Analysis Report

Guided By:

Dr. Sukanta Bhattacharjee

Submitted By:

Atul Bunkar , 214101011

Contents:

Problem Statement:

To Compare the performance of treap with Binary search tree and AVL tree by running them through a large number of operations, several times and observing several parameter outcomes.

Brief Description of trees:

Treap:

- Treap is a Balanced Binary Search Tree, but not guaranteed to have height as O(Log n). The idea is to use Randomization and Binary Heap property to maintain balance with high probability.
- The expected time complexity of search, insert and delete is O(Logn).

Every node of Treap maintains two values.

1) **Key** Follows standard BST ordering (left is smaller and right is greater)

2) **Priority** Randomly assigned value that follows Max/Min-Heap property.

- Like other self-balancing Binary Search Trees, Treap uses rotations to maintain Max-Heap property during insertion and deletion.

- An equivalent way of describing the treap is that it could be formed by inserting the nodes highest-priority-first into a binary search tree without doing any rebalancing.
- The assignment has both “treap.cpp” and “treap.h” files included.

Binary Search Tree:

- A binary search tree (BST), also called an ordered or sorted binary tree, is a rooted binary tree data structure whose internal nodes each store a key greater than all the keys in the node’s left subtree and less than those in its right subtree.

- For this assignment, I have used a simple BST instead of a threaded BST.

- For a long sequence of random insertion and deletion, the expected height of the tree approaches square root of number of nodes, √n.

- Header File “bst.h” is used for this assignment which is accessed by the main.cpp program.

AVL tree:

- AVL tree is binary search tree with additional property that difference between height of left sub-tree and right sub-tree of any node cannot be more than 1.

- Rebalancing of tree is done by rotation to restore this property.
- Lookup, insertion, and deletion all take O(log n) time in both the average and worst cases, where n is the number of nodes in the tree prior to the operation.

- Insertions and deletions may require the tree to be rebalanced by one or more tree rotations.
- Header File "avl.h" is used for this assignment which is accessed by the main.cpp program.

Program Logic And Flow:

Here are the Programs Created for this assignment and their Logic:

1. **Treap.cpp**

   This program implements the treap data structure. I have also created "treap.h" from this to use in main.cpp program which is same as this with some minor changes to suit the analysis.

   Functions Used:

   - **rotRight(Node)** - Rotates the tree right with node as pivot. This is done when priority of left child is less (higher priority) than its parent. This makes the left child as the parent.

- **rotLeft(Node)** - Rotates the tree left with node as pivot. This is done when priority of right child is less (higher priority) than its parent. This makes the right child as the parent.

- **Search(x)** - Searches for x and returns true if present else false.

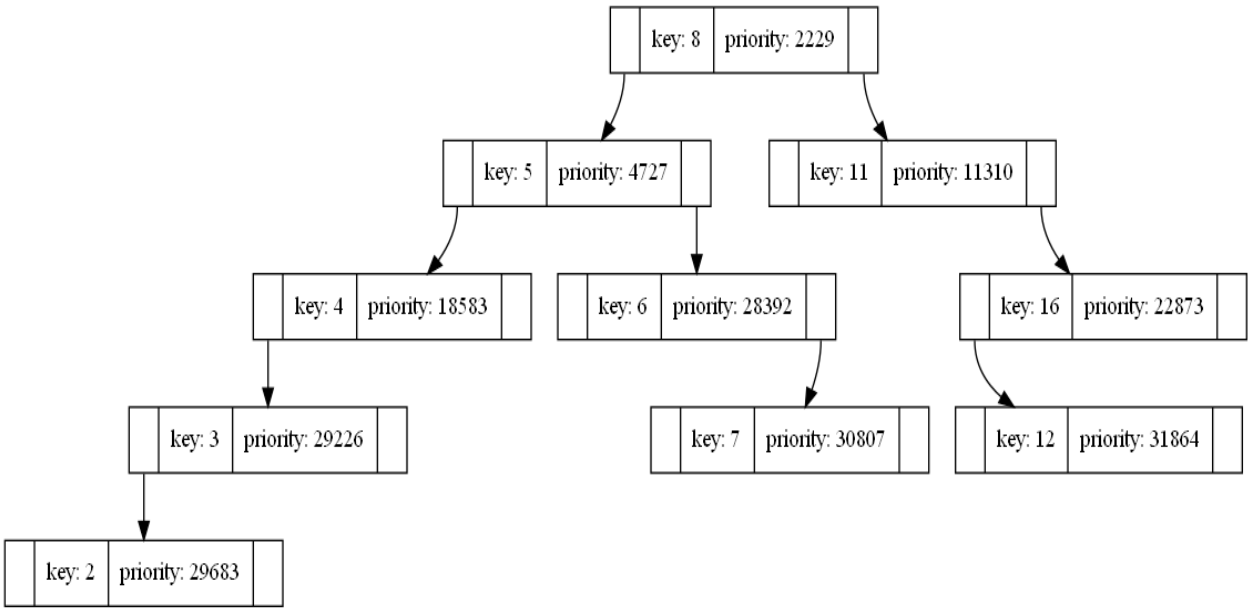
- **insert(x)** - To insert element x based on key, if larger, it goes to right subtree, else goes to left subtree. And if its priority is minimum, the tree is rotated until the node becomes the parent of all nodes that have priority larger than that.

- **Delete(x)** - Deletes element x if present by searching it , if x is larger than current node, search right subtree else search left subtree. Finally, when found, delete the node and appropriately fix the tree pointers , so no node is lost. Also , If deleted node has both children , the one with min priority is set in place of node(made parent) by rotating the tree accordingly.

- **print**() - prints the tree structure as .png image. User has to enter a random file name which is created as .png file.

- **DeleteTree**() - Recursively deletes the tree nodes, freeing the memory before exiting the program. Called by the destructor.

This is how a generic Treap looks (got from print() function) :



Each Node has key and priority. Notice keys on right subtree are larger than root and of left subtree are less than root key. This follows the BST property.

And the priorities of all the children are larger than the parent node. This follows the heap property.

2. **Test_gen.cpp**

→ This program generates test files for the analysis of trees. User must provide test file size i.e., no. of operations for each test file generation.

→ Each test file has a different ratio of Insert to Delete operations.

→ In total 5 test files are created which are used to run the three Binary trees on.

→ There are only 2 operations in each test files vis-a-vis Insert and Delete followed by a random number that is between 0 and Size of Operation but not necessarily limited to it.

→ Number following Delete is generally taken from number following Insert and very marginally taken from outside it.

→The 5 test files generated are named test1.txt, test2.txt and so on.

3. Main.cpp

The crux of the assignment is done in this program.

Here is the Flow of the program:

→Parameters initialization at the very beginning.
→Importing the 3 trees from header files.
→Then, in main():

Execution

- Executing the test files either on TAs own file (single) or on the 5 test files created.
- For this, the execute() function is called which runs the trees through the test files and generated the Evalution Parameters.
- For each test file, the parameters are stored in arrays to analyze later.

Analysis

- All the parameters are printed to screen and written to "Analysis.txt" to evaluate.

Parameters Description:

To Evaluate the performance of treap with BST and AVL trees, the trees were run under a large number of interleaved operations of Insertion and Deletion.

Meanwhile several parameters were calculated, some during the operation, while some after overall operations.

These parameters tells about the overall performance of the trees and compares which tree is overall the best performing of the three.

Following were the parameters used:

1. **Total Key-Comparisons**:

   This tells the overall count of keys compared while searching a node's location when an element is inserted or deleted from the tree.

   It is the sum of all comparisons done for insert and delete method.

   The larger the key-comparisons needed by the tree, the slower its execution is , thus it correlates with the time complexity of the tree.

   Theoretically, the key comparisons needed for Insert or delete in all these 3 trees is O(log(n)), where 'n' is the nodes in tree at time of operation.

   Thus, it can be used to compare with the empirical values.

2. **Total Rotations**:

For Treap and AVL trees, there are tree rotations done to satisfy its individual properties.

For AVL, it is done when the balance factor becomes > +-1.

For Treap, it is when the priority of child is more than the parent.

Rotations take some extra time on its own, so a large no of rotations increases the overall runtime of the tree.

It is thus a good parameter to analyze the performance of trees.

3. **Final Height of tree**:

Returns the height (distance of root from the farthest leaf) of tree after all operations are done. The height of tree is a direct measure of its basic operations.

Executions of operations like search, insert, delete, etc. are generally of O(h), h being the height of tree.

Thus, smaller the tree height, faster is its execution.

Thus, it is one of the prime factors to decide the performance of tree.

Also, theoretical height of trees are:

Treap : log2(n) ; n-> total tree nodes.

BST: 4.311*log(n) - 1.953*(log(log(n)) for large n.

AVL : 1.44 * log2(n) for large n.

So, I have used these values to compare with the tree's empirical values as well.

4. **Average node height**:

This tells the average height of each node (distance from child leaf) in the tree. It can be correlated with the balance of tree.

More the tree is balanced, more number of nodes have 0 or close to it height.

Thus, if Average node height is less, the tree is more balanced overall and is better in performance.

This is because as the tree becomes inclined to one side, the overall time complexity of basic operations tends to O(n).

Observations:

The 5 test files with following specifications were used for the analysis:

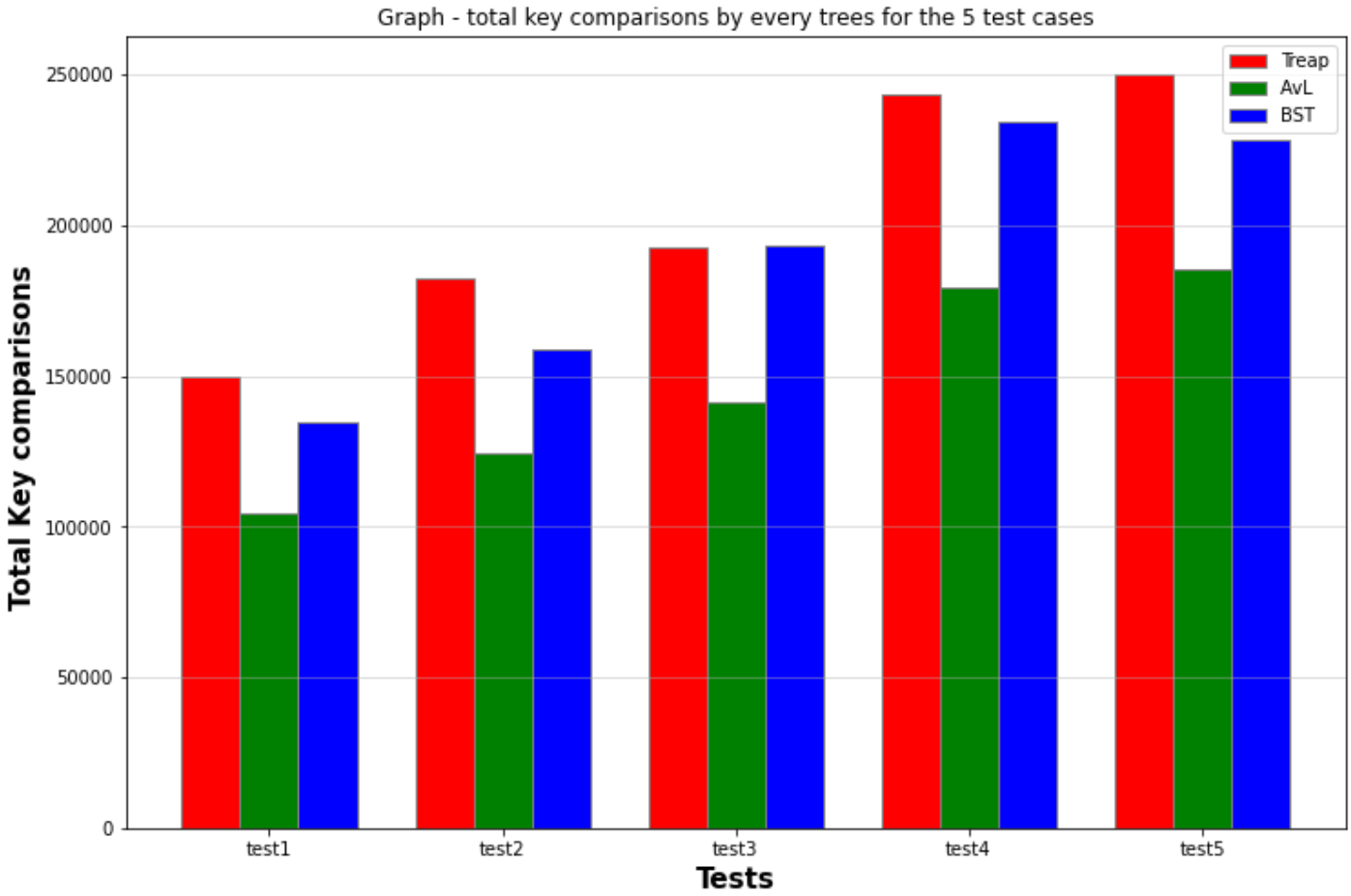
Test1 – Operations: 10k , Ratio of Insert: Delete : 0.7

Test2 - Operations : 12k , Ratio of Insert: Delete : 0.6

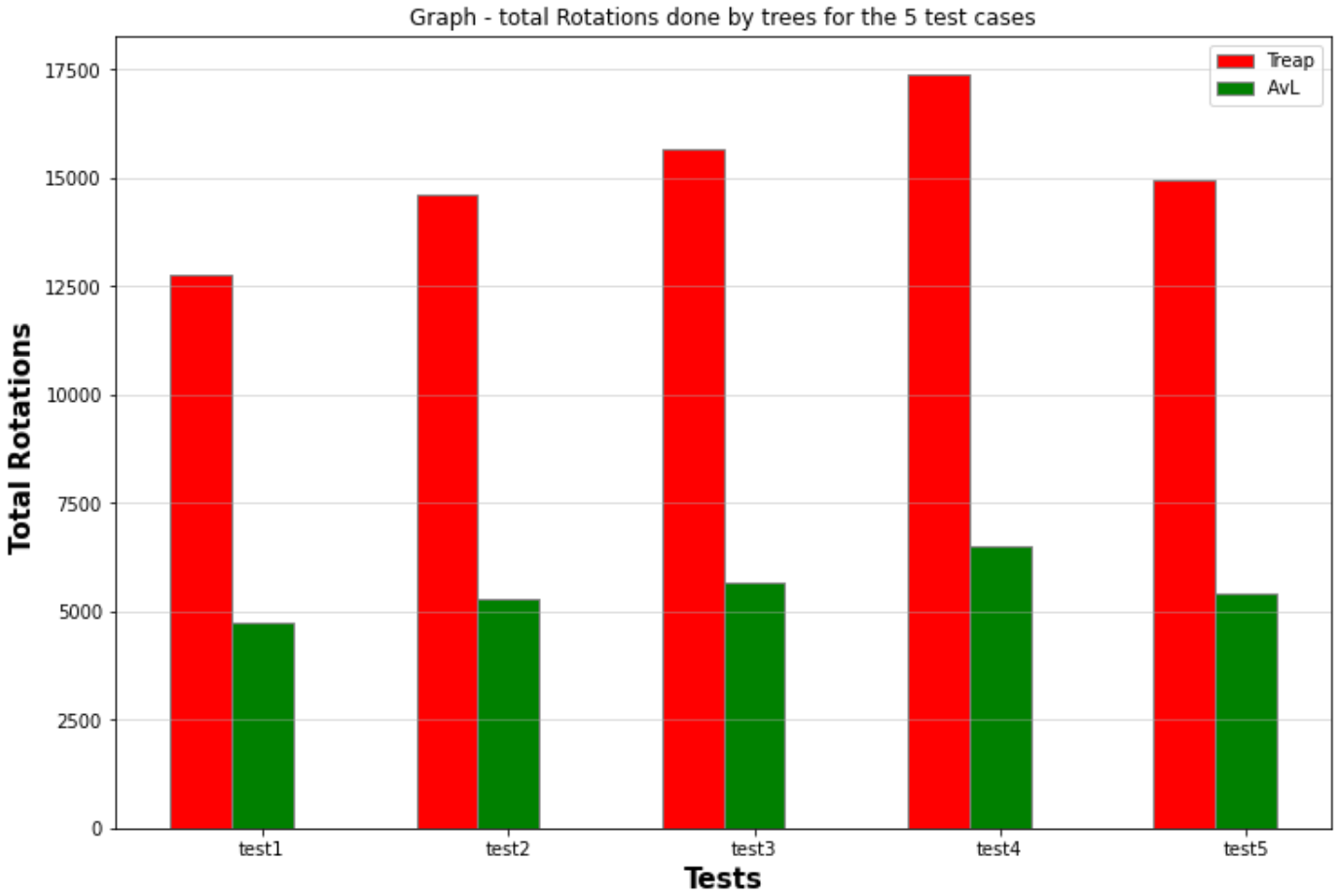
Test3 - Operations : 14k , Ratio of Insert: Delete : 0.5

Test4 - Operations : 18k , Ratio of Insert: Delete 0.4

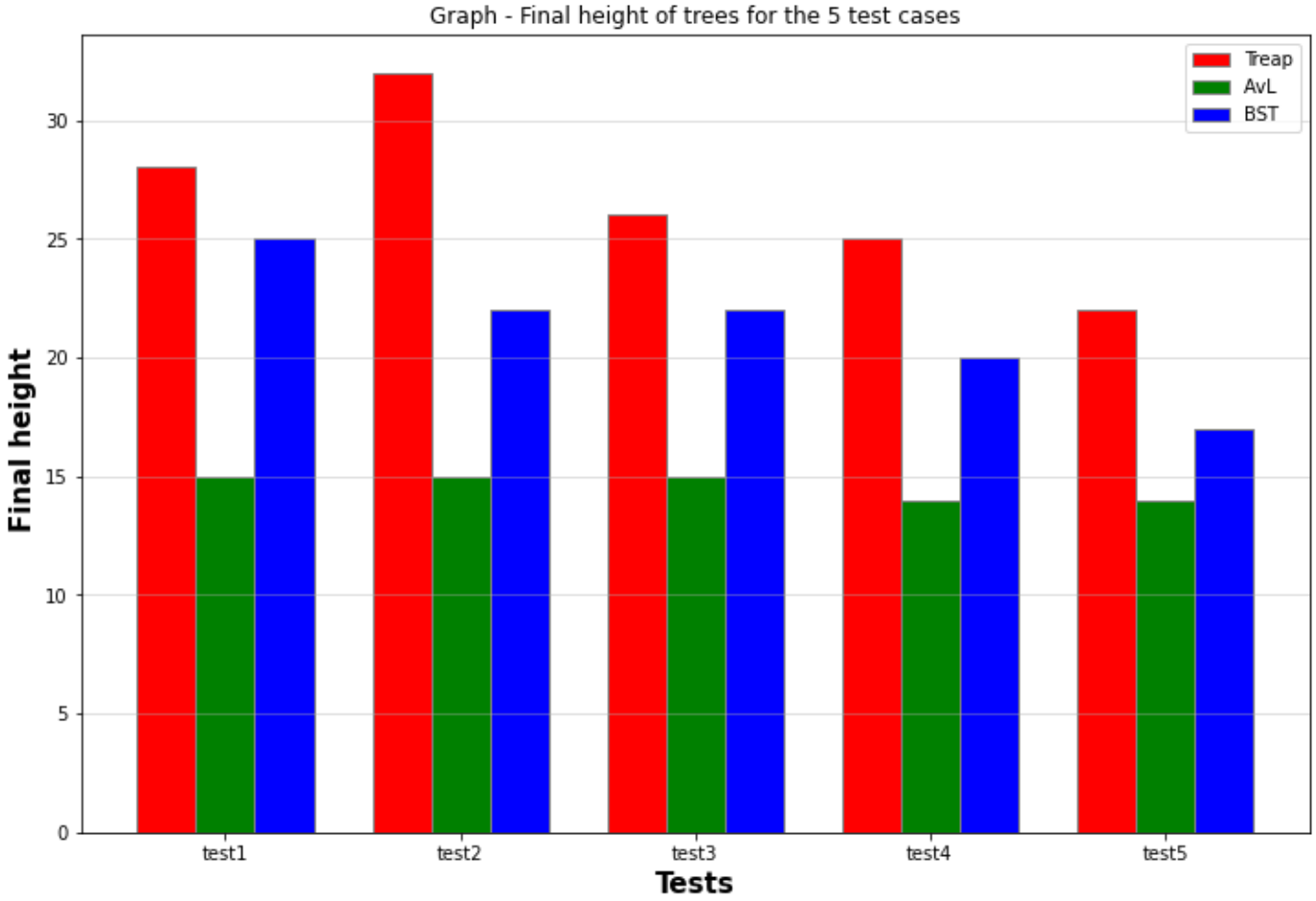
Test5 - Operations : 20k , Ratio of Insert: Delete : 0.3



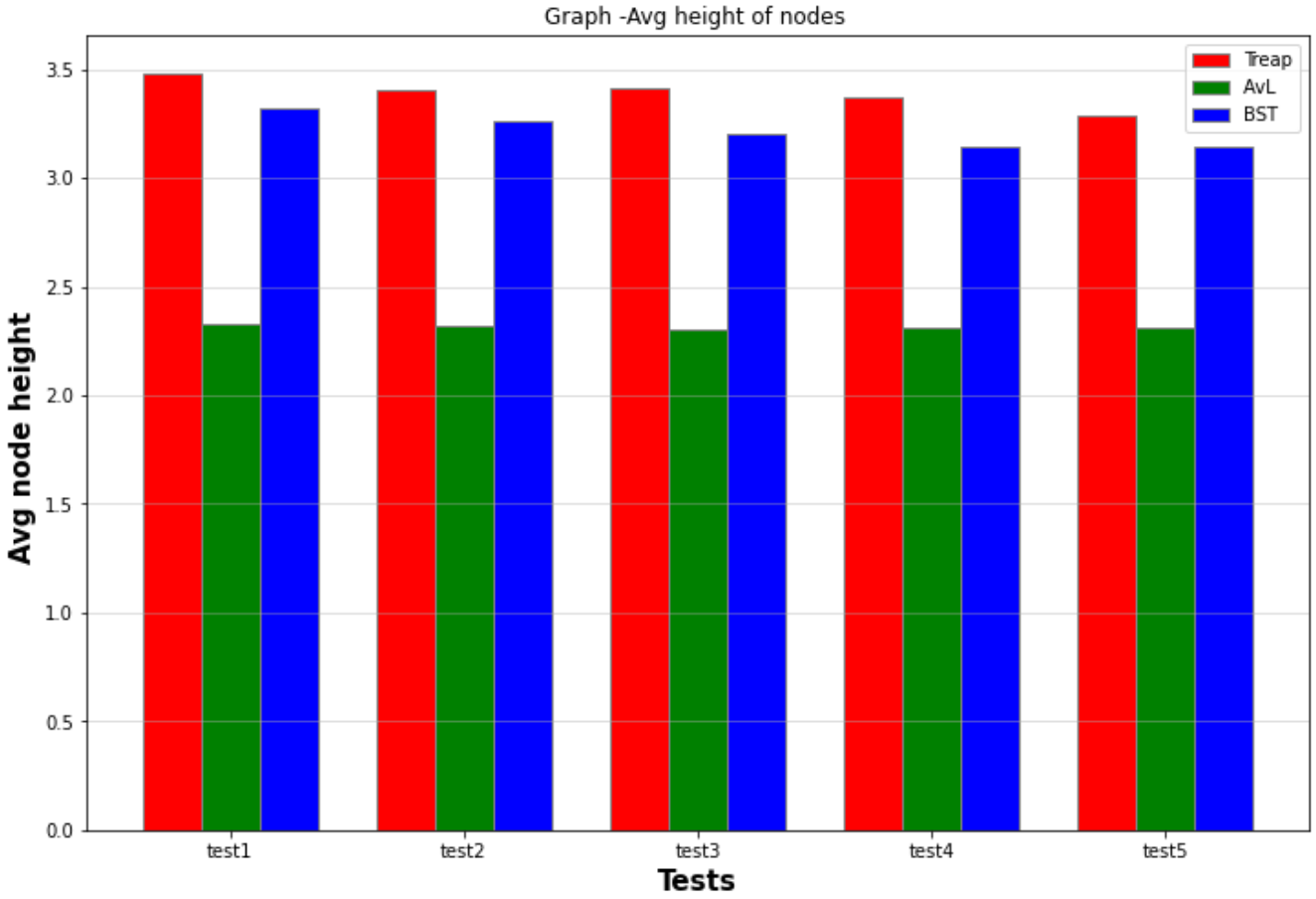
- ❑ Looking at the total key comparisons done by the 3 trees (insert+delete) for all test cases, irrespective of the ratio of insertion and deletion, AVL tree performs best among the three with least no of comparisons done for all test cases.
- ❑ Treap is however close to, but still worse than BST in all the cases, making it the slowest tree here.
- ❑ The reason AVL has the least comparison is because it is well balanced, while treap and BST are not.



- ❑ For the total rotations done for each test case, AVL has a far lesser count compared to Treap .
- ❑ Treap does about 2.5 times the rotations of AVL, which is very bad for the performance.
- ❑ Since the priority is randomly generated, rotations are arbitrary in treap and thus very frequently performed.

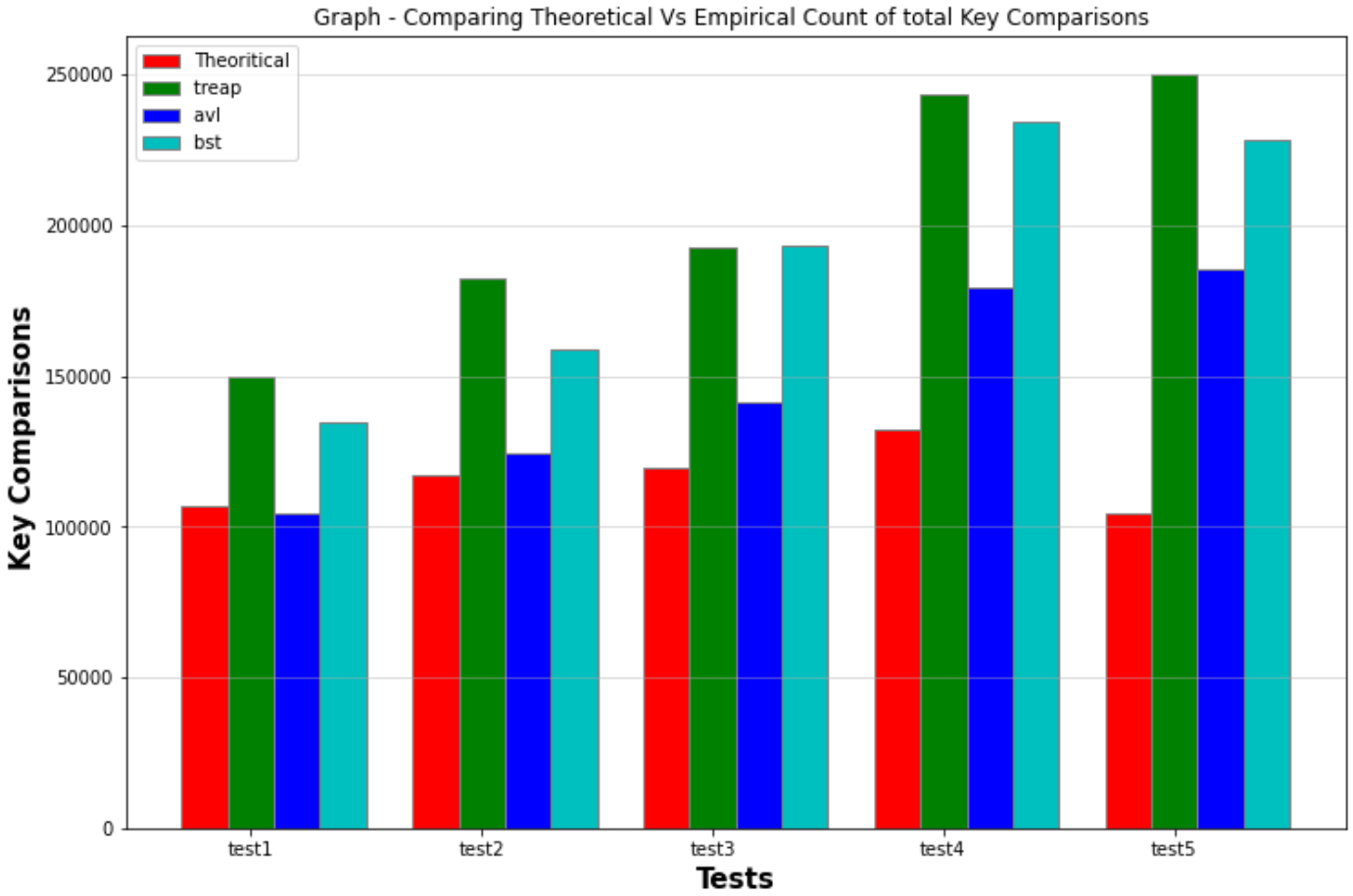


- ❑ Looking at the final height for all test cases, Treap has the largest height for all case, while AVL has the smallest height.
- ❑ Treap has about 1.5x the height of AVL, which comparatively makes its operations very slow.
- ❑ Comparing with BST, Treap has slightly large height in all cases, but not much.
- ❑ Large height of treap can be a result of inclination due to random priority and no balancing factor present like AVL.

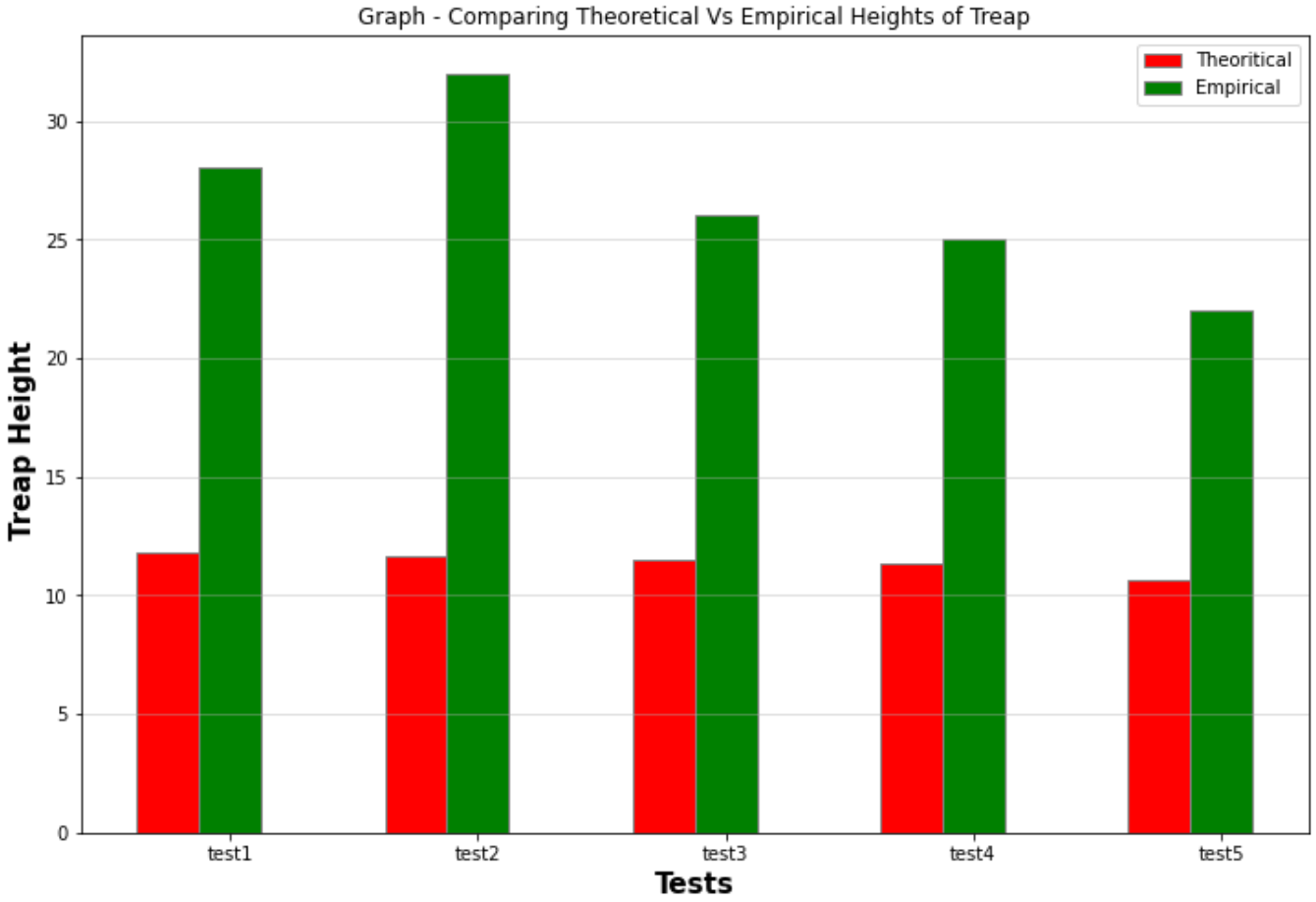


- ❑ Looking at the average node heights, we see that Treap and BST are very close for this, BST still better than Treap.
- ❑ AVL again have the least avg node height, meaning it is very well balanced, while BST and Treap are not so balanced.
- ❑ This means the lookup time in general deviates from log(n) for Treap and BST, making them slower.

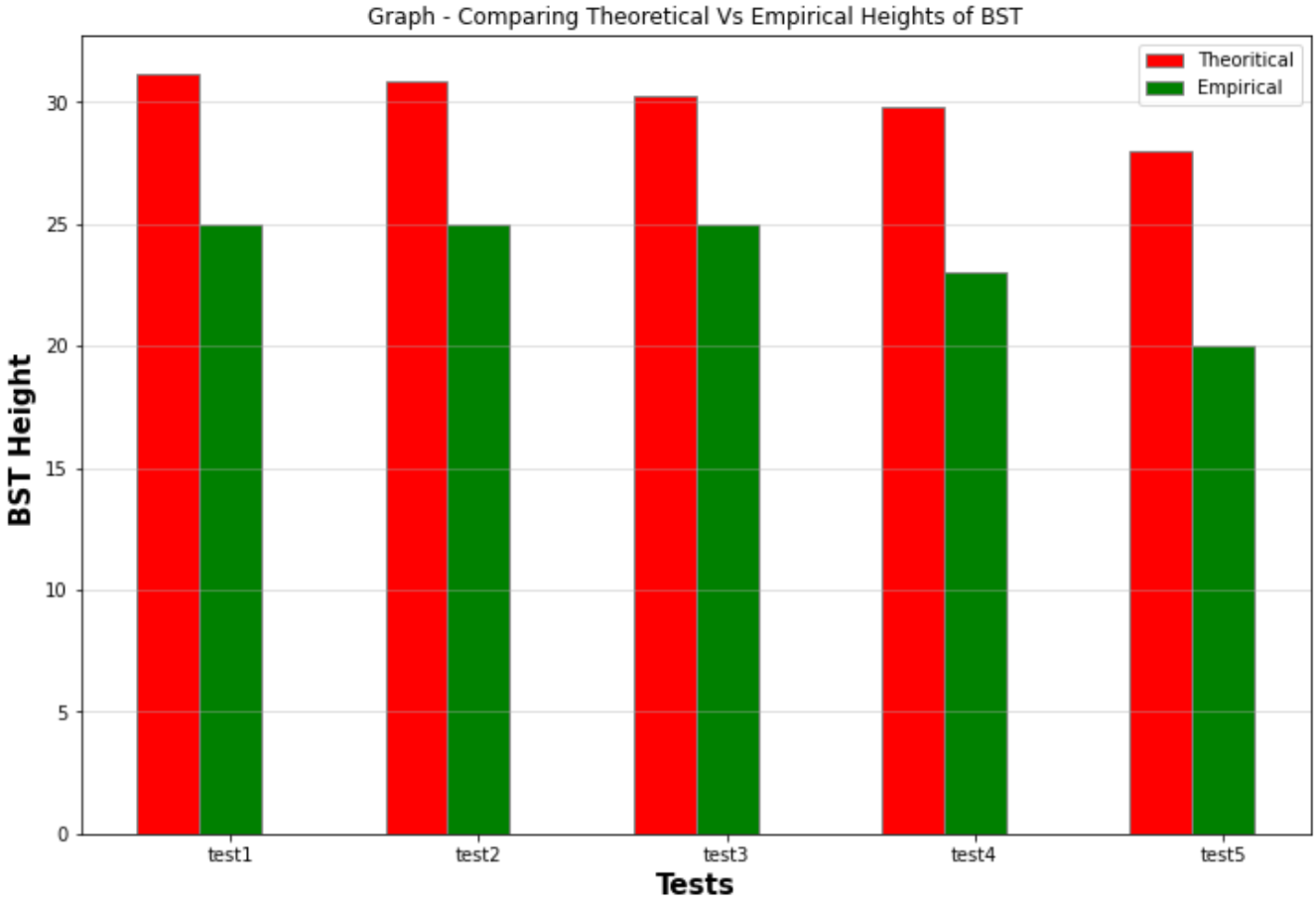
Comparing theoretical and empirical values:



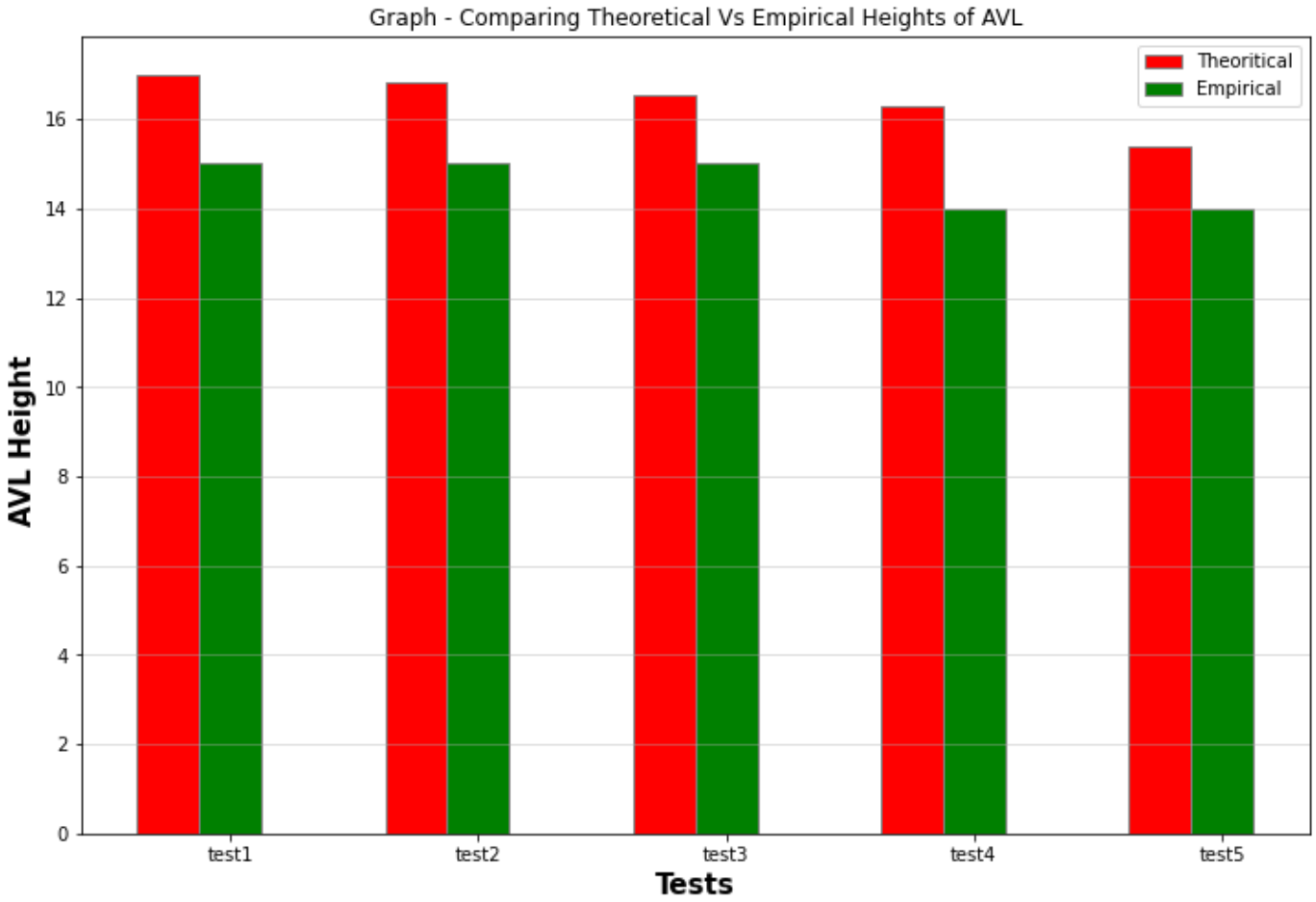
- Theoretically, key comparisons = log (n) ; n -> nodes currently in tree, for all the 3 trees.
- Comparing this value with the data we got, we see, as the test case size increases in tests, the actual data becomes very far from theoretical data.
- Initially, AVL was the closest to theoretical and Treap the farthest. Still, they were comparable when test case size was small.
- By test 5(20k operations) , all tree's actual values were about 2-2.5 times the theoretical value.



- ❑ Theoretical height of treap is : log(n)
- ❑ Looking at the graph, we can say that there is very large difference between the actual height and the theoretical height.
- ❑ In reality, treaps are not balanced and thus inclined towards one side making their actual height far larger than the theoretical height.



- Theoretical height of BST : 4.311*log(n) - 1.953*(log(log(n)) ; for very large no. of nodes . (and not log(n))
- Comparing It with actual height, we see that they are fairly close, giving a good approximation.



- ❑ Theoretical height of AVL: 1.44* log(n), for large n.
- ❑ From the graph, we see this to be quite true for all test cases. The values are fairly close, and can be a good approximation.

Conclusion:

Looking at all the observations, we can fairly say:

→Among the three Binary Trees, Treap is the worst performing tree, closely followed by BST.

→AVL is far better performing than the other trees based on all the parameters, even remaining true to its theoretical approximations.

→Treap does not follow its theoretical approximation.

----------End of Report----------