# CSE310 RedBlackTree Honors Report Journey Hancock

## Data Structures

This project implements a node data structure. A node has 5 attributes: an integer value for key, an integer value for color (0 = black and 1 = red), a node for parent, a node for left child, and a *node* for right child.

```
struct node {
    int key; //Key value
    int color; //Color: 0 = black, 1 = red
    node *parent; //Parent
    node *left; //Left Child
    node *right; //Right child
};
```

## Valid Execution

Firstly, the project can be compiled by running *make* to use the included make file. A valid execution of this project is:

```
./RBTree <I-File> <O-File>
```

In this case I-file is the input file that holds a Red Black Tree and O-file is the file where the resulting Red Black Tree will be printed if the *write* command is given. If the project is not executed correctly the program will print the following:

```
Usage: ./RBTree <ifile> <ofile>
```

If given a valid execution the program will respond to the following commands:

- Stop
    - On reading *stop* the program will print "Instruction: Stop" and stop the program.
- Read
    - On reading *read* the program will print "Instruction: Read".

    - Open the I-file input file. Read the line to get the amount of node declarations. Then iterate through the rest of the file to build the Red Black Tree.
    - Close the I-file.
- Write
    - On reading *write* the program will print "Instruction: Write".
    - Open the O-file output file. Print the current state of the Red Black Tree to the output file using the PrintTree function.
    - Close the O-file.
- PrintTree
    - On reading *PrintTree* the program will print "Instruction: PrintTree".
    - If the Tree is NULL (the root has a value of INT_MAX) the program will print "Error: Tree is NULL" to stdout.
    - Otherwise, the program will print the current state of the Red Black Tree to stdout using the modified preorder traversal discussed in Output Format.
- Insert *key*
    - On reading *Insert* the program will print "Instruction: Insert *key*".
    - If a node with the key is already in the Tree the program will print "Key: *key* already in tree" to stdout.
    - Otherwise, the program will perform BST insert to place the new node into the Tree. Then, it will execute the necessary Red Black Tree fix up cases to maintain the integrity of the Tree.
- Delete *key*
    - On reading *Delete* the program will print "Instruction: Delete *key*".
    - If a node with the key does not exist in the Tree the program will print "Key: *key* not in tree" to stdout.
    - Otherwise, the program will perform BST deletion to remove the node from the Tree. Then, it will execute the necessary Red Black Tree fix up cases to maintain the integrity of the Tree.
- Search *key*
    - On reading *Search* the program will print "Instruction: Search *key*".
    - If a node with the key does not exist in the Tree the program will print "Key: *key* not in tree" to stdout.
    - Otherwise the program will perform BST search to find the node. Then, it will print "Node key: *key* with color: *color*". Where *color* is either "Black" or "Red".
- Invalid Instruction
    - If the program does not recognize an instruction given it will print "Warning: Invalid Instruction".

## Format of Input File

The input file includes three parts. The first line is an integer value of how many node declarations there are in the input file, which totals to (amount of nodes - 1). The second line is the definition of the root node. The rest of the file is the remaining node declarations. The node declarations obey this format: parent node key, color, child type, child key, child color. The key spot can have an integer value, the color spot can have either a 0 (for black) or a 1 (for red). The child type spot can have either a "L" (for left) or "R" (for right). Below is an example input.

```
3
17 0 L 15 0
17 0 R 19 0
19 0 L 18 1
```

So, this input indicates there are three node declarations. Next, the node with key value 17 and color black is given as the root and has a left child with key value 15 and color black. The second node declaration indicates the node with key value 17 and color black has a right child with key value 19 and color black. The last line declares the node with key value 19 and color black has a left child with key value 18 and color red.

## Output Format

When given the *write* command the program will write the result of *printTree* to the given output file. This will be done using a modified preorder traversal, where the order of preorder traversal is followed but the parent of each node is printed before the current nodes value is printed. Since the root has no parent it is always the parent of the first node printed. Additionally, to indicate color the program uses ANSI escape codes. For redundancy, the program also includes the characters "B" or "R" around a key value to indicate its color. Whether a child is left or right is simply indicated by a "LEFT" or "RIGHT" included between the nodes. Below is an example output.

```
B17B ->LEFT-> B15B
B17B ->RIGHT-> B19B
B19B ->LEFT-> R18R
```

From the first line, the node has a key value 17 and color black with a left child with key value 15 and color black. Since there is no left child for the preorder traversal to follow it moves to the right. The next line indicates a node with a key value 17 and color that has a right child with key value 19 and color black. Since this node has a left child for the preorder traversal to follow it explores the left path. The last line describes a a node with a key value of 19 and color black that has a left child with a key value of 18 and color red.

## Implementation

- rotate
  - This function follows the insert process described in the slides. First, given the node x which serves as the bottom of the backbone for the rotation the type of rotation (Left or Right) is determined based on if x is the right or left node of its parent. Once the type of rotation is determined the pointers alpha, beta, gamma, a, b, and pi are set using the notation in the slides that is based on whether it is a left or right rotation. Additionally, the color of x and its parent are swapped, as dictated by the slides. The function also checks to see if the root will be changed to x and if so, it updates root to x so when root it returned at the end of the function root is properly updates for the rest of the program. Lastly, based on whether it is a left or a right rotation, the reassignments to the pointers declared at the start are made following the methods provided in the slides.
- printTree
  - This function is a modified preorder traversal where the root is the first node printed with its left child. Following preorder traversal, this goes down the left side of the tree as much as possible. Once it reaches a NIL node, the algorithm travels up the tree until it is able to explore a right child. Each time a node is printed, it is printed with either the left or right child the algorithm is currently moving onto. Additionally, in order to indicate color this function uses ANSI console escape codes to color the key of the node as well as the letters "B" or "R" around the key of a node.
- Insert
  - This function first follows BST insertion, where the key of the node to be inserted is compared to the current node starting at the root. As the algorithm moves down the Tree it moves to the left child if the to be inserted key is less than the current key and to the right child if the to be inserted key is greater than the current key. Once the algorithm reaches a NIL node, the new node is inserted as the left child if the key is less than the current key or as the right child if the key is greater than the current key.
  - Since the to be inserted node is colored red, to ensure the Tree still follows the properties of a Red Black Tree, I implemented the fix up cases discussed by the slides. First, if the inserted node x is red, the parent of x is red, and the uncle of x is red then the Case 1 fixup is applied. This changes the grandparent of x to red, the parent of x to black, the uncle of x to black, and points x to the grandparent of the previous x. If the color of x is red, the parent of x is red, and x is near its uncle (x and its uncle are both left or both right children) then the Case 2 fixup is applied. This applies a rotation using x and its parent as the backbone and points x to its red child.

    If the color of x is red, the parent of x is red, and x is far from its uncle (x and its uncle are not both left or both right children) then the Case 3 fixup is applied. This applies a rotation using the parent of x and the grandparent of x as the backbone. These cases are checked within a while loop that runs while the color of x is red and the parent of x is red, so that once this violation is solved the Tree is a valid Red Black Tree.

- treeDelete
    - This function first follows BST deletion. Where the Tree is searched through until the key of the node to be deleted is found. If the to be deleted node has no children or one child (Cases 1 and 2 of BST deletion) then the node is simply cut out and if there is a child it is moved into the spot of the deleted node. If the to be deleted node has two children (Case 3 of BST deletion) the successor node is found, cut out, and moved into the spot of the to be deleted node. The only possible cases for the successor node are Case 1 and 2, which are appropriately applied when the successor node is cut out. This BST deletion is done in a separate function called bstDelete in order to return the deleted node so it can be used in the Red Black Tree deletion fixup cases. When the successor is placed in the spot of the to be deleted node, the color of the to be deleted node is kept as described by the slides.
    - Since deleting a node can result in a Tree that does not follow the properties of a Red Black Tree, I implemented the fix up cases discussed by the slides. First, if the node that moves into the spot of the deleted node (called x) is red then it is made black and the Tree is now a valid Red Black Tree. However, if the sibling of x (called w) is red then the Case 1 fixup is applied. This applies a rotation using w and the parent of w as the backbone. If w is black and w has two black children then the Case 2 fixup is applied. This changes w to red and if x is red it is changed to black. If the color of w is black, the child of w near x is red, and the other child of w is black then the Case 3 fixup is applied. This applies a rotation using the child of w near x and w as the backbone. If the color of w is black and the child of w far from x is red then the Case 4 fixup is applied. This applies a rotation using w and the parent of w as the backbone and makes the child of w near x red. These cases are checked within a while loop that runs while x is not the root of the Tree and x is black, so that once this condition is false the Tree is a valid Red Black Tree. If Case 4 is applied, the Tree is guaranteed to be valid so the while loop is exited after Case 4 is finished.
- search
    - This function uses the search function of a BST. Starting from the root, the algorithm moves to the left or right depending if the target key is less or greater than the current node. This algorithm will continue until the current node's key is the target key or the algorithm reaches a NIL node.
- Overall design
    - This implementation of a Red Black Tree depends upon following the root node which is declared in main.cpp. Since it is possible for the root node of the Tree to be

changed by commands such as *Delete*, *Insert*, or functions like rotate, the corresponding functions in rbtree.cpp have a return type of node* which is used to update the root node if it is changed. If the root node is not changed the original root is returned by these functions to maintain the Tree.

- This implementation also uses a NIL node. This is declared in main.cpp with INT_MAX as the key. This way, comparisons can be done using the key to avoid conflicting cases where the node replacing the position of a deleted node is a NIL node. Additionally, the color of the NIL node is black as described by the rules of Red Black Trees. The parent, left child, and right child attributes of the NIL node are declared as nullptr in order to avoid any conflict since this one NIL node is used to for every leaf node in the Tree. Lastly, the NIL node is passed to each function in rbtree.cpp to be used for comparisons and allow the algorithms to determine if they have reached the end of the tree.

## Examples

**Example 1**

Assume the I-file has the following content:

```
7
11 0 R 14 0
14 0 R 15 1
11 0 L 2 1
2 1 R 7 0
7 0 R 8 1
7 0 L 5 1
2 1 L 1 0
```

The following execution is ran:

```
./RBTree I-file O-file
```

The instructions given from stdin are:

```
Read
Insert 4
Write


```

```
PrintTree
Stop
```

Since this is a valid execution, the following will be printed to stdout:

```
Instruction: Read
Instruction: Insert 4
Instruction: Write
Instruction: PrintTree
B7B ->LEFT-> R2R
R2R ->LEFT-> B1B
R2R ->RIGHT-> B5B
B5B ->LEFT-> R4R
B7B ->RIGHT-> R11R
R11R ->LEFT-> B8B
R11R ->RIGHT-> B14B
B14B ->RIGHT-> R15R
Instruction: Stop
```

Additionally, the following will be printed to the O-file:

```
B7B ->LEFT-> R2R
R2R ->LEFT-> B1B
R2R ->RIGHT-> B5B
B5B ->LEFT-> R4R
B7B ->RIGHT-> R11R
R11R ->LEFT-> B8B
R11R ->RIGHT-> B14B
B14B ->RIGHT-> R15R
```

**Example 2**

Assume the I-file has the following content:

```
3
17 0 L 15 0
17 0 R 19 0
19 0 L 18 1
```

The following execution is ran:

```
./RBTree
```

Since this is not a valid execution, the following is printed to stdout:

```
Usage: ./RBTree <ifile> <ofile>
```

**Example 3**

Assume the I-file has the following content:

```
3
17 0 L 15 0
17 0 R 19 0
19 0 L 18 1
```

The following execution is ran:

```
./RBTree I-file O-file
```

The instructions given from stdin are:

```
Read
Delete 15
Write
Search 17
PrintTree
Stop
```

Since this is a valid execution, the following will be printed to stdout:

```
Instruction: Read
Instruction: Delete 15
Instruction: Write
Instruction: Search 17
Node key: 17 with color: Black
Instruction: PrintTree
B18B ->LEFT-> B17B
```

```
B18B ->RIGHT-> B19B
Instruction: Stop
```

Additionally, the following will be printed to the O-file:

```
B18B ->LEFT-> B17B
B18B ->RIGHT-> B19B
```