Algorithms and Data Structures: Homework #9

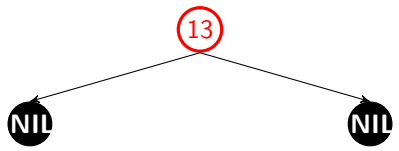
Due on April 14, 2020 at 23:00

Henri Sota

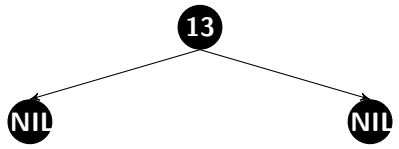
Problem 9.1

a) Draw (or describe by using preorder traversal) the red-black trees that result after successively inserting the values step by step in the following order $[13, 44, 37, 7, 22, 16]$ into an empty red-black tree. You are required to draw (or describe by using preorder traversal) the tree after each insertion, as well as any additional recoloring and balancing.

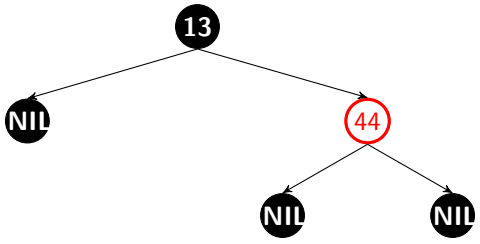
1) Inserting 13 into the tree:



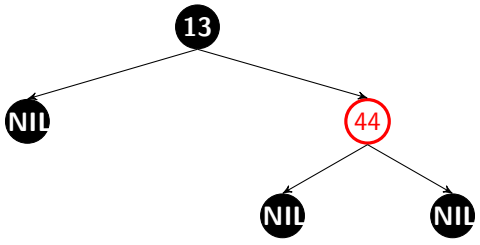
Fix up after insertion:



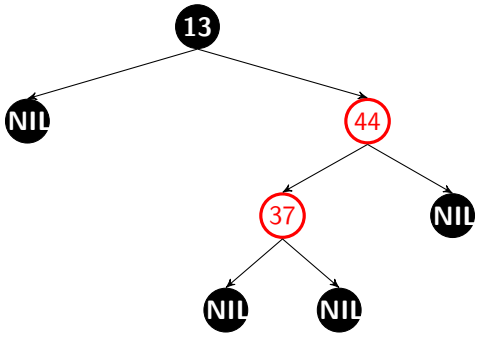
2) Inserting 44 into the tree:



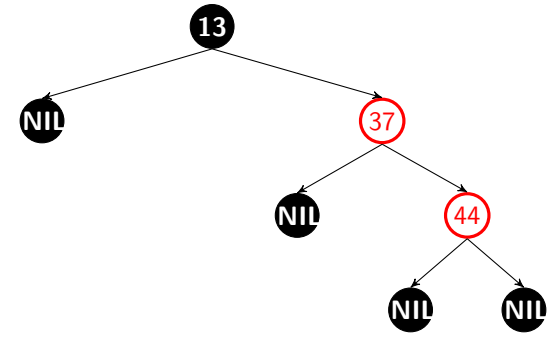
Fix up after insertion:

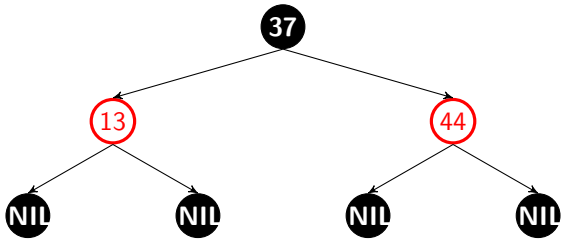


3) Inserting 37 into the tree:

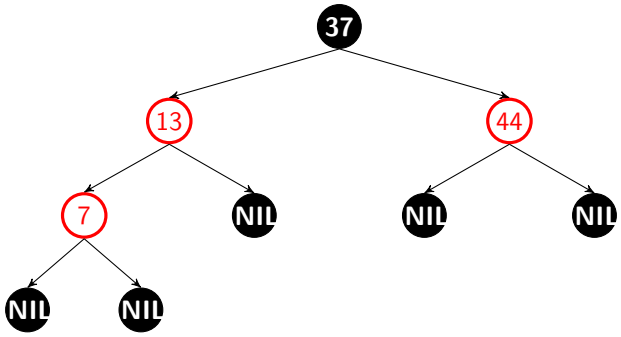


Fix up after insertion:

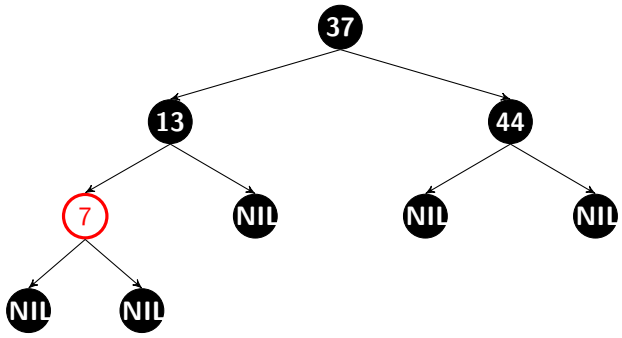




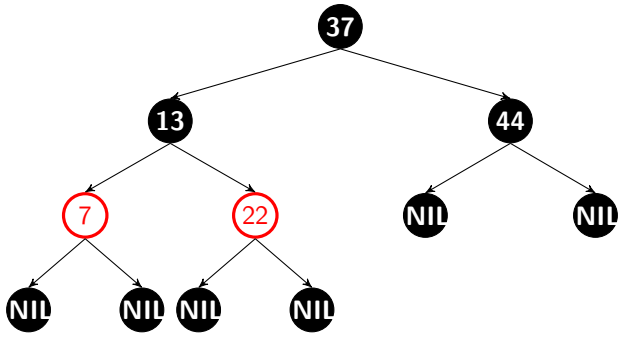
4) Inserting 7 into the tree:



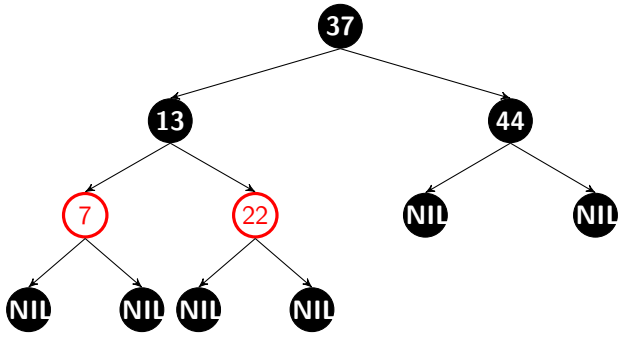
Fix up after insertion:



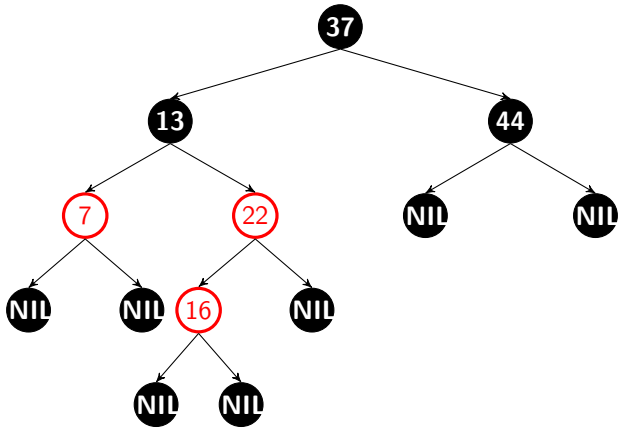
5) Inserting 22 into the tree:



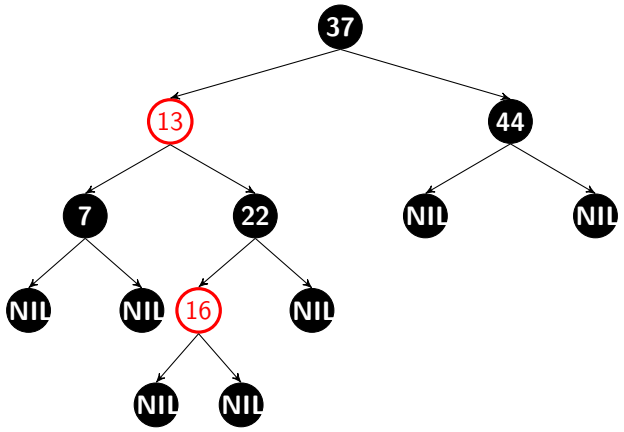
Fix up after insertion:



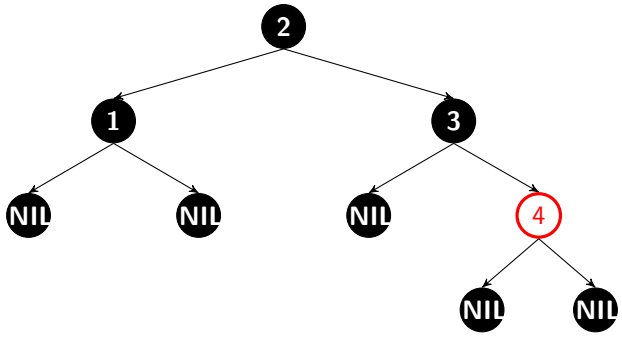
6) Inserting 16 into the tree:

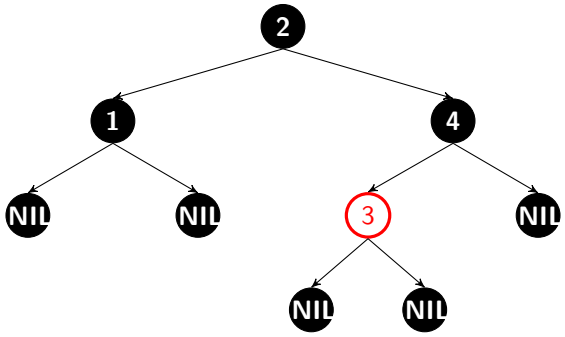


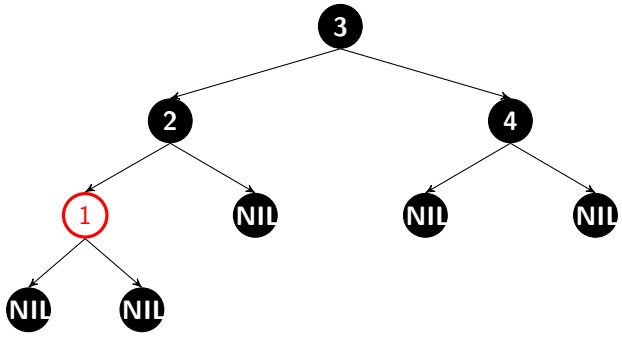
Fix up after insertion:

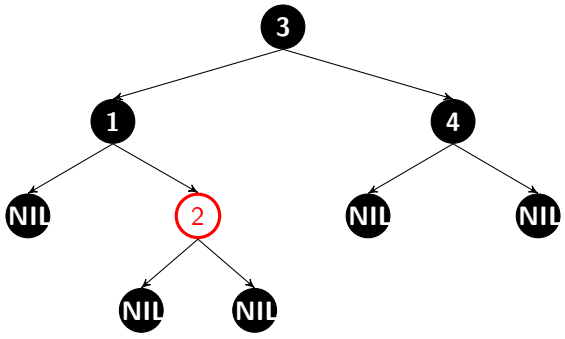


b) Draw (or describe by using preorder traversal) all valid red-black trees that store the values $\{1, 2, 3, 4\}$. There are 4 valid red-black trees that can be created by inserting the numbers $1, 2, 3, 4$ in all possible orders. In order to arrive to this answer, I used my implementation of the Red Black Tree Data Structure and I produced all possible permutations (24) of the set $(1, 2, 3, 4)$, inserted them into the Red Black Tree and produced a set of only 4 unique Red Black Trees. You can find the solution to this inside the directory, "RedBlackTreePermutations".









Problem 9.2

Implement a red black tree (with integer nodes), closely following the specifications and algorithms from the lecture. Make sure you handle errors appropriately by printing messages or throwing exceptions. Your implementation has to be along the interface below with the following or equivalent components:

```
enum Color {RED, BLACK};
struct Node
{
int data;
Color color;
Node *left, *right, *parent;
};
class RedBlackTree
{
private:
Node *root;
protected:
void rotateLeft(Node *&);
void rotateRight(Node *&);
public:
RedBlackTree();
void insert(int);
void delete(Node *&);
Node * predecessor(const Node *&);
Node * successor(const Node *&);
Node * getMinimum();
Node * getMaximum();
Node * search(int);
};
```

Below you can find the definition of my RedBlackTree class which implements the methods expressed in the homework sheet and also several other helper methods. The definition of this class is inside the C++ file "RedBlackTree.cpp". A test function, based off ,Problem 9.1.a, is inside the main function of the file "testRedBlackTree.cpp".

```
enum Color {RED, BLACK};

struct Node {
    int data;
    Color color;
    Node *left, *right, *parent;
};

class RedBlackTree {
    private:
        Node *root;

        Node *NIL;
        void createNILNode(Node *node, Node *parent);

        // Methods to read the Red Black Tree recursively from a given node
        void preOrderTraversalRecursive(Node *x);
        void inOrderTraversalRecursive(Node *x);
        void postOrderTraversalRecursive(Node *x);

        // Methods to modify the Red Black Tree
        void transplant(Node *u, Node *v);
        void insertFixUp(Node *z);
        void deleteNode(Node *node, int key);
        void deleteFixUp(Node *x);
        // Method to search the Red Black Tree recursively from a given node
        Node* searchTreeRecursive(Node *x, int key);

    protected:
        // Methods to rotate a node of the Red Black Tree
        void rotateLeft(Node *x);
        void rotateRight(Node *y);

    public:
        // Constructor
        RedBlackTree();

        // Method to insert element with data property key into Red Black Tree
        void insert(int key);
        // Method to remove element with data property key from Red Black Tree
        void deleteFromTree(int key);
        // Methods to find predecessor and successor of node x in Red Black Tree
        Node* predecessor(Node *x);
        Node* successor(Node *x);
        // Methods to find minimum and maximum node in Red Black Tree
        Node* getMinimum();
        Node* getMaximum();
        // Method to search for a given key inside the Red Black Tree
        Node* search(int key);

        // Methods to read the Red Black Tree from the root
        void preOrder();
        void inOrder();
        void postOrder();

        // Method to find the minimum of the subtree at given node x
        Node* minimum(Node *x);
        // Method to find the maximum of the subtree at given node x
        Node* maximum(Node *x);
};
```

Listing 1: Red Black Tree Class Implementation Declaration in C++