Analysis of Algorithms

BLG 335E

Project 3 Report

MUSTAFA CAN ÇALIŞKAN
caliskanmu20@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 (BST) organize data such that each node has at most two children in a given order, with values on the left smaller and values on the right larger than the parent node. However, their performance can degrade if they are unbalanced, resulting in $O(n)$ time complexity in the worst case. In contrast, red-black trees, a type of self-balancing BST, maintain balance by assigning color attributes (red or black) to nodes and using rotations and color changes during insertion and removal. This balance ensures worst-case $O(logn)$ time complexity for operations such as insertions, and deletions, making red-black trees more consistent due to their balanced nature.

In a red-black tree, rules are in place to maintain its balance and key properties. When nodes are inserted, initially coloring them red helps maintain temporary balance. But if this violates rules against consecutive red nodes or disrupts black-height properties, the tree undergoes rotations and color changes. These adjustments rearrange the structure, potentially changing connections and node colors while respecting red-black tree rules. Similarly, during deletion, the tree ensures consistent black-height across paths. Removing a black node may trigger rotations and color changes to sustain balance and uphold properties. These rules dynamically guide structural changes, keeping the red-black tree balanced and efficient for insertion and deletion while maintaining its defined properties.

As seen from the experiment, the Red-Black Tree balances itself upon each node insertion using its rotation, transplant, and fixup functions, preventing excessive growth in height. In contrast, a Binary Search Tree easily becomes unbalanced due to lacking such functions, leading to a substantial increase in height. Through these functions, the Red-Black-tree attains the complexity level mentioned above.

Due to its more varied (neither strictly increasing nor decreasing) population of 4, it yields more balanced results in both trees. In cases of strictly increasing or decreasing population orders (such as population 2 and population 3), BST fails to maintain balance, causing the tree to elongate significantly. However, RBT remains unaffected by this scenario.

	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.

1.2. Maximum height of RBTrees

The maximum height of a red-black tree is based on the minimum requirements for the number of black nodes along any path from the root to a leaf, which is at least $\frac{h}{2}$, and the constraint that the length of the shortest path from the root to a leaf is at most $\log_2(n+1)$. According to these properties, no path can be more than twice as long as any other path, implying that the black height must be at least half the total height of the tree. Considering the path with the fewest nodes, denoted as P with k nodes, a binary tree with k nodes must have at least $2k-1$ total nodes. This implies that the total number of nodes, n, must be at least $2k-1$ or equivalently $k \leq \log_2(n+1)$. Since the minimal path also represents the black height, there exists a relationship between the total height of the tree (h) and the black height, given by $\frac{h}{2} \leq \log_2(n+1)$, which can be expressed as $h \leq 2\log_2(n+1)$. Consequently, it is proven that the maximum height of a red-black tree with n nodes is $2\log_2(n+1)$.

1.3. Time Complexity

1.3.1. Binary Search Tree Complexities

Functions	Worst-Case	Best-Case	Worst-Case Reason
preorder inorder postorder	O(n)	O(n)	Since it will traverse the tree in the number of nodes, in any scenario, both the best and worst cases result in O(n).
searchTree	O(n)	O(logn)	If the tree is completely unbalanced (like a linked list), it would have to search a completely linear list from start to finish, resulting in a complexity of O(n).
successor predecessor	O(logn)	O(1)	In the worst case scenario (if the tree is balanced), finding the successor involves locating the minimum element in that node’s subtrees. This situation has a complexity of O(logn).
insert	O(n)	O(logn)	In the worst-case scenario of an unbalanced tree, traversal to the tree’s furthest end is required for the addition of an element.
deleteNode	O(n)	O(logn)	In the worst-case scenario of an unbalanced tree, traversal to the tree’s furthest end is required for the deletion of given element.
getHeight	O(n)	O(n)	In the worst-case scenario, the tree is unbalanced, simply equating the length of the list to its height.
getMinimum	O(n)	O(1)	In the worst-case scenario, the tree is unbalanced, and the minimum element is at the very end, necessitating traversal of the entire tree.
getMaximum	O(n)	O(1)	In the worst-case scenario, the tree is unbalanced, and the maximum element is at the very end, necessitating traversal of the entire tree.
getTotalNodes	O(n)	O(n)	In the worst-case scenario, the tree is unbalanced, and to count the nodes, the entire list must be traversed linearly.

Table 1.2: Functions and Their Complexities

The complexity of public functions was determined based on the complexities of their helper functions since the sole purpose of public functions is to call private functions.

1.3.2. Red-Black Tree Complexities

Functions	Worst-Case	Best-Case	Worst-Case Reason
preorder inorder postorder	O(n)	O(n)	Since it will traverse the tree in the number of nodes, in any scenario, both the best and worst cases result in O(n)
searchTree	O(logn)	O(logn)	Due to the tree being completely balanced, it performs searches with a complexity of O(logn).
successor predecessor	O(logn)	O(logn)	Because the tree is balanced, it finds with a complexity of O(logn).
insert	O(logn)	O(logn)	Because the tree is balanced, it adds with a complexity of O(logn).
deleteNode	O(logn)	O(logn)	Because the tree is balanced, it deletes with a complexity of O(logn).
getHeight	O(n)	O(n)	To calculate its height, it needs to traverse all nodes' subtrees, essentially visiting all nodes. Consequently, the complexity becomes O(n).
getMinimum	O(logn)	O(logn)	Because the tree is balanced, it finds with a complexity of O(logn).
getMaximum	O(logn)	O(logn)	Because the tree is balanced, it finds with a complexity of O(logn).
getTotalNodes	O(n)	O(n)	To determine the number of nodes, it needs to traverse all nodes sequentially. Hence, the complexity becomes O(n).
leftRotation rightRotation	O(1)	O(1)	It only performs a rotation operation without any additional loops or recursive calls. Hence, it has a complexity of O(1).
fixInserting	O(logn)	O(logn)	It works to maintain balance after insertion, performing operations up to the height of the tree, operating with a complexity of O(log n).
fixDeleting	O(logn)	O(logn)	It works to maintain balance after deletion, performing operations up to the height of the tree, operating with a complexity of O(log n).
transplantOperation	O(1)	O(1)	It only performs a rotation operation without any additional loops or recursive calls. Hence, it has a complexity of O(1).

Table 1.3: Functions and Their Complexities

The complexity of public functions was determined based on the complexities of their helper functions since the sole purpose of public functions is to call private functions.

1.4. Brief Implementation Details

1.4.1. Binary Search Tree

The implemented functions and their tasks are as follows:

1.4.1.1. howManyNode

Counts the total number of nodes in the tree.

1.4.1.2. preorderWalk, inorderWalk, postorderWalk

Traverse the tree in preorder, inorder and postorder respectively, storing node data in an array.

1.4.1.3. searchNode

Searches for a node with a given value in the tree.

1.4.1.4. findMinNode and findMaxNode

Finds the node with the minimum and maximum values respectively.

1.4.1.5. calculateHeight

Calculates the height of the tree.

1.4.1.6. findSuccessor and findPredecessor

Finds the successor and predecessor nodes of a given node.

1.4.1.7. insertNode

Inserts a new node into the tree with a given value and name.

1.4.1.8. deleteNode

Deletes a node with a given value from the tree.

The functions provided above are helper functions (private) for the functions requested in the assignment description (public). When the functions in the assignment description are called, they call their own helpers.

The preservation of the Binary Search Tree property does not require the creation of additional helper functions (i.e. fixUp) for insertion and deletion. Instead, specific mechanisms exist within the insertion and deletion functions themselves to locate the position for insertion and removal. (For instance, if the value to be inserted is greater than the current node’s value, continually traverse to the right subtree; if smaller, go to the left subtree etc.)

1.4.2. Red-Black Tree

The implemented functions and their tasks are as follows:

1.4.2.1. howManyNode

Counts the total number of nodes in the tree starting from the root.

1.4.2.2. preorderWalk

Traverses the tree in preorder and appends the node data sequentially to an array.

1.4.2.3. inorderWalk

Traverses the tree in inorder (also sorting) and appends the node data sequentially to an array.

1.4.2.4. postorderWalk

Traverses the tree in postorder and appends the node data sequentially to an array.

1.4.2.5. searchNode

Searches for the given value in the tree and returns the node.

1.4.2.6. findMinNode

Finds the minimum node in the tree, the one with the smallest value.

1.4.2.7. findMaxNode

Finds the maximum node in the tree, the one with the largest value.

1.4.2.8. calculateHeight

Calculates the height of the tree.

1.4.2.9. findSuccessor

Finds the successor of a node.

1.4.2.10. findPredecessor

Finds the predecessor of a node.

1.4.2.11. leftRotation

Left rotation operation.

1.4.2.12. rightRotation

Right rotation operation.

1.4.2.13. fixInserting

Fixing (It checks whether the Red-Black Tree rules are being followed, and if there’s any violation, it corrects the inconsistency.) operation after inserting a node.

1.4.2.14. insertNode

Inserts a new node.

1.4.2.15. fixDeleting

Fixing (It checks whether the Red-Black Tree rules are being followed, and if there’s any violation, it corrects the inconsistency.) operation after deleting a node.

1.4.2.16. transplantOperation

Node replacement operation.

1.4.2.17. deleteNode

Deletes a node with the specified value.

The functions provided above are helper functions (private) for the functions requested in the assignment description (public). When the functions in the assignment description are called, they call their own helpers.

The Red-Black Tree, with its extra helper functions (fixDeleting, fixInserting, transplantOperation, and rotation functions), ensures and maintains its rules after every insertion and deletion by checking and preserving them.