Analysis of Algorithms

BLG 335E

Project 3 Report

Fatih Baskın

baskin21@itu.edu.tr

Faculty of Computer and Informatics Engineering

Department of Computer Engineering

Date of submission: 29.12.2023

1. Implementation

1.1. Differences between BST and RBT

Binary search trees satisfy only one criterion, nodes in the left subtree are smaller than the current node, and nodes in the right subtree are larger than the current node. This way, you can search and find a node by making comparisons with the current node. But having only one criterion is a terrible choice since portions of the input stream might be sorted and the tree will be more like a linked list where search time is linear. Therefore we need to make sure that some level of balance must be preserved in the tree during the insertion.

Red-black trees were devised to address these issues of binary search trees, in addition to left-right subtree rules, there are some additional rules. They are:

- Nodes are colored red and black.
- Root is always a black node.
- Leaves are some TIL nodes that don't have a value and are black.
- If a node is red, then its parent is black.
- Black depth of every leaf is equal.

When insertions and deletions are done by preserving these rules, the height is upper bounded by $O(log(n))$ instead of linear $O(n)$, regardless of the input type and size. It is quite obvious in the experiment results:

	Population1	**Population2**	**Population3**	**Population4**
RBT	21	24	24	16
BST	835	13806	12204	65

Table 1.1: Tree Height Comparison of RBT and BST on input data.

As can be seen in the result, height differences in the binary search trees were quite severe, and in some cases, their heights were similar or very close to the input size, note that the input size was 13087. Red-black tree heights also fluctuated depending on the input size, but they were on a logarithmic scale. Since $\lceil 13087 \rceil = 14$ and the red-black tree's minimum heights and maximum heights were in the range of $14(log(n)) < 16 \texttt{ (minimum observed height) } < 24 \texttt{ (maximum observed height) } < 28(2 \times log(n))$. As the experiment results show, the height of a red-black tree is less susceptible to being affected by the input, compared to binary search trees, and their heights are logarithmically bounded.

1.2. Maximum height of RBTrees

The maximum height of a red-black tree is upper bounded by $O(log(n))$. To be exact, height is always smaller than $2 \times log(n+1)$. The proof is made using intuition. If you combine red nodes with black nodes, then you would have a 2-3-4 tree, in which a node has 2, 3, or 4 children. It is possible to have a fluctuation of colors from red to black in the original tree, this means every black node (except the root) has a red parent and every red node has a black parent, in this case, almost half of the nodes would be red and other half would be black. In this tree, there would be $n+1$ TIL leaf nodes, and when we merge these red-black nodes, there will still be $n+1$ TIL nodes and a perfectly balanced 2-3-4 tree.

Since the lower bound for the number of leaves is dependent on the tree height, then $n+1 \geq 2^{\bar{h}}$ where $\bar{h}$ is the height of 2-3-4 tree or the black height of the original tree. Also, since almost half of the nodes were red, it is possible to say that $\bar{h} \geq \frac{h}{2}$ and if we combine these, we will have: $n+1 \geq 2^{\bar{h}} \geq 2^{\frac{h}{2}}$ and if you take the logarithm in the end $log(n+1) \geq \frac{h}{2}$, $h \leq 2 \times log(n+1)$. Therefore height is upper bounded logarithmically.

1.3. Time Complexity

	Insertion	**Deletion**	**Search**
BST	$O(n)$	$O(n)$	$O(n)$
RBT	$O(log(n))$	$0(log(n))$	$O(log(n))$

Table 1.2: Worst case complexities of search, insertion and deletion

Insertion and deletion operations include searching, first, you need to search to tree to find the place to add or remove the node. Therefore their complexities depend on the search.

The search operation is dependent on the height of the tree, since in the search operation, the algorithm traverses the tree dependent on the values. If the tree is like a linked list, the worst case for binary search trees, then the complexity would be linear $O(n)$ since you need to go to the furthest path. But in the red-black trees, height is upper bounded by $2 \times log(n+1) = O(log(n))$, and even if you go down the furthest path, you will have the complexity of $O(log(n))$. Insertion and deletion operations depend on the complexity of the search and finding the successor. In the BST, you insert the node on the most suitable spot you have found, which is very similar to the search algorithm and the upper bound is $O(n)$. The deletion operation requires you to find the node first, which is a search that, has $O(n)$ complexity, and then the right subtree is placed on the successor's left, again there is a search involved for the successor which is upper bounded by $O(n)$ and in the end deletion has $O(n)$ complexity.

In the deletion and insertion of the RBT, there are search and finding successor operations, similar to binary search trees. But their complexities are bounded by $O(log(n))$. In RBTs, there are also rebalancing acts done after the insertion or deletion, which might start from the current node and go until the root, which implies the height, so these rebalancing operations also have $O(log(n))$ complexity.

To conclude, operations of BST have $O(n)$ complexity since it is very likely to have unbalanced BST. Since RBTs are guaranteed to be balanced to a certain extent (height being less than $2 \times log(n+1)$) the complexity of those operations are $O(log(n))$.

1.4. Brief Implementation Details

1.4.1. Insertion in BST

Implementing insertion in BST was quite simple, you traverse the tree and find the suitable place to insert the node into and insert it directly. As explained above, this has $O(n)$ complexity.

1.4.2. Deletion in BST

Deletion was a little bit tricky compared to insertion in the BST, since a node can have some children nodes and we need to decide where to put them. The left subtree of the current node must be linked to the left of the successor to the current node and the right subtree must be linked to the parent of the current node, replacing the current node. There are some edge cases present, If the right subtree doesn't exist, then the left tree should replace the current node. Also, if the parent is not present, i.e. current node is the root, then we need to replace the root with the node being replaced as the current node. These required extensive if-checks in the code. Again, finding the correct node and searching for successor operations makes this algorithm in $O(n)$ complexity.

1.4.3. Insertion in RBT

To make operations simple, initially the new node is thought to have a red color, to preserve the black depth. The node is placed in a suitable place, by traversing through the tree. This is very similar to the search in RBTs. After the insertion is done, the parent node is checked, if it is black, we are safe. But if it is red, this is a forbidden case and we need to fix this issue by using recoloring. There are some cases where recoloring cannot resolve this issue, then we use rotations. There are two types of rotations, left rotation and right rotation. Rotations are used to fix left-right cases into left-left cases and right-left cases into right-right cases and fix those left-left and right-right cases.

Recoloring will work upwards until we can't fix the tree by recoloring or we reach the root. If we reach the root, we make it black. If we cannot fix the issue with the tree by recoloring, we use rotations. Rotations have constant runtime, they have $O(1)$ complexity, and recoloring has $O(log(n))$ complexity since they run towards to root until they reach a case where they cannot fix or reach the root. If recoloring reaches the root, the root is colored black, this implies the tree height is increased by one.

To sum up, including searching, recursive fixup (recoloring), and rotations where recoloring is unfeasible has a combined complexity of $O(log(m))$

1.4.4. Deletion in RBT

The Deletion procedure is very complicated and has different edge cases. Firstly, you search the tree to find the node to be deleted. This has $O(log(n))$ complexity. Then, after finding the node, we need to find the node which will replace it. So there are different cases:

- The current node is red and leaf, remove with no problem.
- The current node has only one child, in this case, the current node has to be black, and the child node has to be red, replace the current node with the child node then make the child node black, and finally remove the current node.
- The current node is black, and it is leaf or has two children. In this case, things start to get complicated. Firstly there is the case of double black, meaning the current node is black and the node to replace the current node is also black. To explain the cases:
  - If the current node is black and it has no children, it needs to be replaced by a TIL node which is black, meaning a double black.
  - If the current node is black and has two children, we need to find its successor to replace it with, if the successor is also black, again this is a double black case.
  - If the successor is red, no worries, this is not a double black case, just recolor replaced node with black.

So, fixing the double black is quite tricky and I resorted to both textbooks and online sources to find a proper solution. To start with, some cases are easily solvable using recoloring and rotations, others require to shift this double black issue to the parent and fix double black recursively. If double black has reached the parent, this means the tree has lost a black height. Some of the cases are briefly explained below:

- If the parent is red and there is a black sibling (if the current node is double black and the parent is red, there has to be a sibling with black color) this is fixable by applying rotations and recoloring, depending on the children of the sibling, no need to push double black up.

- If the parent is black and there is a black sibling (well, if the current case is double black and the parent is black, there has to be a black sibling) this is fixable by applying rotations and recoloring if the sibling has (red, has to be red) children.
- If the parent is black and the sibling is black and the sibling is a leaf or its children are black, then the double black is shifted towards the parent, meaning fixing the double black procedure is called for the parent.

Since fixing double black can involve recursive calls until the root, `fixDoubleBlack` function has $O(log(n))$ complexity, upper bounded by the depth of the tree. With combined complexities of searching, finding the successor, and fixing double black, deletion has $O(log(n))$ complexity.

While thinking about the deletion scenarios, I have found almost 60 different cases (including symmetric cases) and coding according to those cases was turning the code hard to understand, a.k.a spaghetti code, therefore I have made my research to make my code cleaner, these are the sources that I have resorted to while writing the deletion:

- `https://medium.com/analytics-vidhya/deletion-in-red-black-rb-tree-92301e1474ea`
- `https://www.geeksforgeeks.org/deletion-in-red-black-tree/`
- `https://www.programiz.com/dsa/deletion-from-a-red-black-tree`
- Introduction to Algorithms, CLRS, Fourth Edition, p.346-348

In the end, I also checked the property of RBT using a checker, `isRBT` function is used for that.