Assignment 3

Batuhan Faik Derinbay derinbay18@itu.edu.tr
150180705

Part 1

After making sure the input CSV file (`sample.csv` in the example below) is in the same directory with the source code, you can compile and run the program using the following lines:

Compile and Run the Program

```
$: g++ -o 150180705.out main.cpp
$: ./150180705.out sample.csv
```

Part 2

Question 1)

In red black trees, asymptotic upper bounds for the insertion and search operations are the same for both the average and the worst case being $\log_2 n$. This is no coincidence however. Due to red black trees being balanced search trees, their height h which is on the order of $\log_2 n$ determines the complexity of both insertion and search operations. Therefore, $O(\log_2 n)$ is an asymptotic upper bound to average and worst cases of insertion and search operations on red black trees.

Let's prove our findings with mathematical induction. Assuming any sub-tree rooted at node k has **at least** $2^{bh(k)} - 1$ internal nodes where $bh(k)$ denotes the black nodes from k to a leaf, excluding the k itself. The base condition when k is a leaf node holds true where $bh(k) = bh(0) \to 2^0 - 1 = 0$ internal nodes. Generalizing this term, we can see that number of internal nodes α is always greater than our first assumption and can be simplified as follows: $\alpha \geq (2^{bh(k)} - 1) + (2^{bh(k)} - 1) + 1 \geq 2^{bh(k)} - 1$. By showing that $\alpha \geq 2^{bh(k)} - 1$ we guarantee that for the worst case scenario where the colors of nodes are consecutive $(B - R - B - R \cdots)$, height h of the tree is always less than or equal to twice the black height of the tree which can be denoted as $2bh(k) \geq h \to bh(k) \geq \frac{h}{2}$. Going back to our first assumption and plugging $bh(k)$ in, we get $n \geq 2^{\frac{h}{2}} - 1 \to n + 1 \geq 2^{\frac{h}{2}} \to \log_2(n+1) \geq \frac{h}{2} \to 2\log_2(n+1) \geq h$. Hence the upper bound is proven to be tree height dependent which is in order of $\log_2 n$.

Similarly, since the height of the tree determines the time complexity of the insertion operation and time complexity of recoloring while insertion (including rotations) has the upper bound of $O(1)$, upper bound for insertion operation in total is $O(1) * O(\log_2 n) = O(\log_2 n)$.

In conclusion, the average and worst time complexity upper bound of the search and insertion operations of red black trees is $O(\log_2 n)$.

Question 2)

Binary search trees (BST) don't have to be balanced. By adding the color attribute to the nodes and rebalancing the tree after insertion and deletion, red black trees maintain a balanced search tree and are able to perform operations in orders of $\log_2 n$ time with the same space complexity of BSTs. Advantage of using red black trees can be explicitly observed when working with ordered data, where BSTs are guaranteed to perform their worst case with the order of n. However, red black trees are guaranteed to perform with the order of $\log_2 n$.

Question 3)

For this question, rather than implementing five different methods, I propose a single method where each node holds the number of children that belong to a specific position in a mapping array. As an example, each player should have their position attribute as well as a `positions` array. If we map the positions with the following indices {PG:0, SG:1, SF:2, PF:3, C:4}, `positions[0]` holds the number of point guard children of the player node, `positions[1]` holds the number of shooting guard children of the player node, so on and so forth.

To further improve upon this idea, let's create a helper function `number_of_positions(k, p)` where k is the starting player node and p is the position of the player, that returns the number of player nodes with position p in sub-tree rooted at k, including k. Moreover, because NIL player nodes don't have a position, their return value is 0.

With these definitions in mind, a pseudo-code in C++ to the problem defined in Question 3 can be implemented as below with its algorithm given in Algorithm 1. Note that variable p denotes the positions and variable i denotes the i^{th} player with position p.

Algorithm 1 Find i$^{\text{th}}$ Player

```
Input  : player, p, i
Output: ith Player Node with Position p
if player is in p then
    a = 1
else
    a = 0
end
r = number_of_positions(player->left, p) + a
if i = r then
    return player
else if i < r then
    return find_ith_player(player->left, p, i)
else
    return find_ith_player(player->right, p, i-r)
end
```

Pseudo-code of Position Augmentation in RB Tree

```
enum Position { PG, SG, SF, PF, C };
Position p = PG;    // any other position would work
int i = 3;
string player_name = name_of_ith_player(root, p, i);

Player* find_ith_player(Player* player, Position p, int i){
    int is_same_position = (player->position == p) ? 1:0;
    int remainder = number_of_positions(player->left, p) + ↩
        is_same_position;
    if (i == remainder){
        return player;
    } else if (i < remainder) {
        return find_ith_player(player->left, p, i);
    } else {
        return find_ith_player(player->right, p, i-remainder);
    }
}

string name_of_ith_player(Player* player, Position p, int i){
    Player* ith_player = find_ith_player(player, p, i);
    return ith_player->name;
}
```