Implementation

You are going to implement Binary Search Tree (*BST*) and Red-Black Tree (*RBT*) algorithm with the following functionalities. You are going to store population data of cities in the trees. **Keep in mind that you may need more functions (i.e. helper functions) than specified in skeleton code.**

Hint: Since most of the functions are similar, after implementing one of the algorithms, you may slightly change the structure to implement the other one.

Tree Structure

Node structure is already given in skeleton codes. You should not delete that structure and implement on your own. You should set the color attribute of the struct as **0 for "black" nodes** and **1 for "red" nodes.** Otherwise, your code may get an error during the evaluation of color rules in main function.

TREE-INSERT [20 pts]

After you managed to create a structure for nodes and trees, you need to implement "insertion" function. Beware that the insertion procedure differs in *RBT* to preserve the *RBT* rules along with *BST* property (*left subtree values are smaller than the parent, right subtree values are greater than the parent*). You may need additional helper function to fix the tree structure after insertion in red-black trees. This function can be void.

<u>Important Note</u>: *If you encounter the same population value during comparison in insertion procedure, you should insert the node in the right subtree. So,* ***the new inserted node should be treated as the <u>successor of the compared node</u> (e.g. to-be-inserted node (z), with value 20, one of the tree's nodes (x) with value 20, z should be placed in a relevant location in the right subtree of the z as its successor. Hint: This applies for before fix-up in RBTrees, after fixing the tree, z's location might be changed in the tree regarding different cases to preserve RBT rules (e.g. z might become x's parent after fixup calls.).***

TREE-DELETE [20 pts]

This function is used to delete a given node from the tree. Even though deletion procedure is similar in *BST* and *RBT,* deletion procedure in *RBT is* slightly different to ensure that tree structure meets the *RBT* rules. You may need additional helper function to fix the tree structure after deletion in red-black trees. This function can be void.

TREE-SEARCH [10 pts]

The searching procedure is same in *RBT* and *BST*. Aim is to retrieve the node with specified value. This function should return a pointer to the Node structure.

Sorting - Walking Functions [10 pts]

Using the tree, a function should return the ordered list in ascending order. You should implement the walking functions.

Other fundamental operations [10 pts]

Implement simple operations that are called **from the main function** such as finding **minimum, maximum, successor and predecessor.** These functions should return a pointer to the Node structure.

Deliverables

Code Structure [70 pts]

1. **redblacktree.cpp** and **bst.cpp** should be submitted. Skeleton code gives you the idea of which functions should be implemented. You may need helper functions, if so, please implement such functions. **Do not change the names** of the given functions.
2. Those cpp files **should work seamlessly with the main** function.
3. Your code should produce **log.txt, rb_out.csv** and **bst_out.csv**. The generation of these files should be done automatically when code is executed, so do not send these files. Expected outputs are given in Ninova, **ensure that your code's output is same**. **Beware that selected cities in the main.cpp can be changed while evaluating the solutions.**
    a. rb_out.csv should contain the population data along with cities in ascending (increasing) order for RBT, while bst_out.csv should contain data for BST.

Log Structure:

```
RBT:
Height: 24 // height of the red-black tree
Nodes: 13807 // total number of nodes in red-black tree
Min: Agdam // get city name with minimum population in red-black tree
Max: Tokyo // get city name with maximum population in red-black tree
BST:
Height: 12204 // height of the binary search tree
Nodes: 13807  // total number of nodes in binary search tree
Min: Agdam  // get city name with minimum population in binary search tree
Max: Tokyo // get city name with maximum population in binary search tree
```

```
Searching for Paris(B) with population 9904000 // City Name is given with color
as B if color is black, and R if color is red
RBT:
P(B):Guangzhou;L(B):Lagos;R(B):Moscow;S:Istanbul;Pr:Seoul // P: parent, L: left
child, R: right child, S: successor, Pr: predecessor of searched city (Paris in
this case)
BST:
P:Istanbul;L:Seoul;S:Istanbul;Pr:Seoul // there is no color information as this
is the binary search tree
Deleting a city with population 9904000 // deletion operation
RBT nodes: 13806 // number of nodes is decreased by one
BST nodes: 13806 // number of nodes is decreased by one
```

Running the code:

```
g++ main.cpp -o main
./main <DATASET_FILE_NAME>.csv out.csv v
```