L-Università ta' Malta

Data Structures and Algorithms 2

ICS2210

Programmed using Python

By Mark Dingli
20703H

Table of contents

Statement of Completion

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

Student's Signature

Introduction

AVL trees, Red-Black trees, and unbalanced Binary Search Trees (BSTs) are popular data structures used to implement efficient search and retrieval operations. While each of these structures serves the same fundamental purpose, they have different properties that make them more suitable for specific real-world applications. In this report, we will discuss the strengths and weaknesses of AVL trees, Red-Black trees, and unbalanced BSTs and analyze their suitability for real-world applications. Specifically, we will consider factors such as search and retrieval performance, insertion and deletion performance, and the ease of implementation. However, it's important to note that the performance of each type of tree can vary depending on the specific use case and the distribution of the elements in the tree. [1][2]

Creating and Outputting sets X,Y,Z

Description

The *random library* [3] was used to generate three sets of random integers, X, Y, and Z. It was also used to check for duplicates in each set. The size of each set is randomly determined between certain ranges. Set X contains a random number of integers between 1000 and 3000. Set Y contains a random number of integers between 500 and 1000. Set Z contains a random number of integers also between 500 and 1000. For each set, each integer is a unique random number in the range -3000 and 3000.

Output

The code outputs the following:

- The size of the set X (number of unique random integers).
- The size of the set Y (number of unique random integers).
- The size of the set Z (number of unique random integers).
- An error message if duplicates are found in any of the sets.

Example of output for set X:

```
Set X contains 2015 integers.

------------------------------------------------------------------------------------------------------------------------
```

Figure 1 - Output showing the size of the set X

Testing

The code checks for duplicates in three sets by comparing the length of the set to the length of a list of the same set. If the lengths are not equal, it indicates that there are duplicates in the set, and the program prints an error message saying "Error: Set __ contains duplicates." On the other hand, if the lengths are equal, the program prints the size of the set. This testing can be seen in the code snippet below:

```
# Check for duplicates in set Z
if len(Z) != len(set(list(Z))):
    print("\nError: Set X contains duplicates.")
else:
    # Print the size of set Z
    print(f"\nSet Z contains {len(Z)} integers.")
```

Limitations

The random.sample() function can be computationally expensive for very large values of 'p', 'q', or 'r', which may affect the performance of the code.

Determining the Intersection

Description

This code finds the common elements between sets X and Y, and between sets X and Z. It also prints the size of each intersection.

Output

The code outputs the following:

- The size of the intersection between sets X and Y (number of common elements).
- The size of the intersection between sets X and Z (number of common elements).

Example of output for intersection of set X and Y:

```
Sets X and Y have 260 values in common.

--------------------------------------------------------------------------------------------------------------
```

Figure 2 - Output showing the intersection of X and Y

Example of output for intersection of set X and Z:

```
Sets X and Z have 269 values in common.

--------------------------------------------------------------------------------------------------------------
```

Figure 3 - Output showing the intersection of X and Z

AVL Tree

An AVL tree is a self-balancing binary search tree where the heights of the left and right subtrees of any node differ by at most one. each node is associated with a balance factor which is calculated by subtracting the height of its right sub-tree from that of its left sub-tree. An AVL Tree is said to be balanced if balance factor of each node is in between -1 to 1, otherwise, the tree will be unbalanced and needs to be balanced. This balancing property ensures that the worst-case time complexity of searching, inserting, and deleting nodes in the tree is O(log n), where n is the number of nodes in the tree. [4]

Description of code

- Code inspired by [5][6]

- ***AVLNode:*** This class represents a node in the AVL tree. It has four attributes: val represents the value stored in the node, left and right represent pointers to the left and right child nodes, respectively, and height represents the height of the node.

- ***__init__(self, val):*** This is the constructor for the *AVLNode* class. It initializes a new node with the given value, left and right child nodes set to None, and a height of 1.

- ***AVLTree:*** This class represents the AVL tree. It has a root node, number of rotations, number of comparisons, and number of nodes in the tree. It provides various methods for inserting, deleting, and searching nodes in the tree.

- ***__init__(self):*** This is the constructor for the AVLTree class. It initializes an empty AVL tree with a None root node, 0 rotations, 0 comparisons, and 0 nodes.

- ***insert(self, val):*** This method inserts a new node with the given value into the AVL tree and updates the statistics of the tree, such as the number of rotations and comparisons performed during the insertion operation.

- ***_insert_helper(self, val, node):*** This is a recursive helper method used by *insert()* to insert a new node with the given value into the AVL tree rooted at the given node.

- ***_get_height(self, node):*** This method returns the height of the given node in the tree or 0 if the node is None.

- ***_get_balance(self, node):*** This method returns the balance factor of the given node or 0 if the node is None.

- ***_rotate_left(self, node):*** This method performs a left rotation at the given node and returns the new root node.

- ***_rotate_right(self, node):*** This method performs a right rotation at the given node and returns the new root node.

- ***delete(self, val):*** This method deletes the node with the given value from the AVL tree and updates the statistics of the tree, such as the number of rotations and comparisons performed during the deletion operation.

- ***_delete_helper(self, val, node):*** This is a recursive helper method used by *delete()* to delete the node with the given value from the AVL tree rooted at the given node and returns the new root node, the number of rotations, and the number of comparisons.

- ***_find_min(self, node):*** This method finds the node with the minimum value in the subtree rooted at the given node and returns it.

- ***search_avl_tree(self, val, node):*** This method searches for the node with the given value in the AVL tree rooted at the given node and returns a tuple with a Boolean indicating whether the node was found and the number of comparisons performed.

- ***_search_avl_tree_helper(self, val, node):*** This is a recursive helper method used by *search_avl_tree()* to search for the node with the given value in the AVL tree rooted at the given node.

- ***print_tree(self):*** This method prints the values of the nodes in the AVL tree in ascending order.

- ***_print_tree_helper(self, node):*** This is a recursive helper method used by print_tree() to recursively print the values of the nodes in the AVL tree in ascending order.

- ***get_stats(self):*** This method returns a string with the statistics for the AVL tree, such as the total number of rotations required, the height of the tree, the number of nodes, and the number of comparisons performed during insertions and deletions.

Input

The input for the AVL tree is a set of all the unique integers, which are generated using the *random.sample()* function and stored in the set X. These integers are then inserted into the AVL tree using the *insert()* method of the *AVLTree* class.

Output & Testing

```
# Create an empty AVL tree
avl_tree = AVLTree()

# Insert all elements in X into the AVL tree
for x in X:
    avl_tree.insert(x)
```

This code creates an empty AVL tree object called "*avl_tree"* using the *AVLTree()* constructor. The code will then loop through each element of *set X* and insert it into the AVL tree using the *insert()* method of the AVLTree object. The *insert()* method ensures that the tree remains balanced by performing rotations whenever necessary.

```
# Print the elements of the AVL tree in ascending order
# avl_tree.print_tree()
```

To check that all the elements of *set X* are correctly inserted into the AVL Tree, this code prints the elements of the AVL tree in ascending order using the *print_tree()* method of the AVL tree object. The *print_tree()* method performs an in-order traversal of the tree, visiting each node in ascending order, and printing its value. To see the output of all the elements in the tree in ascending order, just remove the *#* comment from the last line.

This output can also be compared to the output of the other trees to make sure that all trees have inserted all the elements of the *set X* correctly.

Snippet output of the elements of the AVL tree in ascending order:

```
-2998
-2989
-2988
-2985
-2984
-2982
-2981
-2980
-2979
-2978
-2977
-2975
-2967
-2963
-2962
-2960
-2959
```

Figure 4 - Output showing elements of the AVL tree in ascending order

The AVL Tree implementation does not support insertion of duplicate values as well as deletion of nodes with duplicate values. If a value is inserted into the AVL Tree that already exists in the tree, it will not be inserted again and the number of nodes in the tree will not increase. This is not a problem since the code that inserts the elements into *set X* ensures that there are no duplicates as previously mentioned.

Limitations

- The AVL Tree implementation uses recursion to insert, delete, and search nodes. This means that very large trees may cause stack overflow errors.

- This AVL Tree implementation is only suitable for use with numerical values, as it compares values using the less than and greater than operators.

Red-Black Tree

A Red-Black Tree is a type of self-balancing binary search tree where each node is coloured either red or black, and maintains two key properties:

1. The root node is black.
2. For any node in the tree, all paths from that node to its leaf descendants contain the same number of black nodes.

These two properties ensure that the height of the tree is approximately balanced, which in turn guarantees that the time complexity of search, insert, and delete operations is O(log n) in the worst case, where n is the number of nodes in the tree. To maintain the balance of the tree, red-black trees use a set of rules for inserting, deleting, and rotating nodes that preserve the two properties mentioned above. As a result, red-black trees are efficient, reliable, and widely used in various applications where balanced trees are required. [7]

Description of code

- Code inspired by [8][12]

- ***Node class:*** This class represents a node in the RB tree. It has five attributes: val represents the value stored in the node, left and right represent pointers to the left and right child nodes, respectively, height represents the height of the node and a colour attribute that can be "RED" or "BLACK".

- ***__init__(self, val):*** This is the constructor for the *Node* class. It initializes a new node with the given value, left and right child nodes set to None, a height of 1 and colour to “RED”.

- ***RedBlackTree:*** This class represents the RB tree. It has a root node, colour of node, number of rotations, number of comparisons, and number of nodes in the tree. It provides various methods for inserting, deleting, and searching nodes in the tree.

- ***__init__(self):*** This constructor initializes the Red-Black Tree with a root node set to None, a nil node with value None and colour "BLACK", and counters for the number of rotations, comparisons, and nodes in the tree.

- ***insert(self, val):*** Inserts a new node with the given value into the Red-Black Tree and then fixes any violations to the Red-Black Tree properties.

- ***_fix_violations(self, val):*** Fixes any Red-Black Tree violations caused by inserting a node, by rotating nodes as necessary and recolouring them.

- ***_left_rotate(self, val):*** Rotates the given node to the left in the Red-Black Tree.

- ***_right_rotate(self, val):*** Rotates the given node to the right in the Red-Black Tree.

- ***search(self, val):*** Searches for a node with the given value in the Red-Black Tree and returns it if found; otherwise, returns None.

- ***delete(self, val):*** Deletes the node with the given value from the Red-Black Tree, if it exists, and then fixes any "double black" violations that may arise.

- ***_fix_double_black(self, val):*** Fixes the "double black" violation that may occur after a node is deleted from the Red-Black Tree, by rotating nodes and recolouring them.

- ***_get_successor(self, val):*** Returns the node that would come after the given node in a sorted list of the Red-Black Tree's values.

- ***_get_height_helper(self, val):*** A recursive helper function that calculates the height of the Red-Black Tree by finding the maximum height of its left and right subtrees.

- ***get_height(self):*** Returns the height of the Red-Black Tree by calling the *_get_height_helper* function on the root node.

- ***_print_tree_helper(self, val):*** A recursive helper function that prints the values in the Red-Black Tree in ascending order by traversing the left subtree, printing the current node's value, and traversing the right subtree.

- ***print_tree(self):*** Prints the values in the Red-Black Tree in ascending order by calling the *_print_tree_helper* function on the root node.

- ***__str__(self):*** Returns a string representation of the Red-Black Tree, including information about the number of rotations, the height, the number of nodes, and the number of comparisons performed during searches.

Input

The input for the RB Tree is a set of all the unique integers, which are generated using the *random.sample()* function and stored in the set X. These integers are then inserted into the RB Tree using the *insert()* method of the *RedBlackTree* class.

Output & Testing

```
# Create an empty Red Black tree
rb_tree = RedBlackTree()

# Insert all elements in X into the Red-Black tree
for x in X:
    rb_tree.insert(x)
```

This code creates an empty RB Tree object called "rb_*tree"* using the RedBlackTree*()* constructor. The code will then loop through each element of *set X* and insert it into the RB Tree using the *insert()* method of the *RedBlackTree* object. The *insert()* method ensures that the tree remains balanced by performing rotations and recolouring whenever necessary.

```
# Print the elements of the Red-Black tree in ascending order
# rb_tree.print_tree()
```

To check that all the elements of *set X* are correctly inserted into the *RB Tree*, this code prints the elements of the *RB Tree* in ascending order using the *print_tree()* method of the *RedBlackTree* object. The *print_tree()* method performs an in-order traversal of the tree, visiting each node in ascending order, and printing its value. To see the output of all the elements in the tree in ascending order, just remove the *#* comment from the last line.

This output can also be compared to the output of the other trees to make sure that all trees have inserted all the elements of the *set X* correctly.

Snippet output of the elements of the Red Black Tree in ascending order:

```
-2998
-2989
-2988
-2985
-2984
-2982
-2981
-2980
-2979
-2978
-2977
-2975
-2967
-2963
-2962
-2960
-2959
```

Figure 5 - Output showing elements of the Red Black Tree in ascending order

It is important to note that when outputting all the elements of the AVL Tree, Red-Black Tree and the Unbalances BST, the output is the same. This confirms that all the elements of the *Set X* where correctly inserted into all of the trees.

Similar to the AVL Tree, the Red-Black Tree implementation does not support insertion of duplicate values as well as deletion of nodes with duplicate values. If a value is inserted into the Red-Black Tree that already exists in the tree, it will not be inserted again and the number of nodes in the tree will not increase. This is not a problem since the code that inserts the elements into *set X* ensures that there are no duplicates as previously mentioned.

Limitations

- The code does not have any balancing strategies for cases where the tree becomes heavily unbalanced. This can lead to degraded performance in the worst-case scenarios.

Binary Search Tree (Unbalanced)

An unbalanced binary search tree is a type of binary search tree where the height of the tree is not necessarily minimized. In other words, the tree may have some branches that are significantly longer than others. In an unbalanced binary search tree, the worst-case time complexity of various operations such as searching, inserting, and deleting nodes can become O(n), where n is the number of nodes in the tree. This is because the unbalanced tree can resemble a linked list in the worst case, where all nodes are connected in a single path. To improve the performance of operations on binary search trees, it is often desirable to keep the tree balanced. [9]

Description of code

- Code inspired by [10][11]

  - ***BSTNode:*** This class represents a node in the BST. It has three attributes: val represents the value stored in the node and left and right represent pointers to the left and right child nodes, respectively.

  - ***__init__(self, val):*** This is the constructor for the *BSTNode* class. It initializes a new node with the given value, left and right child nodes set to None, and a height of 1.

  - ***BST:*** This class represents the BST. It has a root node and number of comparisons. It provides various methods for inserting, deleting, and searching nodes in the tree.

  - ***__init__(self):*** This constructor initializes a new empty binary search tree with root set to None and comparisons set to 0.

  - ***insert(self, val):*** This method inserts a new node with value *val* into the BST. If the BST is empty (root is None), then the new node is created as the root. Otherwise, the *_insert_helper* recursive function is called to traverse the BST and insert the new node in the correct position.

  - ***_insert_helper(self, val, node):*** Recursive helper function to insert a new node with value *val* into the BST. The function compares the *val* with the current node's value *"node.val"* and recursively calls itself on the left subtree if *val* is less than "*node.val"*, or on the right subtree if *val* is greater than or equal to "*node.val".* If the left or right child of the current node is None, then the new node is created as that child.

  - ***height(self, node):*** Recursive function to calculate the height of the BST rooted at node. The function returns 0 if node is None, and otherwise returns 1 plus the maximum of the heights of the left and right subtrees.

  - ***num_nodes(self, node):*** Recursive function to calculate the number of nodes in the BST rooted at node. The function returns 0 if node is None, and otherwise returns 1 plus the sum of the number of nodes in the left and right subtrees.

- ***delete(self, val):*** Deletes the node with value *val* from the BST. The function calls the *_delete_helper* recursive function to traverse the BST and remove the node with *val*.

- ***_delete_helper(self, val, node):*** Recursive helper function to delete the node with *val* from the BST rooted at node. The function compares the *val* with the current node's value *"node.val"* and recursively calls itself on the left subtree if *val* is less than *"node.val"*, or on the right subtree if *val* is greater than *"node.val"*. If the node to be deleted has no children, then it is simply removed from the BST. If it has one child, then that child is moved up to take its place. If it has two children, then the minimum node in the right subtree is found and its value is used to replace the value of the node to be deleted. The *_delete_helper* function is then called recursively on the right subtree to remove the minimum node.

- ***_find_min(self, node):*** Recursive function to find the minimum node in the *BST* rooted at node. The function returns the node with the smallest value by traversing the left subtree until it reaches a leaf node.

- ***search_bst(self, val, node):*** Searches for a node with *val* in the BST rooted at node. The function returns True and increments the comparisons counter if it finds a node with *val*. If *val* is less than *"node.val",* the function recursively searches the left subtree; if *val* is greater than *"node.val"*, the function.

- ***print_tree(self):*** Function that prints the values in the *BST* in ascending order. The function calls the *_print_tree_helper* recursive function to traverse the *BST* in order and print the values.

- ***_print_tree_helper(self, node):*** Recursive helper function to print the values in the *BST* in ascending order. The function traverses the left subtree, prints the current node's value, and then traverses the right subtree.

Input

The input for the Unbalanced BST is a set of all the unique integers, which are generated using the *random.sample()* function and stored in the set X. These integers are then inserted into the BST using the *insert()* method of the *BST* class.

Output & Testing

```
# Create an empty BST tree
bs_tree = BST()

# Insert all elements in X into the BST
for x in X:
    bs_tree.insert(x)
```

This code creates an empty BST object called "bs_*tree"* using the BST*()* constructor. The code will then loop through each element of *set X* and insert it into the AVL tree using the *insert()* method of the AVL tree object. The *insert()* method ensures that the tree remains balanced by performing rotations whenever necessary.

```
# Print the elements of the BST in ascending order
# bs_tree.print_tree()
```

To check that all the elements of *set X* are correctly inserted into the *BST*, this code prints the elements of the *BST* in ascending order using the *print_tree()* method of the BST object. The *print_tree()* method performs an in-order traversal of the tree, visiting each node in ascending order, and printing its value. To see the output of all the elements in the tree in ascending order, just remove the *#* comment from the last line.

This output can also be compared to the output of the other trees to make sure that all trees have inserted all the elements of the *set X* correctly.

Snippet output of the elements of the Unbalanced BST in ascending order:

```
-2998
-2989
-2988
-2985
-2984
-2982
-2981
-2980
-2979
-2978
-2977
-2975
-2967
-2963
-2962
-2960
-2959
```

Figure 6 - Output showing elements of the Unbalanced BST in ascending order

It is important to note that when outputting all the elements of the AVL Tree, Red-Black Tree and the Unbalances BST, the output is the same. This confirms that all the elements of the *Set X* where correctly inserted into all of the trees.

Similar to the AVL Tree and the Red-Black Tree, the unbalanced BST implementation does not support insertion of duplicate values as well as deletion of nodes with duplicate values. If a value is inserted into the BST that already exists in the tree, it will not be inserted again and the number of nodes in the tree will not increase. This is not a problem since the code that inserts the elements into *set X* ensures that there are no duplicates as previously mentioned.

Limitations

- The BST implementation does not include any rebalancing algorithms, which means that the tree may become unbalanced over time. This can cause performance issues for certain operations, such as search or insertion, which can become slower as the tree becomes unbalanced.
- The height of the BST is limited by the recursion stack depth of the machine running the program. If the BST is too tall, the program may run out of stack memory, which can cause the program to crash.

Inserting Into Trees Statistics

After inserting all the values into each tree, the number of rotations performed in total (in the AVL and RB tree, not in the BST), the height of the tree, the number of nodes in the tree, and the number of comparison operations (left/right decisions) made in total are outputted.

Outputs

Outputs of the statistics of all three trees:

```
AVL: 1631 tot. rotations req., height is 11, #nodes is 1642, #comparisons is 14597

--------------------------------------------------------------------------------

RBT: 5671 tot. rotations req., height is 18, #nodes is 1642, #comparisons is 25214.

--------------------------------------------------------------------------------

BST: height is 833, #nodes is 1642, #comparisons is 674173.

--------------------------------------------------------------------------------
```

Figure 7 – Output 1 showing statistics of the AVL, BR, and Unbalanced BS Trees

```
AVL: 2375 tot. rotations req., height is 12, #nodes is 2388, #comparisons is 22527

--------------------------------------------------------------------------------

RBT: 8263 tot. rotations req., height is 19, #nodes is 2388, #comparisons is 38857.

--------------------------------------------------------------------------------

BST: height is 1195, #nodes is 2388, #comparisons is 1425636.

--------------------------------------------------------------------------------
```

Figure 8 – Output 2 showing statistics of the AVL, BR, and Unbalanced BS Trees

Result Analysis (Insert)

Figure 7 (Output 1)

- The *AVL tree* required a total of 1611 rotations to maintain balance and has a height of 11. It has 1622 nodes and 14368 comparison operations were performed during tree operations.

- The *Red Black Tree* required a total of 5561 rotations to maintain balance and has a height of 18. It also has 1622 nodes but required more comparison operations than the AVL tree, with 24825 comparisons.

- The *Unbalanced Binary Search Tree* has a height of 839, the highest among the three trees, and also has 1622 nodes. However, it required significantly more comparison operations than the other two trees, with 658477 comparisons.

- Overall, the AVL tree and Red Black tree are balanced trees that require less comparison operations compared to the Binary Search Tree. This makes them more efficient for insertion. Among the two, the AVL tree seems to be slightly more efficient with fewer rotations and comparison operations than the Red Black tree. The Binary Search Tree, on the other hand, is unbalanced and requires more comparison operations, making it less efficient for these operations.

Figure 8 (Output 2)

- The updated output provides statistics similar to the previous outputs but for trees with even more nodes than before.

- The AVL tree required a total of 2375 rotations to maintain balance and has a height of 12. It has 2388 nodes and 22527 comparison operations were performed during tree operations.

- The Red Black tree required a total of 8263 rotations to maintain balance and has a height of 19. It also has 2388 nodes but required more comparison operations than the AVL tree, with 38857 comparisons.

- The Binary Search Tree has a height of 1195, which is higher than the previous outputs, and also has 2388 nodes. However, it required significantly more comparison operations than the other two trees, with 1425636 comparisons.

- Similar to the previous outputs, the AVL tree and Red Black tree are balanced trees that require fewer comparison operations compared to the Binary Search Tree. The AVL tree still seems to be slightly more efficient than the Red Black tree in terms of rotations and comparison operations. The Binary Search Tree still has the highest height and requires significantly more comparison operations, making it less efficient for searching and insertion operations. Overall, the results reaffirm that balanced trees like AVL and Red Black trees can provide better performance for the insertion operation compared to unbalanced Binary Search Trees, especially when dealing with a larger number of nodes.

In conclusion, the result analysis provides insights into the performance of three different types of binary search trees: AVL tree, Red Black tree, and Binary Search Tree (BST). In both Figures 7 and 8, the AVL and Red Black trees are balanced trees that require fewer comparison operations compared to the unbalanced Binary Search Tree. **The AVL tree consistently outperformed the Red Black tree in terms of rotations and comparison operations, making it a slightly more efficient option for insertion.** The Binary Search Tree consistently had the highest height and required significantly more comparison operations than the balanced trees, making it less efficient for this. Overall, the results suggest that balanced trees like AVL and Red Black trees can provide better performance for insertion operations compared to unbalanced Binary Search Trees.

Deleting in Trees Statistics

After deleting all the all the elements in the *set Y* from each of the three trees, the number of rotations performed in total (in the AVL and RB tree, not in the BST), the height of the tree, the number of nodes in the tree, and the number of comparison operations (left/right decisions) made in total were outputted.

Outputs

Output of the statistics of all three trees:

```
After deleting set Y:
AVL: 1635 tot. rotations req., height is 11, #nodes is 866, #comparisons is 22501
--------------------------------------------------------------------------------
After deleting set Y:
RBT: 5729 tot. rotations req., height is 13, #nodes is 1434, #comparisons is 33585.
--------------------------------------------------------------------------------
After deleting set Y:
BST: height is 729, #nodes is 1434, #comparisons is 980209.
--------------------------------------------------------------------------------
```

Figure 9 – Output 1 showing statistics of the AVL, BR, and Unbalanced BS Trees

```
After deleting set Y:
AVL: 2379 tot. rotations req., height is 12, #nodes is 1729, #comparisons is 29594
--------------------------------------------------------------------------------
After deleting set Y:
RBT: 8345 tot. rotations req., height is 14, #nodes is 2124, #comparisons is 46138.
--------------------------------------------------------------------------------
After deleting set Y:
BST: height is 1071, #nodes is 2124, #comparisons is 1792106.
--------------------------------------------------------------------------------
```

Figure 10 – Output 2 showing statistics of the AVL, BR, and Unbalanced BS Trees

Result Analysis (Delete)

Figure 9 (Output 1)

- For the AVL tree, it required a total of 1635 rotations, and its height is 11, with 866 nodes and 22501 comparisons. This suggests that the AVL tree has performed well, as it has a low number of rotations and comparisons, and a reasonable height.

- For the Red-Black tree, it required a total of 5729 rotations, and its height is 13, with 1434 nodes and 33585 comparisons. This indicates that the Red-Black tree has not performed as well as the AVL tree, as it has a higher number of rotations and comparisons, and a higher height.
- For the Binary Search Tree, it has a height of 729, with 1434 nodes and 980209 comparisons. This shows that the Binary Search Tree has not performed well, as it has a very high number of comparisons and a very high height. In general, Binary Search Trees that have unbalanced structures that can lead to a high height and poor performance.

- Therefore, the output suggests that the AVL tree is the best performing tree in this scenario, followed by the Red-Black tree, while the Binary Search Tree has performed the worst. However, it's worth noting that the performance of each type of tree can vary depending on the specific use case and the distribution of the elements in the tree.

Figure 10 (Output 2)

- For the AVL tree, it required a total of 2379 rotations, and its height is 12, with 1729 nodes and 29594 comparisons. Although the number of rotations and comparisons have increased from the previous example, the AVL tree is still performing well, as it has a reasonable height and a low number of comparisons and rotations.

- For the Red-Black tree, it required a total of 8345 rotations, and its height is 14, with 2124 nodes and 46138 comparisons. This indicates that the Red-Black tree has not performed as well as the AVL tree, as it has a higher number of rotations and comparisons, and a higher height.

- For the Binary Search Tree, it has a height of 1071, with 2124 nodes and 1792106 comparisons. This shows that the Binary Search Tree has not performed well, as it has a very high number of comparisons and a very high height.

- Overall, the output suggests that the AVL tree is still the best performing tree in this scenario, followed by the Red-Black tree, while the Binary Search Tree has performed the worst.

In conclusion, the analysis of the output from the delete operations performed on AVL, Red-Black, and Binary Search Trees reveals that the **AVL tree outperformed the other two trees in both scenarios.** The AVL tree required a low number of rotations and comparisons and had a reasonable height, suggesting that it is a well-balanced and efficient data structure. **The Red-Black tree also performed well, but not as well as the AVL tree**, with a higher number of rotations and comparisons and a higher height. The Binary Search Tree, on the other hand, performed poorly in both scenarios, with a very high number of comparisons and a very high height. Overall, these results highlight the importance of choosing an appropriate data structure for a particular use case and demonstrate that the performance of each type of tree can vary depending on the specific distribution of elements in the tree.

Searching in Trees Statistics

It was assumed that the search for the *set Z* was done on the same tree that *set Y* was previously deleted from.

After searching every element in the *set Z* in each of the three trees, the number of comparisons, the amount of numbers found and the amount of numbers not found were outputted.

Outputs

Output of the statistics of all three trees:

```
After searching for every element in the set Z:

AVL: 7941 total comparisons required, 186 numbers found, 586 numbers not found.

--------------------------------------------------------------------------------

After searching for every element in the set Z:

RBT: 41638 total comparisons required, 186 numbers found, 586 numbers not found

--------------------------------------------------------------------------------

After searching for every element in the set Z:

BST: 186 total comparisons required, 186 numbers found, 586 numbers not found.

--------------------------------------------------------------------------------
```

Figure 11 – Output 1 showing statistics of the AVL, BR, and Unbalanced BS Trees

```
After searching for every element in the set Z:

AVL: 8096 total comparisons required, 261 numbers found, 479 numbers not found.

--------------------------------------------------------------------------------

After searching for every element in the set Z:

RBT: 54230 total comparisons required, 261 numbers found, 479 numbers not found.

--------------------------------------------------------------------------------

After searching for every element in the set Z:

BST: 261 total comparisons required, 261 numbers found, 479 numbers not found.

--------------------------------------------------------------------------------
```

Figure 12 – Output 2 showing statistics of the AVL, BR, and Unbalanced BS Trees

Result Analysis (Search)

Figure 11 (Output 1)

- The number of comparisons required for each search operation is a good metric to compare the performance of the trees, as fewer comparisons generally indicate a faster search operation.

- Looking at the output, we can see that the BST performed the best in terms of the total number of comparisons required, with only 186 comparisons needed to search through the *set Z*. This is significantly better than both the AVL Tree and the RBT, which required 7941 and 41638 comparisons respectively.

- However, it's important to note that all three trees were able to find the same number of elements, with 186 numbers found and 586 numbers not found. This means that the BST had the best search performance in terms of the number of comparisons required.

Figure 12 (Output 2)

- Looking at the output, we can see that the BST once again performed the best in terms of the total number of comparisons required, with only 261 comparisons needed to search through the *set Z*. This is significantly better than both the AVL Tree and the RBT, which required 8096 and 54230 comparisons respectively.

- It's interesting to note that the number of elements found by each tree is the same, with 261 numbers found and 479 numbers not found. This means that the BST was able to find all the elements in the *set Z* with the least number of comparisons, making it the most efficient in this particular scenario. However, it's important to note that the performance of the trees may vary depending on the specific characteristics of the data being searched and other factors such as tree structure or memory usage.

Based on the two outputs, it appears that the **Unbalanced Binary Search Tree (BST) outperforms both the AVL Tree and the Red Black Tree (RBT) in terms of search performance when comparing the total number of comparisons required.** The BST required significantly fewer comparisons to search through the sets compared to the AVL Tree and the RBT, and was able to find all the elements in both sets with the least number of comparisons.

Comparing the AVL Tree and the Red Black Tree (RBT), **we can see that the RBT requires more comparisons than the AVL Tree in both outputs.** This may be due to the fact that RBTs have more complex balancing rules than AVL Trees, which can lead to more complex operations during insertions and deletions. **However, RBTs do have the advantage of having a more balanced tree structure, which can result in faster search operations in some cases.** [13]

AVL Trees VS Red-Black Trees

The findings from the statistics output of AVL trees and Red Black trees in terms of insertion deletion and searching operations show that AVL trees tend to be slightly more efficient than Red Black trees. This is evident from the lower number of rotations and comparison operations required by AVL trees compared to Red Black trees in both Output 1 and Output 2.

For insertion, in Output 1, the AVL tree required 1611 rotations while the Red Black tree required 5561 rotations. In Output 2, the AVL tree required 2375 rotations while the Red Black tree required 8263 rotations. Similarly, the AVL tree required fewer comparison operations than the Red Black tree in both outputs. These results suggest that AVL trees are more efficient than Red Black trees for insertion and deletion operations, particularly when dealing with a larger number of nodes.

Regarding search operations, the statistics output shows that Red Black trees may have a slight advantage over AVL trees due to their more relaxed balancing rules and use of colour flips. However, the unbalanced Binary Search Tree (BST) outperformed both AVL trees and Red Black trees in terms of the total number of comparisons required to search through the sets in both Output 1 and Output 2.

Overall, the results suggest that AVL trees are a more efficient option than Red Black trees for insertion and deletion operations, particularly when dealing with a larger number of nodes. However, the choice between AVL trees and Red Black trees ultimately depends on the specific use case, the implementation of the algorithm and the characteristics of the data being stored in the tree. Additionally, the statistics output demonstrates that the choice of an appropriate data structure is critical in achieving efficient operations.

Conclusion

In conclusion, the analysis of the statistics of the AVL Tree, Red Black Tree, and Binary Search Tree (BST) provides insights into the performance of these three types of trees in different operations, including insertion, deletion, and search. The results suggest that the AVL tree and Red Black tree are balanced trees that require fewer comparison operations compared to the unbalanced Binary Search Tree, making them more efficient for insertion and deletion operations. **The AVL tree consistently outperformed the Red Black tree in terms of rotations and comparison operations, making it a slightly more efficient option for insertion and deletion. However, the Red Black Tree has the advantage of having a more balanced tree structure, which can result in faster search operations in some cases.** On the other hand, the Binary Search Tree consistently had the highest height and required significantly more comparison operations, making it less efficient for searching and insertion operations. [14]

When it comes to the search operation, the BST outperformed both the AVL Tree and the RBT in terms of the total number of comparisons required to search through the sets. The BST was able to find all the elements in both sets with the least number of comparisons, making it the most efficient in this particular scenario. However, it's important to note that the performance of the trees may vary depending on the specific characteristics of the data being searched and other factors such as tree structure or memory usage.

Overall, the results highlight the importance of choosing an appropriate data structure for a particular use case and demonstrate that the performance of each type of tree can vary depending on the specific distribution of elements in the tree. The analysis provides valuable insights for developers and researchers working with trees, helping them to make informed decisions on which type of tree to use for their specific use cases. [15][16]

References

[1] "Binary Search Tree vs AVL Tree," JavaTpoint, [Online]. Available: https://www.javatpoint.com/binary-search-tree-vs-avl-tree. [Accessed: Mar. 10, 2023].

[2] "Red-Black Tree vs AVL Tree," GeeksforGeeks, [Online]. Available: https://www.geeksforgeeks.org/red-black-tree-vs-avl-tree/. [Accessed: Mar. 10, 2023].

[3] "random — Generate pseudo-random numbers," Python 3.9.7 documentation, [Online]. Available: https://docs.python.org/3/library/random.html. [Accessed: Mar. 10, 2023].

[4] "AVL Tree," JavaTpoint, [Online]. Available: https://www.javatpoint.com/avl-tree. [Accessed: Mar. 10, 2023].

[5] "Insertion in an AVL Tree," GeeksforGeeks, [Online]. Available: https://www.geeksforgeeks.org/insertion-in-an-avl-tree/. [Accessed: Mar. 1, 2023].

[6] "AVL Tree in Data Structure," Programiz, [Online]. Available: https://www.programiz.com/dsa/avl-tree. [Accessed: Mar. 1, 2023].

[7] "Introduction to Red-Black Tree," GeeksforGeeks, [Online]. Available: https://www.geeksforgeeks.org/introduction-to-red-black-tree/. [Accessed: Mar. 5, 2023].

[8] "Red-Black Tree in Python," FavTutor, [Online]. Available: https://favtutor.com/blogs/red-black-tree-python#:~:text=A%20red-black%20tree%20is%20the%20variation%20of%20the%20binary,specifically%2C%20either%20red%20or%20black. [Accessed: Mar. 5, 2023].

[9] P. Morin, Open Data Structures: An Introduction, 0.1 ed. Waterloo, Canada: University of Waterloo, 2013, ch. 6.2, "BinarySearchTree - An Unbalanced Binary Search Tree."

[10] "Search n elements in an unbalanced Binary Search Tree in O(Logm) time," GeeksforGeeks, [Online]. Available: https://www.geeksforgeeks.org/search-n-elements-in-an-unbalanced-binary-search-tree-in-on-logm-time/. [Accessed: Mar. 6, 2023].

[11] P. Morin, Open Data Structures: An Introduction, 0.1 ed. Waterloo, Canada: University of Waterloo, 2013, ch. 6.1, "Binary Tree - An Unbalanced Binary Tree."

[12] "Red-Black Tree in Data Structure," Programiz, [Online]. Available: https://www.programiz.com/dsa/red-black-tree. [Accessed: Mar. 6, 2023].

[13] "Red-Black Tree vs AVL Tree," Baeldung, [Online]. Available: https://www.baeldung.com/cs/red-black-tree-vs-avl-tree. [Accessed: Mar. 10, 2023].

[14] "Red-Black Tree over AVL Tree," Stack Overflow, [Online]. Available: https://stackoverflow.com/questions/13852870/red-black-tree-over-avl-tree. [Accessed: Mar. 10, 2023].

[15] "Red-Black Tree vs AVL Tree," Coding Ninjas, [Online]. Available: https://www.codingninjas.com/codestudio/library/red-black-tree-vs-avl-tree. [Accessed: Mar. 10, 2023].

[16] "AVL Tree and Red Black Tree in Data Structure," YouTube, Jun. 4, 2021. [Online]. Available: https://www.youtube.com/watch?v=EmxWC18iuSc. [Accessed: Mar. 10, 2023].