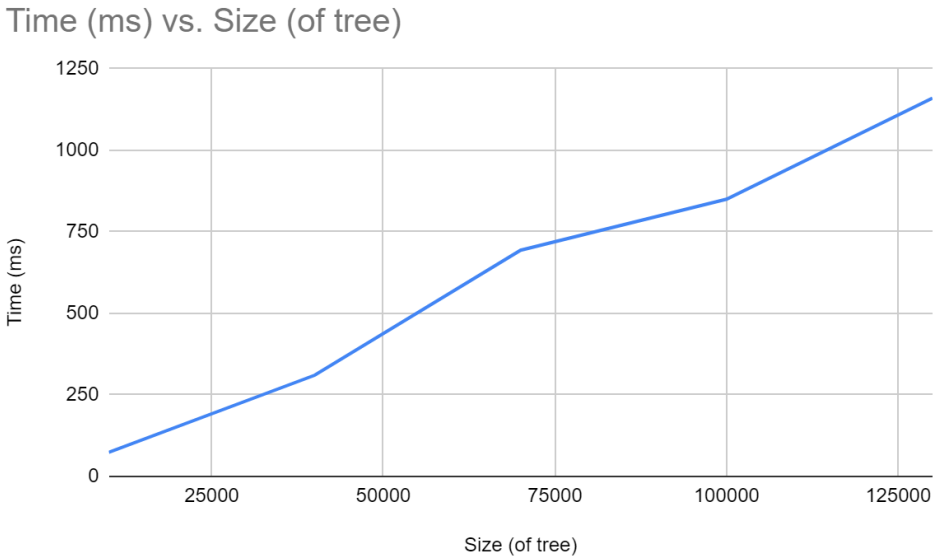
Design Notes

1. A red black tree provided a number of functionality that an ordinary binary search tree does not. First and foremost is the rebalancing and recolouring. Between those two the overall tree is kept balanced between the right and left side. That way one branch is not massively larger than any other. In turn this increases the efficiency of searching for a specific value. Since each branch is relative the same depth search will take generally the same amount of time for any value in the tree. In an ordinary binary search the time varies due potential differences in the branches, it may take longer if the value is in a long branch or shorter in a short branch.
2. A number of error handling has to be and is accounted for. Due to the option of deleting nodes existing the RedBlackTree is an optional. This way the system will not necessarily crash if the required is not found/met (instead returns None). Similarly panic macros are added in the case where an operation (i.e., insert) fails to perform correctly.
3. Both are a tree based method. That is to say they both host a number of values that connect to each other through a parent child relationship (one's the parent the other the child). Both have values less than the current value go to the left branch while greater than goes to the right. Also, both methods are self-balancing. Once a change has been made the tree rebalances itself to ensure both left and right sides remain similar in size. However, there is a difference in the methodology in re-balancing. Red Black Trees use colors to ensure branches remain relatively balanced, only shifting the tree if need be. While AVL trees go straight to shifting the tree.
4. This is difficult because despite the two methods' similarity there are subtle but major differences between the two. The first and foremost is that the Red Black Tree requires a color for each of its nodes while AVL does not. As such many instances where code could be reused has to be tweaked and changed. From the enums used to define a node to the implementation of insert. However, some functionality can be shared between the two trees as well as any new kinds of trees. The shifting methods can be reused since they rely on the structure of the tree and does not involve the colors (Red Black Tree will try to recolor before shifting). Similarly functions like counting depth or leaves can be reused since they don't require changing the base tree.

Benchmarking for Red Black Trees

To Benchmark Red Black Trees, elements were inserted into an initialized red black tree. The elements were inserted in ascending order (for 0 upto the target size of the tree). After inserting the elements, the bottom 10% elements of the tree are searched for. The following table is the summary of the result of the benchmark tests:

Size	Time
10000	73.73ms
40000	310.16ms
70000	693.61ms
700000	849.95ms
130000	1.16s



Unfortunately, we were not able to finish AVL Trees in time. We have the avl library that effectively works as a binary tree. We tried to benchmark this in a similar way to Red-Black trees; however, the stack was overflowing so we decided to leave the benchmarks tests to the red-black tree.

Limitations:

Due to the time constraints, we were unable to finish all the functionality for the project. Here are the known errors and defects:

- The Red-Black Tree would sometimes drop nodes after inserting. When starting with a fresh tree, the following insert sequence would drop the 0th insert: 0 → 1 → 2 → 3 → 4.
- Inserts, for both trees, do not work as a set. Duplicate insert will result in duplicate nodes in the tree. This means that if 4 was inserted twice, there would be 2 nodes with the data 4.

- The AVL Tree, for now, works as a regular binary tree. We were not able to finish the rebalancing. We have the right rotation and left rotation done but we had some troubles tying the available functions together in order to properly implement the balancing after an insert/delete. Given that, the inserts and deletes do work as regular binary tree inserts and deletes.

User Manual

Open the terminal in the appropriate directory.

- If you would like to use the avl trees, open the terminal inside avl/
- If you would like to use red-black trees, open the terminal inside redblacktree/

Run $ cargo run

1. Enter the value of the starting Node
2. Enter the next command according to what you would like to do:
    - What do you want to do with the tree?
    - Type i if you want to insert into the tree
    - Type d if you want to delete data from the tree
    - Type c if you want to know how many leaves are in the tree
    - Type h if you want to know the height of the tree
    - Type o if you want to print the tree in-order traversal
    - Type e if you want to check if the tree is empty
    - Type p if you want to print the tree itself with direction
    - Type s if you want to know how many nodes the tree has
3. Type exit to exit the program

For the benchmark test, open the terminal inside benchmark/ and run the command: $ cargo run