1 Objectives

In this lab you will focus on the following objectives:

1. Review basic binary tree data structure and operations
2. Develop familiarity with Red-black trees, and augmenting existing data structures

2 Tasks

1. Put your code in a folder to be zipped and turned in at the end.

2. Augment your implementation of BinaryTree into a class called RBTree with the following basic structure:

```cpp
class RBTree{
    private:
        struct TreeNode {
            int key;
            color_t color;
            TreeNode* left;
            TreeNode* right;
            TreeNode* parent;
        };
        TreeNode* root;
        // optional, but a good idea for efficiency
        // need to initialize outside the class declaration
        static TreeNode* const nil = new TreeNode({0, BLACK, nullptr, nullptr, nullptr});
    public:
        /* Fill in with methods */
    };

    // initialize nil
    RBTree::TreeNode* const RBTree::nil = new TreeNode({0, BLACK, nullptr, nullptr, nullptr});

```
where the type color_t is defined by the enum type

```c
enum color_t {
    RED,
    BLACK
}
```

3. You may find it useful to define methods to transform a color_t variable to a string for the purpose of displaying tree data. In C++, you can use an enum type as the argument to a switch statement!

4. Your class should have the following public methods. You may include other private methods to carry out the “standard” behaviors, such as inorder traversal.

(a) insert, to add new keys to the tree
(b) search, which, given a key, determines whether there is a node with that key in the tree
(c) minimum, which returns the smallest key in the tree
(d) maximum, which returns the largest key in the tree
(e) successor, for node with key k, returns the smallest key in the tree larger than k.
(f) inorder, which prints the keys in the tree in ascending order
(g) delete, which removes a given key from the tree, if it exists
(h) print, to display the contents of the tree, in a format of your choice, as long as it is clear.

5. Define as private methods the Red-Black Tree helper methods Right-Rotate, Left-Rotate, and transplant.

6. The insert and delete methods must use the self-balancing algorithms discussed during lecture for Red-Black trees. See the appendix for reproductions of the pseudo code.

7. Write a test program to demonstrate (clearly) the correctness of each of the above methods, displaying the tree after modifications.

8. Write some comparison code to test the efficiency of your RBTree against your BinaryTree structures (time to insert, delete, and search are most important).

9. Include a Makefile to build your code.

10. Include a README file to document your code, any interesting design choices you made, and answer the following questions completely and thoroughly:

(a) Summarize your approach to the problem, and how your code addresses the abstractions needed.
(b) What is the theoretical time complexity of your algorithms (best and worst case), in terms of the size of the tree? Be sure to vary the parameters enough to use the observations to answer the next questions!
(c) Use timing tools to study the cost of each of the data structure algorithms. Does the data align with the theoretical guarantees?
(d) How could the code be improved in terms of usability, efficiency, and robustness?
3 Submission

All submitted labs must compile with your provided Makefile and run on the COSC Linux environment.

Upload your project files to MyClasses in a single .zip file.

Turn in (stapled) printouts of your source code, properly commented and formatted with your name, course number, and complete description of the code.

Also turn in printouts reflecting several different runs of your program (you can copy/past from the terminal output window). Be sure to test different situations, show how the program handles erroneous input and different edge cases.

4 Bonus

(Up to 10 pts) Add a size attribute to each node – as described in the CLRS text, section 14.1 – and add methods to retrieve the kth smallest key of the tree. Time these methods (or count traversal operations) to verify the O(log n) runtime.

A Helper Methods

```c
RB-TRANSPLANT(T, u, v)
1  if u.p == T.nil
2     T.root = v
3  elseif u == u.p.left
4     u.p.left = v
5  else u.p.right = v
6     v.p = u.p
```

Figure 1: RBTree Transplant

```c
LEFT-ROTATE(T, x)
1  y = x.right
2  x.right = y.left
3  if y.left != T.nil
4      y.left.p = x
5  y.p = x.p
6  if x.p == T.nil
7      T.root = y
8  elseif x == x.p.left
9      x.p.left = y
10     else x.p.right = y
11     y.left = x
12     x.p = y
```

Figure 2: RBTree Rotate
RB-INSERT\( (T, z) \)
1 \( y = T.\text{nil} \)
2 \( x = T.\text{root} \)
3 while \( x \neq T.\text{nil} \)
4 \( y = x \)
5 if \( z.\text{key} < x.\text{key} \)
6 \( x = x.\text{left} \)
7 else \( x = x.\text{right} \)
8 \( z.\text{p} = y \)
9 if \( y = T.\text{nil} \)
10 \( T.\text{root} = z \)
11 elseif \( z.\text{key} < y.\text{key} \)
12 \( y.\text{left} = z \)
13 else \( y.\text{right} = z \)
14 \( z.\text{left} = T.\text{nil} \)
15 \( z.\text{right} = T.\text{nil} \)
16 \( z.\text{color} = \text{RED} \)
17 RB-INSERT-FIXUP\( (T, z) \)

Figure 3: RBTree Insert

RB-INSERT-FIXUP\( (T, z) \)
1 while \( z.\text{p}.\text{color} = \text{RED} \)
2 if \( z.\text{p} = z.\text{p}.\text{p}.\text{left} \)
3 \( y = z.\text{p}.\text{p}.\text{right} \)
4 if \( y.\text{color} = \text{RED} \)
5 \( z.\text{p}.\text{color} = \text{BLACK} \)
6 \( y.\text{color} = \text{BLACK} \)
7 \( z.\text{p}.\text{p}.\text{color} = \text{RED} \)
8 \( z = z.\text{p}.\text{p} \)
9 elseif \( z = z.\text{p}.\text{right} \)
10 \( z = z.\text{p} \)
11 LEFT-ROTATE\( (T, z) \)
12 \( z.\text{p}.\text{color} = \text{BLACK} \)
13 \( z.\text{p}.\text{p}.\text{color} = \text{RED} \)
14 RIGHT-ROTATE\( (T, z.\text{p}.\text{p}) \)
15 else (same as then clause
16 with “right” and “left” exchanged)
17 \( T.\text{root}.\text{color} = \text{BLACK} \)

Figure 4: RBTree Insert Fixup
\textbf{RB-DELETE}(T, z)

1. \( y = z \)
2. \( y\text{-original-color} = y\text{.color} \)
3. \textbf{if} \( z\text{.left} == T\text{.nil} \)
4. \( x = z\text{.right} \)
5. \textbf{RB-TRANSPLANT}(T, z, z\text{.right})
6. \textbf{elseif} \( z\text{.right} == T\text{.nil} \)
7. \( x = z\text{.left} \)
8. \textbf{RB-TRANSPLANT}(T, z, z\text{.left})
9. \textbf{else} \( y = \text{TREE-MINIMUM}(z\text{.right}) \)
10. \( y\text{-original-color} = y\text{.color} \)
11. \( x = y\text{.right} \)
12. \textbf{if} \( y\text{.p} == z \)
13. \( x\text{.p} = y \)
14. \textbf{else} \textbf{RB-TRANSPLANT}(T, y, y\text{.right})
15. \( y\text{.right} = z\text{.right} \)
16. \( y\text{.right}\text{.p} = y \)
17. \textbf{RB-TRANSPLANT}(T, z, y)
18. \( y\text{.left} = z\text{.left} \)
19. \( y\text{.left}\text{.p} = y \)
20. \( y\text{.color} = z\text{.color} \)
21. \textbf{if} \( y\text{-original-color} == \text{BLACK} \)
22. \textbf{RB-DELETE-FIXUP}(T, x)
RB-DELETE-FIXUP(T, x)

1. while x $\neq$ T.root and x.color == BLACK
   2. if x == x.p.left
      3. w = x.p.right
      4. if w.color == RED
         5. w.color = BLACK
         6. x.p.color = RED
         7. LEFT-ROTATE(T, x.p)
         8. w = x.p.right
   9. if w.left.color == BLACK and w.right.color == BLACK
      10. w.color = RED
      11. x = x.p
   12. else if w.right.color == BLACK
      13. w.left.color = BLACK
      14. w.color = RED
      15. RIGHT-ROTATE(T, w)
      16. w = x.p.right
      17. w.color = x.p.color
      18. x.p.color = BLACK
      19. w.right.color = BLACK
      20. LEFT-ROTATE(T, x.p)
      21. x = T.root
      22. else (same as then clause with “right” and “left” exchanged)
      23. x.color = BLACK

Figure 6: RBTree Delete-Fixup