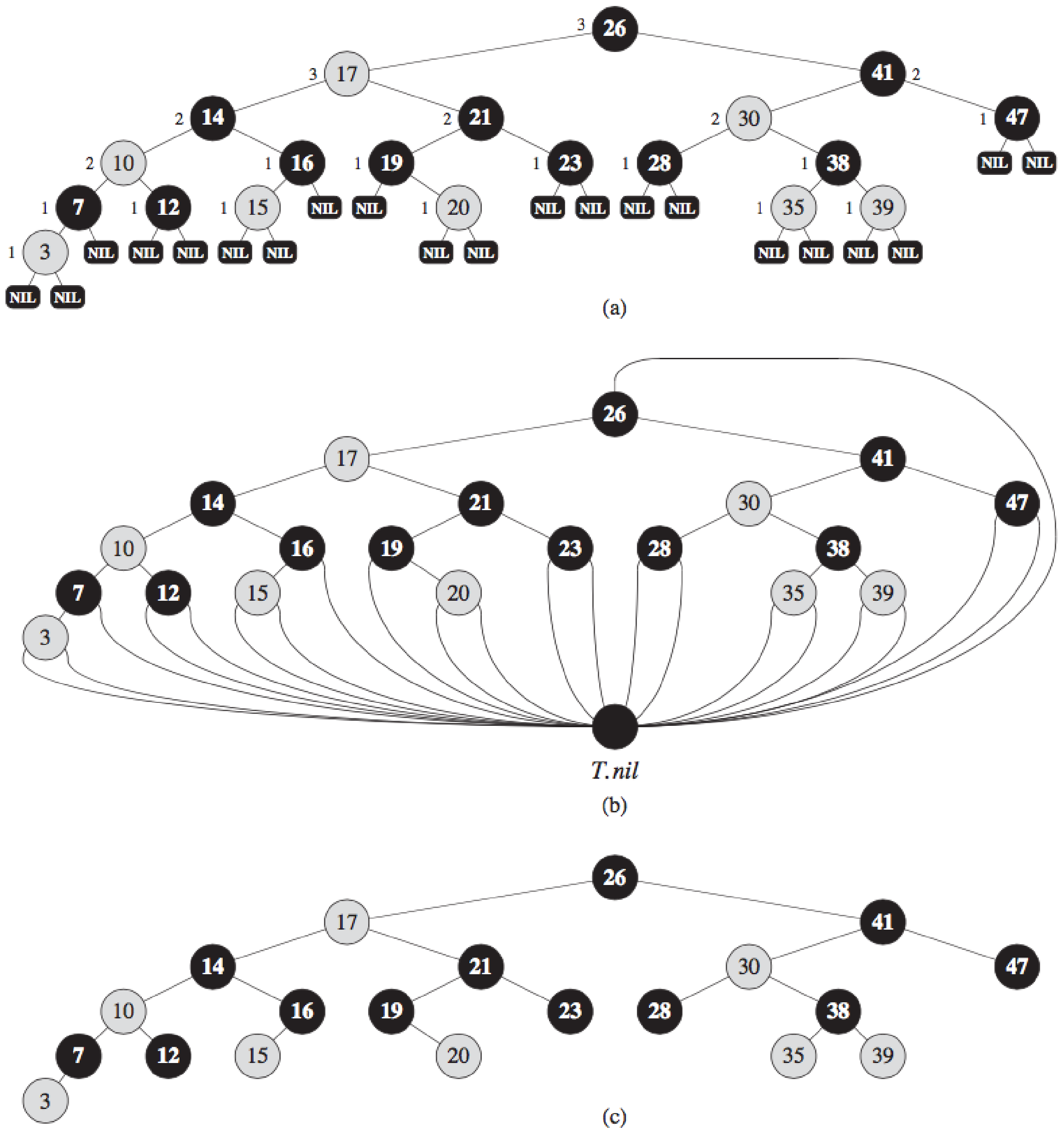


Figure 2: A red-black tree with black nodes darkened and red nodes shaded. *(a)* Every leaf, shown as a NIL, is black. *(b)* The same red-black tree but with each `NIL` replaced by the single sentinel `T.nil`, that is always black. The root's parent is also the sentinel. *(c)* The same red-black tree but with leaves and the root's parent omitted entirely.

```
  }
  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);
```