COURSE PROJECT 2023

Data Structures and Algorithms 2

Kelsey Bonnici
17203L

Table of Contents

Implementation

The code for this project can be found split up in different Python files, with the Main.py file connecting them all.

Sets

In Sets.py, the code that declares all the sets and their intersections required can be found.

Tree Information

In each of the tree files, there is a dictionary treeInfo declared which is used to hold information about the trees to be displayed to the user. The dictionary holds the following:

- Height
- Number of nodes
- Number of comparisons
- Numbers found in the search
- Number of rotations (only in AVL and RBT)

The height is calculated using the getHeight function found in each tree. The code of this function differs according to the implementation of the tree.

The number of nodes is incremented each time a new node is created and decremented when one is deleted.

The number of comparisons is incremented during tree traversal whenever there is a decision of whether to go to the left or right child.

The numbers found during the search is incremented every time a value is found in the search function.

The number of rotations is incremented every time either a left or a right rotation is done.

Binary Search Tree

The implementation for the binary search tree can be found in BST.py. The implementation was done following explanations found in [1].

A dictionary called treeInfo can be found declared in this file which is used to hold information about the tree such as the height, the number of nodes, comparisons done, and the numbers found during the search.

The insertion is done by recursively traversing the tree until a node which has not been set is found. Anytime a new node is created, the number of nodes found in the treeInfo dictionary is incremented. The traversal is done by comparing the value that is to be inserted with the current's node value and deciding whether to go left or right as per the rules of a BST. Each time a comparison is done, the comparisons value in the treeInfo

dictionary is incremented. At the end of the insertion the node's height is calculated using the function getHeight.

The delete function starts off in a similar manner as the insertion by recursively traversing the tree until the node is found. Note that this function also handles values which are not found in the tree.

When the node is found there are three different cases which can be encountered:

1. Node only has a child on the right.
2. Node only has a child on the left.
3. Node has children on both sides.

In case 1, the node on the right is stored in the variable called nodeX and the node is set to none. The same is done for case 2 but with the left child. In case 3, nodeX is set as the node with the minimum value from the right subtree of the node. This is found using the function minimumNode. The node is then replaced with nodeX and the node that was moved up is then deleted.

At the end of the function the height of the node is updated. Similar to the insertion, the comparisons done and the number of nodes is adjusted accordingly in the dictionary treeInfo.

The search is done by recursively traversing the tree in a similar manner as was done in the insertion and deletion. Note that the function also handles values which are not in the tree. Any time a number is found, the numbersFound value is incremented. The comparisons value is also incremented every time a comparison is done.

AVL Tree

The implementation for the AVL tree can be found in AVL.py. The implementation was done following explanations found in [2].

The structure of this implementation follows that of the BST but with added rotations to follow the rules of an AVL tree.

After a node is inserted into the tree using the same procedure as that of the BST, the balance factor of the tree is calculated. If the balance factor is 0 or 1 that means that the tree is balanced. A balance factor bigger than 1 means that the height of the left subtree is greater than that of the right subtree. Similarly, a height smaller than -1 means that the height of the right subtree is greater than that of the left. Rotations are done as required and the height for the node is calculated using the getHeight function.

The deletion also follows the deletion of the BST with additional checks for the balance factor. The check for the balance factor works in a similar manner to the insertion, but it checks the balance factors of the subtrees instead of just the values of the children.

The search function is identical to the one done for the BST since search always stays the same on any type of tree.

Red-Black Tree

The implementation for RBTs can be found in RBT.py. The implementation was done following pseudocode found in [3].

The structure of the code differs from that of the BST and the AVL as will also be discussed later on when discussing the Main function.

At the start of the insertion, the node is created, and its children are set to NIL. The NIL nodes are required to make sure that none of the RBT rules are being violated. The location of the node in the tree is found iteratively instead of recursively. After the node is inserted, the code checks if any violations of the RBT properties have occurred.

There are three cases in which an RBT violation can happen:

1. The parent and uncle nodes are both red.
2. The parent is red, the uncle is black, and the new node is a right child.
3. The parent is red, the uncle is black, and the new node is a left child.

In case 1, the parent and uncle are set to black, and the grandparent node to red. The grandparent node is then set to the current node and further checks for violations are done. In case 2, the current node's parent is rotated to the left. In case 3 the parent is set to black, the grandparent to red and the grandparent node is rotated to the right. The while loop continues until the root node is reached or the current node's parent is black. Finally, the code turns the root to black to ensure that it is always like that.

The delete starts off by search iteratively for the node that is to be deleted. Note that this method also handles values which are not found in the tree. Similar to the BST and AVL tree, there are the three different cases when a node is found that need to be handled:

1. Node only has a child on the right.
2. Node only has a child on the left.
3. Node has children on both sides.

These cases are handled in a similar manner to that of the BST/AVL tree.

After the node is deleted, the check for any RBT rules violations starts. There are four cases in which a violation occurs:

1. The sibling is red.
2. The sibling is black, and both its children are black.
3. The sibling is black, its left child is red and its right child is black
4. The sibling is black, its left child is black, and its right child is red

Note that these are the cases for when nodeY (which represents the node that replaces the delete note) is a left child. The cases for when nodeY is a right child are symmetric. These violations are handled accordingly.

The search method has the same logic as when traversing through the tree in both the insert and delete methods. It searches through the tree iteratively. Note that the function also handles values which are not in the tree.

Main

In Main.py, all the trees are declared, and each required function is executed.

The code starts off by declaring all the sets using the createSets function. The trees (and nodes required for their function) are then declared.

The first for loop inserts all the values in set X into each tree. The height of each tree is calculated. The results are then printed to the user.

Next, set Y is deleted from each tree. Before doing so, the comparison and rotation values are reset to 0. After the deletion is done, the height is recalculated, and the results are displayed to the user.

Finally, the search is done. The comparisons are again set to 0 and each value in set Z is searched for in the tree. The results are then displayed to the user.

Results

Running the main function produces the following results:

```
Set X contains 2309 integers.
Set Y contains 541 integers.
Set Z contains 852 integers.

Sets X and Y have 185 values in common.
Sets X and Z have 345 values in common.

Inserting set X
BST: height is 25, #nodes is 2309, #comparisons is 29825.
AVL: 1590 rotations req., height is 13, #nodes is 2309, #comparisons is 23011.
RBT: 1321 rotations req., height is 14, #nodes is 2309, #comparisons is 23117.

Deleting set Y
BST: height is 24, #nodes is 2124, #comparisons is 7664
AVL: 57 rotations req., height is 13, #nodes is 2124, #comparisons is 5864
RBT: 57 rotations req., height is 14, #nodes is 2124, #comparisons is 5810

Searching set Z
BST: 11156 total comparisons required, 313 numbers found, 539 numbers not found
AVL: 8430 total comparisons required, 313 numbers found, 539 numbers not found
RBT: 8978 total comparisons required, 313 numbers found, 539 numbers not found
```

The number of nodes in each tree is the same as the length of set X which shows that each number was inserted successfully.

After deleting, the number of nodes drops to 2124. Subtracting 185 (length of X and Y intersection) from 2309 results in 2124 which shows that each node was deleted successfully.

When searching, 313 numbers are stated to be found. This does not match the length of X and Z intersection due to nodes being deleted. To prove that each node is found as required, the code was run again using the same sets but without executing the delete functions.

```
Set X contains 2309 integers.
Set Y contains 541 integers.
Set Z contains 852 integers.

Sets X and Y have 185 values in common.
Sets X and Z have 345 values in common.

Inserting set X
BST: height is 25, #nodes is 2309, #comparisons is 29825.
AVL: 1590 rotations req., height is 13, #nodes is 2309, #comparisons is 23011.
RBT: 1321 rotations req., height is 14, #nodes is 2309, #comparisons is 23117.

Searching set Z
BST: 11337 total comparisons required, 345 numbers found, 507 numbers not found
AVL: 8527 total comparisons required, 345 numbers found, 507 numbers not found
RBT: 9052 total comparisons required, 345 numbers found, 507 numbers not found
```

As can be seen from above, 345 numbers were found which is the same as the length of X and Z intersection. This proves that each value was found successfully.

Discussion

The following table demonstrates the worst-case time complexity. Data retrieved from [3] and [4]:

	BST[1]	AVL	RBT
Insert	$O(h)$	$O(\log n)$	$O(\log n)$
Delete	$O(h)$	$O(\log n)$	$O(\log n)$
Search	$O(h)$	$O(\log n)$	$O(\log n)$

As can be seen in this table, the BST has the worst time complexity out of all the three types of trees. This is due to it being unbalanced. When referring back to the results obtained, it is observed that the BST has the most comparisons and a much larger height.

When comparing the AVL tree and RBT, the heights are not that much different from each other. The main difference is in the number of comparisons and rotations required. The AVL requires more rotations but less comparisons and on the other hand, the RBT requires less

[1] Where h is the height of the tree

rotations but more comparisons. Both are efficient data structures. The choice of one over the other depends on their usage.

If the application involves more frequent insertions and deletion, then an RBT tree is more recommended since it performs less rotation. If the application is more search based, then an AVL tree is recommended [5] since it performs less comparisons.

In conclusion, out of the three different types, the BST clearly has the worst performance. A best cannot be chosen from the AVL tree and the RBT as their efficiency heavily depends on where the tree is being used.

References

[1] GeeksforGeeks, “Binary Search Tree,” 21 March 2023. [Online]. Available: https://www.geeksforgeeks.org/binary-search-tree-data-structure/.

[2] GeeksforGeeks, “AVL Tree Data Structure,” 22 March 2023. [Online]. Available: https://www.geeksforgeeks.org/introduction-to-avl-tree/.

[3] T. H. Cormen, C. E. Leiserson, R. L. Rivest and C. Stein, Introduction to Algorithms, Fourth Edition, MIT Press, 2022.

[4] E. Alexander, “AVL Trees,” 2011. [Online]. Available: https://pages.cs.wisc.edu/~ealexand/cs367/NOTES/AVL-Trees/index.html.

[5] GeeksforGeeks, “Introduction to Red-Black Tree,” GeeksforGeeks, 15 March 2023. [Online]. Available: https://www.geeksforgeeks.org/introduction-to-red-black-tree/.

Statement of Completion

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

Plagiarism Declaration Form

FACULTY OF INFORMATION AND COMMUNICATION TECHNOLOGY

Declaration

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 assignment 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.

Kelsey Bonnici _____ _____

Student Name Signature

ICS2210 _____ Course Project 2023 _____

Course Code Title of work submitted

09/04/2023 ___

Date