Analysis of Algorithms

BLG 335E

Project 3 Report

Çağla Mıdıklı 150200011
midikli20@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

The fundamental difference between a binary search tree and a red-black tree is the inclusion of the color property in red-black trees. Additionally, the implementation rules of red-black trees make them more balanced than binary search trees. The color property in red-black trees is utilized to maintain the balance of the tree. Two nil nodes must always have the same black height value, and this rule ensures the preservation of balance in the tree.

The set of rules in a red-black tree ensures a more balanced structure. this set of rules : a. All nodes are either black or red. b. The NILs, or roots and leaves, are black. c. A node's parent is black if it is red. d. The number of black nodes on all simple pathways from any node x to a descendant leaf is equal to black-height(x). The advantage of these rules is that they ensure a more balanced distribution of the tree, reducing the time complexity in tree searches.

Due to being a more balanced tree structure, red-black trees (RB trees) are less affected about length of tree by the order of inputs compared to binary search trees (BST).

If the inputs are given in a sorted order, a binary search tree would become very long and unbalanced due to its inherent structure. For instance, if the inputs are inserted in descending order, the tree would heavily branch to the left, resulting in a length proportional to the number of nodes. However, in a red-black tree, even if the inputs are sorted, the tree automatically maintains balance within itself through the use of right rotate and left rotate functions.Red-black trees are one type of balanced tree that ensures that, in the worst scenario, fundamental dynamic-set operations take O(logn) time. In a red-black tree, the order of inserting inputs is less critical; however, inserting them in a random manner is still more optimal.In the case of a binary search tree (BST), the situation is different, and there is a significant difference between random and sorted insertions. Random insertion is more optimal in a BST.

	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 shown in the table, in Population 4, numbers have been inserted randomly, resulting in the most optimal outcome. In Population 2, the data has been inserted in a sorted order, and in this scenario, the height of the binary search tree experiences a significant change, whereas the height of the red-black tree changes in a more balanced

manner. In general, the red-black tree undergoes changes in a more balanced manner, whereas the binary search tree undergoes changes more unbalanced manner.

1.2. Maximum height of RBTrees

2(log(n+1)) All NIL nodes must have equal black height, and in a red-black tree, the highest height occurs when it goes in a sequence of black-red-black-red. This is because we aim to maximize the black height. 2**(bh(x)) - 1 internal nodes. The reason for performing a deletion by subtracting 1 is to ensure that both the root and NIL nodes are black. 2**(bh(x)-1) - 1 +.2**(bh(x)-1) - 1 +1 = 2**(bh(x)) - 1

According to property 4, at least half the nodes on any simple path from the root to a leaf, not including the root, must be black. As a result, the root's black-height must be at least h/2, and thus so n >= 2**(h/2) - 1 If we add 1 to both sides of the equation above and take the logarithm, we obtain the height. 2(log(n+1))

1.3. Time Complexity

1.3.1. RB Tree

searching:every case= O(logn),It traverse only a single branch.
deletion:every case= O(logn), because including searching
insertion:every case= O(logn) = O(h) ,h = height;because including searching. It use search to select the nil where the new node will be added.
minimum, maximum:every case= O(logn) = O(h) , h = height ,
successor and predecessor:every case= O(logn) = O(h) , h = height,
height: O(n);When calculating the height, it traverse all branches and select the longest one.O(n) for every case
all-nodes:O(n):When calculating to all nodes, it traverse all branches and select the longest one.O(n) for every case
left and right rotate:every case= O(logn)
The height of an RB tree is balanced, so there is no significant difference between the best and worst cases. Both are log(n) for most operations. Exceptions are mentioned above in specific scenarios.

1.3.2. BS Tree

searching:= O(h), average case O(logn) , h = height , worst case O(n) if height equal to n-1.It traverse only a single branch.
deletion:= O(h), average case O(logn) , h = height , worst case O(n) if height equal to n-1. because including searching
insertion:= O(h), average case O(logn) , h = height , worst case O(n) if height equal to n-1. Because including searching. It use search to select the nil where the new node will be added.

minimum, maximum:= O(h), average case O(logn) h = height , worst case O(n) if height equal to n-1.
successor and predecessor:= O(h), average case O(logn) , h = height, worst case O(n) if height equal to n-1.
height: O(n);When calculating the height, it traverse all branches and select the longest one.O(n) for every case
all-nodes: O(n):When calculating to all nodes, it traverse all branches and select the longest one.O(n) for every case.
Binary search tree's worst-case height occurs when the height is n1. the scenario where all nodes create a single branch.

1.4. Brief Implementation Details

Explain your implementation with also addressing the following guide questions;

1. I ensured its compliance with the color-related rules through the delete fix-up functions. I had to write some routines that called both insert and remove operations while still respecting color and height constraints.
   left rotate:It converts the structure on the right side of the diagram to the structure on the left side. Node x has a right child y that cannot be null. The left rotation shifts the subtree that was previously rooted at x to the left by changing the relationship between x and y. Node y is the new root of the subtree, with x equaling y's left child and y's original left child equaling x's right child. Right rotate is the symmetry ıf this functiion

2. The deletehelperNode function deletes a node by using the searchTree function. This solves the situation where the node does not have a subtree to its left or right. When the deleteNode function is called, it correctly adjusts the node's parent and connects the node that replaces the deleted node. This deletionNodeHelper is a utility function that deletes one node (x) and replaces it with another (y). If node x is the current root, then node y must be the new root. If node x is a parent's left child, it replaces the parent's left child with y; if node x is a parent's right child, it replaces the parent's right child with y. Finally, if node y is not NULL, it connects node y's parent to node x's parent.

3. getTotalNodesHelper: Calculate all nodes recursively starting the reference point

4. getTotalNodes: Calculate all nodes recursively, for all tree send root as a reference

5. getMinimumHelper: Search the leftmost child started reference node

6. getMinimum: Search the leftmost child, for calculate all tree ' s minumum send to root for reference

7. getMaximumHelper: Search the rightmost child started reference node

8. getMaximum: Search the rightmost child, for calculate all tree ’ s maximum send to root for reference
9. getHeight: Sends the root as a reference to calculate the total height of the tree
10. getHeightHelper: It traverses the entire tree, starting at the reference node
11. successor: for finding e node with the smallest data value greater than x.data
12. searchTree: It traverses the branches of the tree until it finds tinput value, determining its direction by comparing the input data with the current node s data at each step.
13. rightrotate: In this function, node y is the new root of the subtree, with x as its left child and y’s original left child as its right child. Left rotate is symmetry of this function.
14. postorderhelper: Walk first left child, second right child then parent
15. postorder: For walk all tree, call postorderfunction with root
16. inorderhelper: First left child, second parent then right child
17. inorder: For walk all tree, call inorderhelperfunction with root
18. preorderhelper: First parent, second left child then right child
19. preorder: For walk all tree, call preorderfunction with root