Data Structures and Algorithms 2 (ICS2210)

Nicholas Vella 0440803L

The difference between AVL trees, Red-Black trees, and unbalanced BSTs.

When it comes to balancing the tree, AVL Trees require more frequent rebalancing compared to Red-Black trees. While the unbalanced BST is the simplest form of a binary search tree, it is not self-balancing. For the height, AVL guarantees to maintain a height balance factor of at most 1, while Red-Black trees have a height balance factor of at most 2 and this makes it slightly slower than AVL trees for search-intensive applications.

Then for the worst-case time complexity, AVL trees have the worst-case time complexity of O(log n) for search, insertion, and deletion operations. While the Red-Black trees have a worst-case time complexity of O(log n) and the unbalanced BST has a worst-case time complexity of O(n) for search, insertion, and deletion operations.

Red-Black trees are more efficient in terms of memory usage compared to AVL trees, since AVL trees are more memory-intensive due to the need to store a balance factor at each node. Meanwhile, unbalanced BST is memory-efficient, but can suffer from poor performance in certain cases. The unbalanced BST is useful for small datasets or when the cost of rebalancing outweighs the benefits of a self-balancing tree.

AVL trees are generally faster than Red-Black trees for search-intensive applications, but Red-Black trees are generally preferred over AVL trees for general-purpose use due to the better balance between performance and memory usage. And, because they are easier to implement compared to AVL trees.

AVL Tree

To keep the tree balanced, the AVL tree keeps track of each node's balancing factor. Each node in the AVL tree in my code has a value, a left child, a right child, a height, and a comparison count.

The insert method can be used to update the AVL tree. In order to create a new node with the desired value, it first checks to see if the node is None. If the value is less than the value of the current node, insertHelper is called recursively on the left child, and on the right child if the value is greater. The comparison count is raised each time a comparison is performed to determine where the new value should be in the tree. If the new node's balance factor is more than 1 or less than -1, rotations are performed to rebalance the tree and update the new node's height.

A value can be removed from the AVL tree using the delete method. The node is initially verified as being None. It recursively executes deleteHelper on the left child if the value is smaller than the value of the current node. It recursively executes deleteHelper on the right child if the value is higher than the value of the current node. The smallest value in the right subtree is used to replace the value of the node if the node to be deleted has both a left and a right child. The duplicate value in the right subtree is then deleted. The node's height is updated, and rotations are carried out to rebalance the tree if the node's balance factor is greater than 1 or less than -1.

The AVL tree is rotated via the leftRotate and rightRotate methods to restore balancing. When a node's balance factor exceeds 1 and the balance factor of its left child is more than or equal to 0, a left rotation is carried out. When a node's balance factor is less than -1 and its right child's balance factor is similarly less than or equal to 0, a right rotation is carried out. Each technique modifies the height of the node and any additional child nodes.

By recursively traversing the tree and counting each node, the nodeCount method returns the total number of nodes in the AVL tree. The getComparisonCount method adds the comparison counts of all the nodes in the tree and provides the total number of comparisons performed during insertion. The AVL tree's total number of rotations during rebalancing is returned by the rotationCount method.

Red-Black Tree

There are various methods available in the RedBlackTree class, including insert() (add a node with a given value to the tree), find() (look for a node with a given value in the tree), and delete() (remove a node with a given value from the tree), which, in turn.

The RedBlackTree class also has a number of utility functions that carry out various tasks necessary to preserve the attributes of a red-black tree, including rotateLeft(), rotateRight(), and is_red().

The class Node, which has the characteristics val, left, right, height, color, and comparisonCount, is the parent class of each node in the tree. The left and right attributes point to the left and right child nodes, respectively, while the val attribute records the value of the node. The height, color, and comparisonCount attributes all retain information about the node's height, color (which can be "RED" or "BLACK"), and the number of comparisons that were performed when the node was added to or removed.

Other characteristics of the RedBlackTree class include root, which corresponds to the tree's root node, and left_rotations and right_rotations, which count the left and right rotations made during insertion or deletion.

To make sure the tree is balanced and effective, the RedBlackTree class makes use of the following three properties:

1. Every node is either red or black.
2. The root node is black.
3. If a node is red, then both its children are black.

By carrying out different operations on the nodes upon insertion and deletion, such as rotating the tree and altering node colors, the class implements these attributes. Additionally, the RedBlackTree class records the number of comparisons performed during insertion or deletion, which can be used to evaluate the algorithm's effectiveness.

Binary Search Tree

Each node in the system has a maximum of two offspring, the left child and the right child, in a tree-like structure.

The structure of each node in the tree is specified by the class's nested class Node. Each node contains references to its left and right children as well as a value. The root node of an empty tree is initialized to None via the __init__ method.

By invoking the helper function insertHelper, the insert method adds a new node with the specified value to the tree. In order to determine the proper location for the new node depending on its value, it starts at the root node and moves recursively through the tree. If the value is lower than the value of the existing node, the new node is added to the left subtree. It is put in the appropriate subtree if not.

By using a helper function called deleteHelper, the delete method eliminates a node with a specified value from the tree. To locate the node with the specified value, it searches the tree recursively from the root node up. If the node is located, it is substituted with the minimum or maximum value of its right or left subtrees to remove it from the tree.

The maximum number of edges between the root node and a leaf node is the height of the tree, which is what the height method returns. The total number of nodes in the tree is returned by the nodeCount method. The number of comparisons performed overall during insert and remove operations are returned by the comparisonCount method.

Statement of completion – MUST be included in your report

Item	**Completed (Yes/No/Partial)**
Created sets X, Y, and Z without duplicates and showing intersections.	YES
AVL tree insert	YES
AVL tree delete	YES
AVL tree search	YES
RB tree insert	YES
RB tree delete	YES
RB tree search	YES
Unbalanced BST insert	YES
Unbalanced BST delete	YES
Unbalanced BST search	YES
Discussion comparing tree data structures	YES
If partial, explain what has been done	

Plagiarism Declaration Form

Plagiarism is defined as "*the unacknowledged use, as one's own, of work of another person, whether or not such work has been published, and as may be further elaborated in Faculty or University guidelines*" (University Assessment Regulations, 2009, Regulation 39 (b)(i), University of Malta).

I, the undersigned, declare that the report submitted is my work, except where acknowledged and referenced. I understand that the penalties for committing a breach of the regulations include loss of marks; cancellation of examination results; enforced suspension of studies; or expulsion from the degree programme.

Work submitted without this signed declaration will not be corrected, and will be given zero marks.

Nicholas Vella	ICS2210	23/04/2023
Student's full name	**Study-unit code**	**Date of submission**

Title of submitted work: Data Structures and Algorithm course work

Student's Signature