ECE 522 Project: Trees, Trees, and More Trees

You have just been hired by a new company, which is planning to release a set of Rust libraries (crates) to provide faster, more efficient data structure solutions.

A good example of an efficient data structure is the Red-black tree as it provides means to insert, delete, and search the structure in O(log n) time. You can start researching red-black tress at https://en.wikipedia.org/wiki/Red-black_tree. The red-black tree is a binary search tree that adds logic to rebalance after inserts. Within this operation, it is crucial to know when to stop "balancing"—which is where the inventor thought to use two colors: **red** and **black**.

The red-black tree is a binary search tree that satisfies a set of rules:

1. The root node is always black
2. Each other node is either red or black
3. All leaves are considered black
4. A red node can only have black children
5. Any path from the root to its leaves has the same number of black nodes

By enforcing these rules, a tree can be programmatically verified to be balanced. How are these rules doing that? Rules 4 and 5 provide the answer: if each branch has to have the same number of black nodes, neither side can be significantly longer than the other unless there were lots of red nodes.

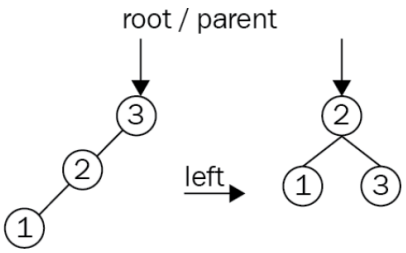
Like the binary search tree, each node in a tree has two children, with a key either greater than, equal to, or less than that of the current node. In addition to the key (as in a key-value pair), the nodes store a color that is red on insert, and a pointer back to its parent. Why? This is due to the required rebalancing, which will be described later. A typical node can be described as follows:

```
use std::cell::RefCell;
use std::rc::Rc;
#[derive(Clone, Debug, PartialEq)]
enum NodeColor {
    Red,
    Black,
}
type Tree = Rc<RefCell<TreeNode<u32>>>;
type RedBlackTree= Option<Tree>;
struct TreeNode<T> {
    pub color: NodeColor,
    pub key: T,
    pub parent: RedBlackTree,
    left: RedBlackTree,
    right: RedBlackTree,
}
```

Using these nodes, a tree can be created just like a binary search tree. In fact, the insert mechanism is exactly the same except for setting the parent pointer. Newly inserted nodes are always colored red, and, once in place, the tree might violate some of the five rules. Only then is it time to find and fix these issues.

After an insert, the tree is in an invalid state that requires a series of steps to restore the red-black tree's properties. This series, comprised of *rotation* and **recolor**, starts at the inserted node and goes up the tree until the root node is considered valid. In summary, a red-black tree is a binary search tree that is rotated and recolored until the balance is restored.

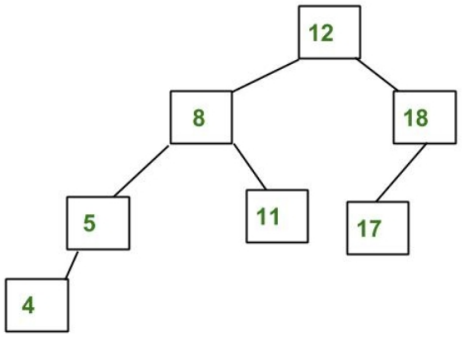
Recolour is simply changing the color of a specified node to a specific color, which happens as a final step when doing tree rebalancing. **Rotation** is an operation on a set of three nodes: the current node, its parent, and its grandparent. It is employed to fold list-like chains into trees by rotating either left or right around a specified node. The result is a changed hierarchy, with either the left or right child of the root node on top, and its children adjusted accordingly:



Clearly, this example is too simple, and it can only happen within the first few inserts. Rotations require recolors after redefining the hierarchy of a set of nodes. To add further complexity, rotations regularly occur in succession.

Another self-balancing binary tree is AVL (Adelson-Velsky and Landis) trees. You can start researching AVL trees at https://en.wikipedia.org/wiki/AVL_tree. An AVL tree has the following properties:

- The sub-tree of every node differ in height by at most one.
- Every sub-tree is an AVL tree.



The above tree is AVL because the differences between the heights of the left and the right subtrees for every node is less than or equal to 1.

Most of the binary search tree operations (e.g., search, max, min, insert, delete.. etc.) take O(h) time where h is the height of the tree. The cost of these operations may become O(n) for a skewed binary tree. If we make sure that the height of the tree remains O(Log n) after every insertion and deletion, then we can guarantee an upper bound of O(Log n) for all these operations. The height of an AVL tree is always O(Log n), where n is the number of nodes in the tree.

To make sure that the given tree remains AVL after every insertion, we must augment the standard binary search tree insert operation to perform some rebalancing. Therefore, rotations (left and right rotation) are applied to rebalance the tree.

Both red-black trees and AVL trees are great self-balancing binary trees. Both appeared around the same time, yet AVL trees are considered to be superior thanks to a lower height difference between the branches. Regardless of which tree structure is used, both are significantly faster than their less complex sibling, the binary search tree.

Project Requirements

Part 1: You are required to create your own implementation for a red-black tree. Your library must allow the user to perform the following operation:

1- Insert a node to the red-black tree.
2- Delete a node from the red-black tree.
3- Count the number of leaves in a tree.
4- Return the height of a tree.
5- Print In-order traversal of the tree.
6- Check if the tree is empty.
7- Print the tree showing its colors and structure. (Using println!("{:#?}",tree); is NOT sufficient)

Part 2: You are required to create your own implementation for a AVL tree. Your library must allow the user to perform the following operation:

1- Insert a node to the AVL tree.
2- Delete a node from the AVL tree.
3- Count the number of leaves in a tree.
4- Return the height of a tree.
5- Print In-order traversal of the tree.
6- Check if the tree is empty.
7- Print the tree showing its structure. (Using println!("{:#?}",tree); is NOT sufficient)

For both parts 1 and 2, you must use Rust smart pointers (preferably Rc and RefCell). Do **not** use raw pointers or unsafe code in any of your code.

To assist you with these parts, please ponder (and answer) the following questions as you go:

1- What does a red-black tree provide that cannot be accomplished with ordinary binary search trees?
2- Please add a command-line interface (function main) to your crate to allow users to test it.
3- Do you need to apply any kind of error handling in your system (e.g., panic macro, Option<T>, Result<T, E>, etc..)
4- What components do the Red-black tree and AVL tree have in common? Don't Repeat Yourself! Never, ever repeat yourself – a fundamental idea in programming.
5- How do we construct our design to "allow it to be efficiently and effectively extended"? For example. Could your code be reused to build a 2-3-4 tree or B tree?

Part 3. You had a discussion with your project manager about whether a Red-black tree or AVL tree has a better performance in insertion and search time. To have the final say in this argument, you thought that you should do some benchmarking to test the two trees in some of the worst-cases.

To decide on the test cases, you thought that the worst case for a binary search tree is when elements are inserted continuously in increasing or decreasing values (e.g., 3, 5, 7, 8, 11,…). On the other hand, the worst case for searching would be when we search for elements that are located at the top or at the bottom of the tree.

Hence, this is what you are planning to do to benchmark the two trees:

for *tree_size* in (10,000, 40,000, 70,000, 100,000, 130,000) **do**:
    Start by creating an empty tree.
    Values with *tree_size* are inserted into the tree.
    A search is conducted for the (*tree_size*/10) lowest values.
end

For example, the first benchmark case would be inserting 10K elements in the tree and then search for the 1000 lowest elements. **Please benchmark insert and search separately!**

Save the benchmark results and illustrate them using the appropriate graphs and charts. Provide your own comments concluded from the results. Which data structure is more efficient? Do you think we need to accommodate other test cases? Do you think we need to include additional data structures in the benchmarking to perform as the baseline (i.e., binary search tree)?

Deadlines & Submission Instructions

The company has now decided to get you to start rewriting the two libraries for the red-black tree and AVL tree. The company will launch your crates on **December 8th, 2023 @ 11:59 p.m M.S.T.**; this is a hard, absolutely unmovable deadline. Your performance will be measured on:

- The accuracy of the delivered code with the requirements above.
- The performance (efficiency) of the code.
- Bonus credits will be given for new additional features beyond the old version.

By the deadline, you must deliver:

- A copy of the source code of the two libraries.
- A design document outlining:
    - Major innovations – Additional to the project specification.
    - A detailed rationale for your augmented decisions with regard to the above design questions.
    - A list of known errors, faults, defects, missing functionality, etc. Telling us about your system's limitations will score better than letting us find them!
    - A user manual
    - A 2-minute video highlighting the new system – marketing is everything ☺.

You must also demo the code live to the company (i.e. the Professor and the T.A.) between **December 11th, 2022** and **December 12th, 2022** inclusive.

Please hand in all components by submitting via eclass to the group account, and hence all sub-components, by definition, must be machine-readable. Also, file types should be only .pdf or zipped files for your Rust projects.

Good luck and happy coding!