DS LAB CS513: Assignment 03 (TREAP PERFORMANCE)

Vijay Purohit | 214101058

M.Tech CSE, IITG

Contents

List of Figures

1 FILES AND FUNCTIONS DESCRIPTION

1.1 ANode.h

This header file contains node structure for AVL Tree. Node Name is "ANode". Node Structure consists of key, balance factor, height, left child pointer and right child pointer.

1.2 AVLClass.h

This header file contains the definition of functions for the Class AVL "AVLTree". It contains the functions for insert operation, delete operation and other necessary helper functions. AVL Tree is implemented using Balance Factor i.e using code of assignment 02 after some modifications for this assignment.

1.3 BNode.h

This header file contains node structure for Threaded Binary Search Tree. Node Name is "BNode". Node Structure consists of key, height, left child pointer, right child pointer, left thread boolean value and right thread boolean value.

1.4 BSTClass.h

This header file contains the definition of functions for Class Threaded BST "ThBST". It contains the functions for insert operation, delete operation and other necessary helper functions. BST is implemented using Threads i.e using code of assignment 01 after some modifications for this assignment.

1.5 TNode.h

This header file contains node structure for TREAP. Node Name is "TNode". Node Structure consists of key, priority, height, left child pointer, right child pointer.

1.6 TreapClass.h

This header file contains the definition of functions for TREAP Class "Treap". It contains the functions for insert operation, delete operation and other necessary helper functions.

1.6.1 Copy Constructor Functions

- **Treap(const Treap &o_obj))**
  This is copy constructor which is used to copy one Treap Class object to another Treap Class object.
- **void operator=(const Treap &o_obj))**
  This is overloading of assignment operator, used to copy from one Treap Class object to another Treap Class object.
- **TNode* copyLeftRightNode(TNode* tnode)**
  This function is a helper function for copy constructor, taking input as root and recursively copying the tree structure. It creates clone by copying key, height, left and right pointers.

1.6.2 void t_display(TNode *tnode, int level)

This is helper function for Displaying Tree in the console for better run-time visualization and proper operations handling at run-time. It start from printing level as 1 as input and taking root as input.

1.6.3 int Search(int item)

This function is used for searching element key in the Treap. The Average time complexity is $O(\log n)$ and in worst case it is $O(n)$.

1.6.4 void Insert(int item)

This is insert function for inserting a node key 'item' into Treap and after that calling helper function TNode * t_insert(TNode *root, int e, int p). If element is already present then it will throw a exception. Priorities are generated randomly using rand() function.

1.6.5 void Insert(int item, int priority)

This is insert function for inserting a node key 'item' with 'priority' into Treap and after that calling helper function TNode * t_insert(TNode *root, int e, int p). If element is already present then it will throw a exception.

1.6.6 TNode * t_insert(TNode *root, int e, int p)

This is helper recursive function for inserting into Treap and performing necessary rotations. Inputs are item 'e' to be inserted and Priority 'p' value.

- **LL Imbalance (right rotation)**
  This part calls another function (Refer 1) which performs the right single rotation.
- **RR Imbalance (left rotation)**
  This part calls another function (Refer 2) which performs the left single rotation.

The Average time complexity is $O(\log n)$ and in worst case it is $O(n)$.

1.6.7 void Delete(int item)

This is delete function for deleting a node key 'item' from Treap and after that calling helper function TNode * t_delete(TNode * root, int e). If element is already present then it will throw a exception.

1.6.8 TNode * t_delete(TNode * root, int e)

This is helper recursive function for deleting from Treap and after that performing necessary rotations. Input is the item to be deleted.

- **LL Imbalance (right rotation)**
  This part calls another function (Refer 1) which performs the right single rotation.
- **RR Imbalance (left rotation)**
  This part calls another function (Refer 2) which performs the left single rotation.

The Average time complexity is $O(\log n)$ and in worst case it is $O(n)$.

1.6.9 Rotation Functions

1. **TNode* Imbalance_LL(TNode *root)**
   This function is used to perform right rotation whenever there is LL Imbalance at the 'root'. (Figure 1a)
2. **TNode* Imbalance_RR(TNode *root)**
   This function is used to perform left rotation whenever there is RR Imbalance at the node 'root'. (Figure 1b)

1.6.10 void print_graphviz(string file_name)

This function does a level order traversal of the tree and generate two files according to file name provided. One is '.gv' file for graph-viz to read and another final 'png' output file of the tree. It will automatically execute the graph-viz command to generate the image file.

1.6.11 unsigned long int calculate_sum_of_nodes_heights(TNode *)

This recursive function is used to get Average Height of the Nodes of the Tree. It calculate sum of height of each node and then dividing by total nodes.

1.7 main_TREAP.cpp

This is the 'cpp' file for the TREAP implementation part containing interactive menu for the operations for insert, delete, search and to print the tree. It uses two header files 'TNode.h' and 'TreapClass.h'.

1.8 main.cpp

This 'cpp' file contain operations for test file generations and evaluating the parameters for the AVL, Threaded BST and TREAP. It contains header files of the respective trees for the operations.

1.8.1 void GenerateDefaultTestCases()

This function is used to generate default test cases files which contains operations for the insert and delete in randomized manner.
Files are named "nop_+*number of operations*+.txt" and stored in the folder 'input_test_files' present in the same directory.
Use global variable 'max_op' to change the limit for the maximum number of the operations and each file is generated from initial operation 100 to maximum operations, increment each time with 500 operations.
The number of the insert operation frequency in any file at a time is randomly selected from the global array 'ins_ratio'.

1.8.2 void GenerateTestCases(int nop, float insert_ratio_value)

This function is used for generating test case files from the console menu by the user providing the number of the operations 'nop' and insert frequency 'insert_ratio_value'.
Multiple values can be provided in the console and it will generate all the files with provided values named with "top_+*number of operations*+.txt" and stored in the folder 'input_test_files' present in the main directory.

1.8.3 void ReadDefaultTestCases()

This function is used to read default test case files generated by 1.8.1.
Each file is read and then required operations are performed on the AVL, Threaded BST and Treap. After reading each file, necessary parameters are calculated.
Parameters of each file read are stored in the file named "analysis_default_.csv" and stored in the folder 'output_analysis_files' present in the main directory.

1.8.4 void ReadTestCases()

This function is used to read test case files provided by the user from the console menu.
To read user provided test case files, file names should be stored in the file named '_read_user_test_files.txt' with first line being the number of the files present and each next line will be full name of the file. Files should be present in the same folder.
Each file is then read and then required operations are performed on the AVL, Threaded BST and Treap. After reading each file, necessary parameters are calculated.
Parameters of each file read are stored in the file named "t_+*num of files read*+ _user_test_file_analysis.csv" and stored in the folder 'output_analysis_files' present in the main directory.

2 TREAP OPERATIONS LOGIC

The priority of the Treap Node is generated using the pseudo-random number generator using the library "random" of cpp.

2.1 Print Function Logic

We print the graph using package graphviz. Initially we traverse the tree in Level order manner (BFS Traversal) and store the node information, its children in the file with extension '.gv' according to graphviz format. Then program execute the graphviz command to generate the respective '.png' file. In a program, we can provide the file name suffix in order to distinguish our different operations and generate different '.png' file without overwriting.

2.2 Search Function Logic

Logic is same as standard BST search. Priority is not considered for searching operation. (Function Reference: int Search(int item))

- We are finding the location of search element 'k' starting from the root down to the leaf, until either element exist or we get nullptr reference.
- If it is equal to the current node Key value, then return its reference.
- If it is less than current node Key value, then go to left sub-tree, repeat process.
- Else we go to the right sub-tree and repeat the process.

2.3 Insert Function Logic

Insertion is done using a key like a standard BST insert operation and rotations are performed based on priority. (Function Reference: TNode * t_insert(TNode *root, int e, int p)).

- Create a new node with key equals to 'item' and priority equals to a random value.
- Perform standard BST insert. Once we reach a leaf node we insert the new node there.
- A newly inserted node gets a random priority, so Min-Heap property may be violated. We make use of single rotations to make sure that inserted node's priority follows min heap property. During insertion, we recursively traverse all ancestors of the inserted node.
- If new node is inserted in left subtree and root of left subtree has lower priority, then perform right rotation. (Refer 1).
- If new node is inserted in right subtree and root of right subtree has lower priority, then perform left rotation. (Refer 2).

2.4 Delete Function Logic

Deletion is done by finding the key like a standard BST search operation and after that rotations are performed making that node a leaf node and then finally deleting it. (Function Reference: TNode * t_delete(TNode * root, int e)).

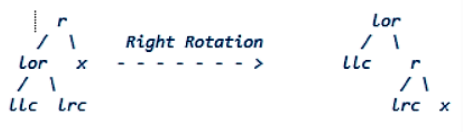
- If node is a leaf node then simply delete it.
- If node has one child, perform necessary rotation to make it leaf node.
- If node has both children as non-NULL, find minimum priority of left and right children.
  - If priority of right child is lesser, perform left rotation at node. (Refer 2).
  - If priority of left child is lesser, perform right rotation at node. (Refer 1).

2.5 LL Imbalance Logic

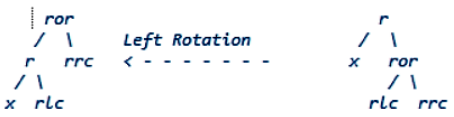
(Rotation Function Reference: 1).
Here we are performing right rotation and change the necessary pointers as shown in figure. (Refer Figure 1a).

2.6 RR Imbalance Logic

(Rotation Function Reference: 2).
Here we are performing left rotation and change the necessary pointers as shown in figure. (Refer Figure 1b).



(a) LL Imbalance



(b) RR Imbalance

Figure 1: Rotation: Single Rotation

3 TREAP TEST CASES

This Section contains the test cases for the TREAP operations performed in the program.

```
------- TREAP MENU  -------
1. Display - Console
2. Default Treap - Insertion

3. INSERT an element k
4. SEARCH an element k
5. DELETE an element k
6. INSERT Multiple elements in a Line

p. Print Tree - GraphViz
n. Exit - Bye

 --Choice :
```

Figure 2: Treap: Main Menu Console

3.1 Search(k)

- **Original Tree**

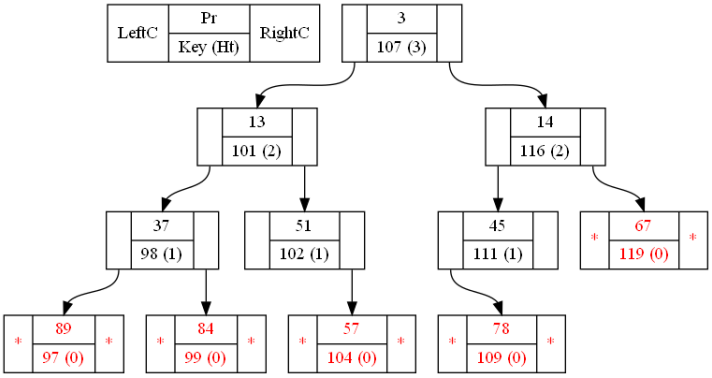


Figure 3: Treap: Searching: Original Tree

- **Treap: Search: 456, 97, 107**

```
                    67|k: 119 (h 0)
               14|k: 116 (h 2)
                         45|k: 111 (h 1)
                              78|k: 109 (h 0)
Root->:3:  3|k: 107 (h 3)
                              57|k: 104 (h 0)
                    51|k: 102 (h 1)
               13|k: 101 (h 2)
                              84|k: 99 (h 0)
                         37|k: 98 (h 1)
                              89|k: 97 (h 0)

------- TREAP MENU  -------
1. Display - Console
2. Default Treap - Insertion

3. INSERT an element k
4. SEARCH an element k
5. DELETE an element k
6. INSERT Multiple elements in a Line

p. Print Tree - GraphViz
n. Exit - Bye

 --Choice : 4

<-------->
Enter a Node value to SEARCH : 456

Element (456) Not Found.
```

(a) Search 456: Not Exist

```
                    67|k: 119 (h 0)
               14|k: 116 (h 2)
                         45|k: 111 (h 1)
                              78|k: 109 (h 0)
Root->:3:  3|k: 107 (h 3)
                              57|k: 104 (h 0)
                    51|k: 102 (h 1)
               13|k: 101 (h 2)
                              84|k: 99 (h 0)
                         37|k: 98 (h 1)
                              89|k: 97 (h 0)

------- TREAP MENU  -------
1. Display - Console
2. Default Treap - Insertion

3. INSERT an element k
4. SEARCH an element k
5. DELETE an element k
6. INSERT Multiple elements in a Line

p. Print Tree - GraphViz
n. Exit - Bye

 --Choice : 4

<-------->
Enter a Node value to SEARCH : 97

Search Successful for Element (97).
       Its parent key and address: 98 , 0xbd1b20
       Its location: 97 , 0xbd6310
```

(b) Search Leaf 97: Exist

```
                    67|k: 119 (h 0)
               14|k: 116 (h 2)
                         45|k: 111 (h 1)
                              78|k: 109 (h 0)
Root->:3:  3|k: 107 (h 3)
                              57|k: 104 (h 0)
                    51|k: 102 (h 1)
               13|k: 101 (h 2)
                              84|k: 99 (h 0)
                         37|k: 98 (h 1)
                              89|k: 97 (h 0)

------- TREAP MENU  -------
1. Display - Console
2. Default Treap - Insertion

3. INSERT an element k
4. SEARCH an element k
5. DELETE an element k
6. INSERT Multiple elements in a Line

p. Print Tree - GraphViz
n. Exit - Bye

 --Choice : 4

<-------->
Enter a Node value to SEARCH : 107

Search Successful for Element (107).
       Its location: 107 , 0xbd1ac0
```

(c) Search Root Node 107: Exist

Figure 4: Treap: Searching: Search 456, 97, 107

3.2 Insert(k)

Test Case 1: Final Output of Insertion and Duplicate Value Detection.

- **Inserting: 107, 101, 98, 111, 102, 104, 119, 109, 99, 97, 116**

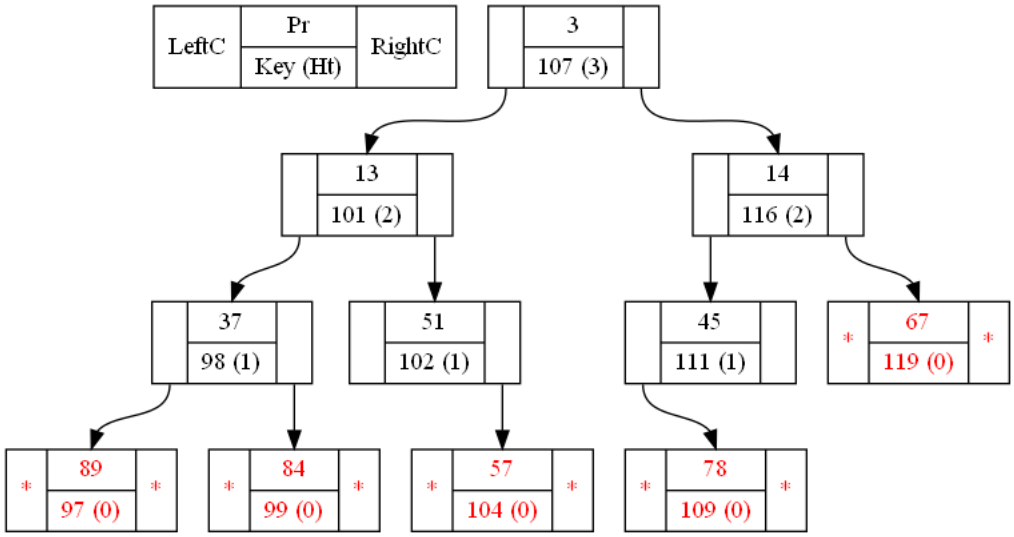


Figure 5: Treap: Insertion: Initial Insertion

- **Inserting: Again 107 (Duplicate)**

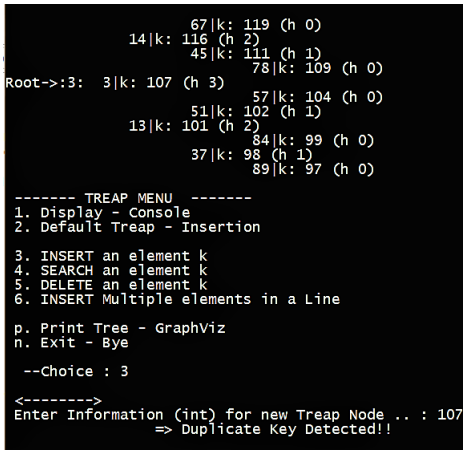


Figure 6: Treap: Insertion: Duplicate Insertion

- **Inserting: Multiple in a line: 107, 101, 98, 111, 102, 104, 119, 109, 99, 97, 116**

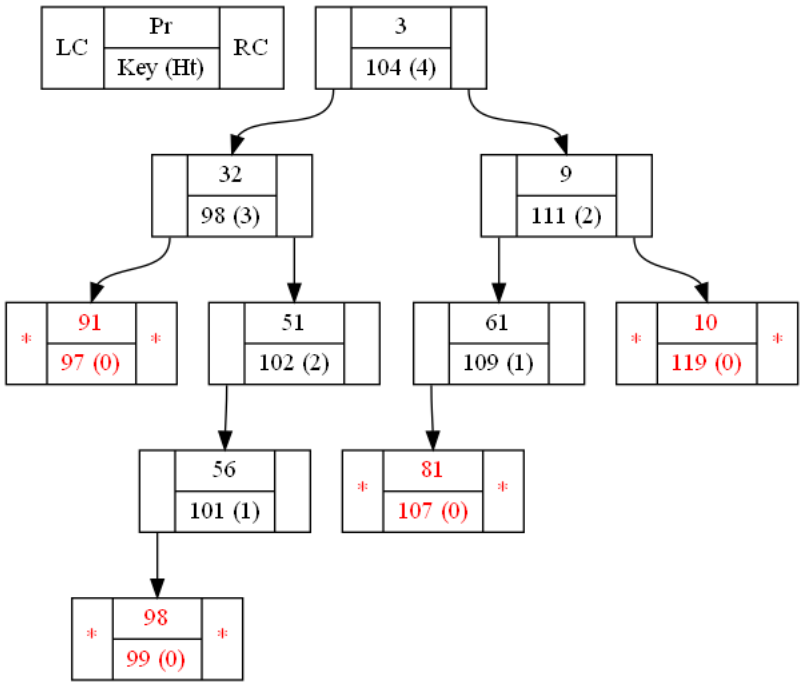
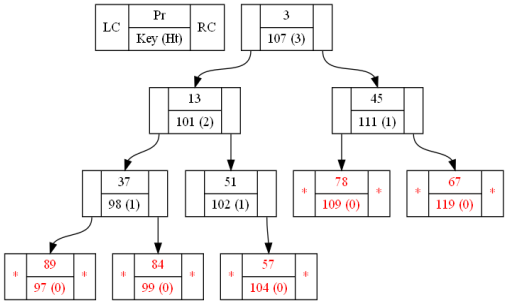


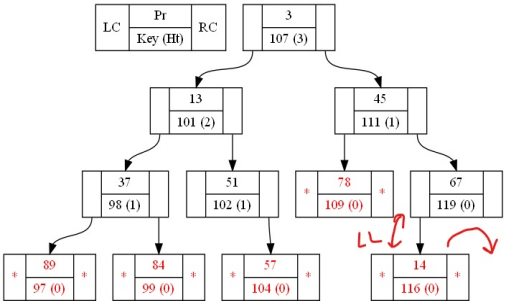
Figure 7: Treap: Insertion: Multiple Insertion, Different priorities

Test Case 2: Checking Rotations

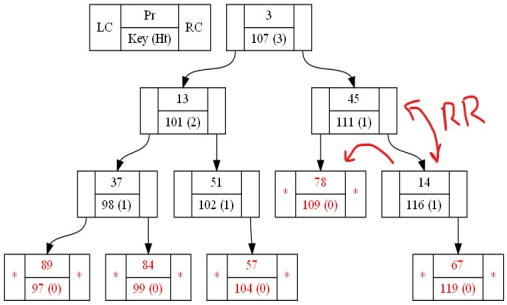
- **Inserting: 14 and then Rotations**



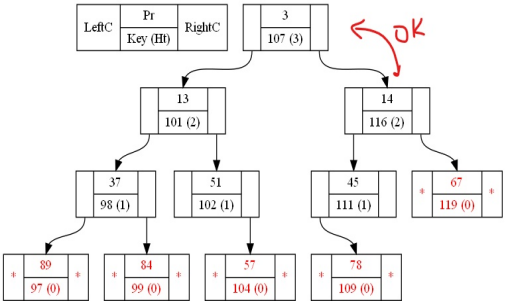
(a) Before inserting 14



(b) Just Inserted 14, LL Imbalance, Right Rotation



(c) RR Imbalance, Left Rotation



(d) Final Tree After Inserting 14

Figure 8: Treap: Rotations in Insert

3.3 Delete(k)

- **Original Tree**

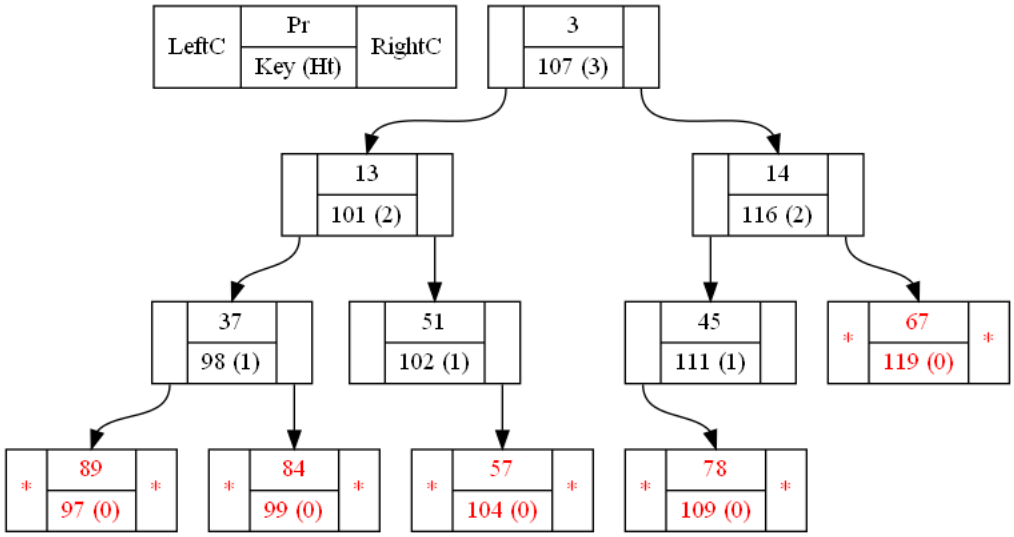


Figure 9: Treap: Deletion: Original Tree

Test Case 1: Empty Tree and Element Does not Exist.

```
------- Treap Menu  -------
1. Display - Console
2. Default Treap - Insertion

3. INSERT an element k
4. SEARCH an element k
5. DELETE an element k
6. INSERT Multiple elements in a Line

p. Print Tree - GraphViz
n. Exit - Bye

 --Choice : 5

--------
Enter a Node value to DELETE : 3
       Tree Empty
```

(a) Deletion on Empty Tree

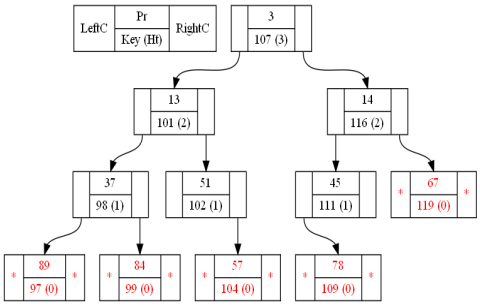
```
Root->:0:  89|k: 97 (h 0)

------- Treap Menu  -------
1. Display - Console
2. Default Treap - Insertion

3. INSERT an element k
4. SEARCH an element k
5. DELETE an element k
6. INSERT Multiple elements in a Line

p. Print Tree - GraphViz
n. Exit - Bye

 --Choice : 5

--------
Enter a Node value to DELETE : 973
       Element not Present in the Treap.
```

(b) Deleting 973, Element does not exist

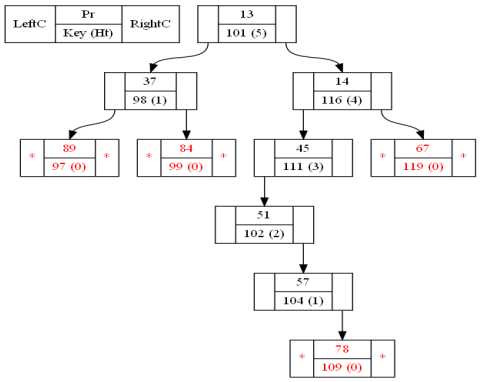
Figure 10: Treap: Deletion: Empty Tree/Element not exist

Test Case 2: Checking Rotations.

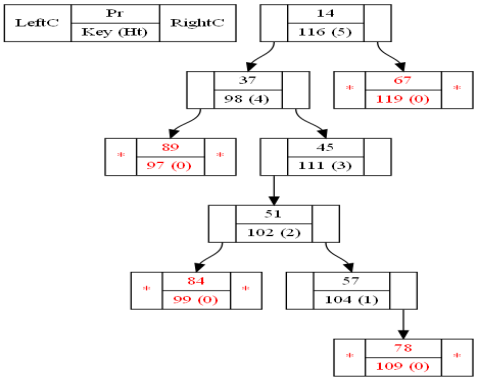
- **Deleting: 107, 101, 116, 98, 111, 102, 104**



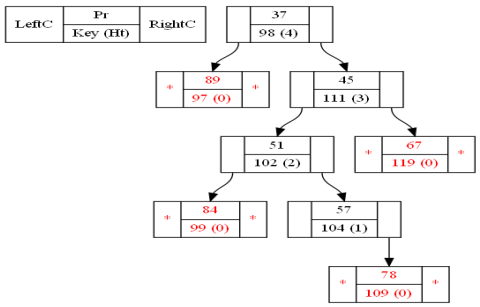
(a) Before Deleting 107



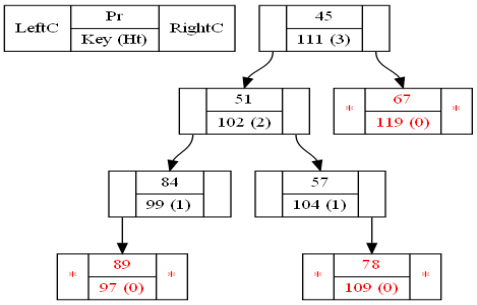
(b) After Deleting 107



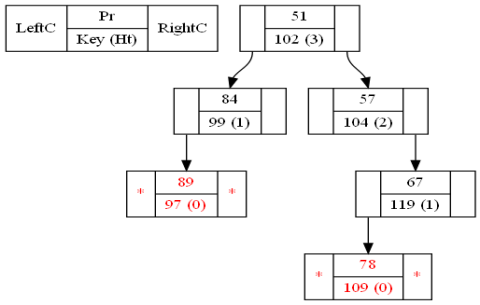
(c) After Deleting 101



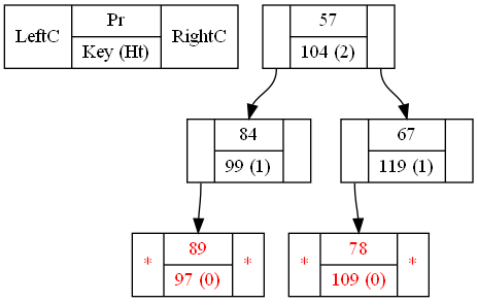
(d) After Deleting 116



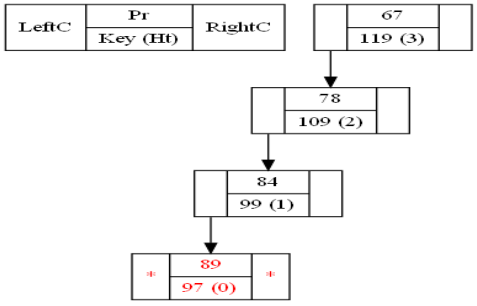
(e) After Deleting 98



(f) After Deleting 111



(g) After Deleting 102



(h) After Deleting 104

Figure 11: Treap: Rotations in Delete

4 PARAMETERS

The parameters calculated for each of the AVL, Threaded BST and Treap for analysis are:

- Total Number of the Nodes (total_nodes)
- Height of the final Tree (height_final)
- Average Height of the Nodes in the final tree (height_avg_node)
- Number of Node Comparisons in insert and delete (comp_final)
- Number of Rotations in insert and delete (rot_final)

4.1 Total Number of Nodes

This parameter is used to store the number of the nodes in the tree at a given point. At the end of the reading each test case file it contains the total nodes present in the final tree.

- **It is calculated** whenever we successfully insert and delete the node from the tree and therefore incrementing or decrementing respectively.
- **This is used** to make sure all the three trees have the same number of nodes inserted. It is also used to calculated average height of the nodes.

4.2 Height of the Tree

This parameter is used to store the height of the tree at a given point. At the end of the reading each test case file it contains the final height of the tree. The height of a node is the number of edges on the longest path from the node to a leaf. A leaf node will have a height of 0.

- **It is calculated** (For AVL and Treap) whenever we successfully insert and delete the node from the tree and rotation changes the height of the tree. (For BST) **It is calculated** at the end of the final tree using recursive function that calculate height of each node and sum of height of each node.
- **This is used** as Height of the tree is good parameter to evaluate whether the tree is balanced or unbalanced. Balance tree has worst case time complexity $O(\log n)$ while unbalanced may have $O(n)$.

4.3 Average Height of the Nodes

This parameter is used to store the average height of the node at the end of the reading each test case file.

- **It is calculated** at the end of the final tree using recursive function (calculate_sum_of_nodes_heights()) that calculate height of each node and sum of height of each node. After that dividing it with total number of nodes present in the tree gives the average height of the node of the tree.
  $H_{avg} = \frac{1}{N} \sum_{i=1}^{N} h_i$ where N is the total number of nodes, h_i is the height of node 'i'.

4.4 Number of Node Comparisons

This parameter is used to store the number of comparisons done at the time of insert and delete operations.

- **It is calculated** whenever we perform insert and delete of a node and travel along a path from root to leaf in order to search key value of that node. Therefore it is incremented appropriately on node comparison at each subtree.
- **This is used** as number of comparisons give us good idea about how much calculations are done to search a node key in the tree. Low comparisons are good for dictionary data structure.

4.5 Number of Rotations

This parameter is used to store the number of rotations done at the time of insert and delete operations. Double rotation in AVL tree are counted as two single rotations. For BST, as rotations are not performed therefore not calculated for it.

- **It is calculated** whenever we perform insert and delete of a node and after that rotations are required to balance the tree. Rotation is required in AVL in case of imbalance of the tree and in Treap for violating heap property. Rotations are not required for BST therefore it is not calculated.
- **This is used** as number of rotations give us good idea about how much calculations are done to maintain the balance or heap property of the tree. Balance tree give us good average time complexity in the operations.

5 TEST FILES

File and functions used for generating test case files, reading test case files and generating parameters output are explained in section 1.8. User can generate Default Test Case Files as well as generate files according to his need of number of operations and insert frequency. User can evaluate Default Test Case Files as well as other files provided.

5.1 Test Files Generation

- **Default Test Files** are generated and named "nop_+*number of operations*+.txt". They are stored in the folder 'input_test_files' present in the same directory. Function void GenerateDefaultTestCases().
- **Maximum Number of Operations** can be changed using global variable 'max_op' which changes the limit for the maximum number of the operations. Each file is generated from initial operation 100 to maximum operations, incrementing each file operation with 500 operations.
- **Default Number of Files** generated depend on the global variable 'max_op'.
  Number of Files = (MaxOp - 500)/500 + 1 + 1 (for 100 op).
- **Insert:Delete Ratio**. The number of the insert operation frequency in any file at a time is randomly selected from the global array 'ins_ratio'. For Default Test Files generation Insert Ratio used are 0.5, 0.6, 0.7, 0.8. While for generating manual test files, there is provision to provide insert frequency.
- **Manual Test Files**. Multiple Manual Test Files can be generated from the console and it will generate all the files with provided number of the operations 'nop' and insert frequency 'insert_ratio_value' for each file separated with comma. Files are named with "top_+*number of operations*+.txt" and stored in the folder 'input_test_files' present in the main directory. Function void GenerateTestCases(int nop, float insert_ratio_value).

5.2 Evaluations of Parameters

- **After reading each of the test case files,** in a resultant tree final value of the parameters such as average height of the node are calculated by calling function "write_parameters()" by each AVL, BST and Treap object. It finalises the value of the each parameters used with necessary error handling.
- **Default Test Files** are evaluated by calling function "void ReadDefaultTestCases()".
  Parameters of each file read are stored in the file named "analysis_default_.csv" and stored in the folder 'output_analysis_files' present in the main directory.
- **User Provided Test Files** are evaluated by calling function "void ReadTestCases()".
  To read user provided test case files, file names should be stored in the file named '_read_user_test_files.txt' with first line being the number of the files present and each next line will be full name of the file. Files should be present in the same folder.
  Parameters of each file read are stored in the file named "t_+*num of files read*+ _user_test_file_analysis.csv" and stored in the folder 'output_analysis_files' present in the main directory.

5.3 Randomness of Operations

- The program as used the standard random generator mt19937 "mersenne_twister_engine" provided by CPP library "random". The Mersenne Twister is a pseudorandom number generator (PRNG). It is by far the most widely used general-purpose PRNG. Its name derives from the fact that its period length is chosen to be a Mersenne prime.
- The generator is seeded with system clock provided by CPP library 'chrono'. The seed value is the amount of time between this time_point and the clock's epoch.
- Discrete Uniform Distribution (unifor_int_distribution) provided by CPP library "random" is used to make choice

- for selecting the insert ratio from the global array 'ins_ratio' at the beginning of reading each file.
  Range [0, size_of_array -1].
  uniform_int_distribution<int> my_random_ratio(0, 3);
- for choice of Insert or Delete Operation to be present at a given time in the Test File. If number of insert operation become greater than the insert ratio provided then we do delete operation only and vice versa.
  Range [0,1].
  uniform_int_distribution<int> my_random_op(0, 1);
- for selection of integer value for the insert operation and then write in the test case file.
  Range [1, Number_of_operations].
  uniform_int_distribution<int> my_random_ins(1, nop);
- for selection of the element for delete operation from the array of inserted elements.
  Range [0, Number_of_inserted_elements].
  uniform_int_distribution<int> my_random_del(0, nop_ins-1);
- for selection of priority for the insertion in the Treap while reading test case file.
  Range [0,Number_of_operations_in_file].
  uniform_int_distribution<int> my_random_priority(1, read_nop);

Discrete Uniform Distribution refers to the distribution with constant probability for discrete values over a range and zero probability outside the range. The probability density function P(x) for uniform discrete distribution in interval [a, b] is constant for discrete values in the range [a, b] and zero otherwise.

6 ANALYSIS OF AVL, BST and TREAP

The analysis is done by generating 201 files from incrementing operations by 500 from 100, 500 to 1 Lakh operations. Each operation consist of randomly selected insert frequency from values (0.5,0.6,0.7,0.8).

6.1 Insert:Delete Ratio vs Operations

- The figure (12) shows the count of particular insert:delete ratio used, and its percentage of usage with respect to total files in analysis. $41 + 49 + 62 + 49 = 201$.

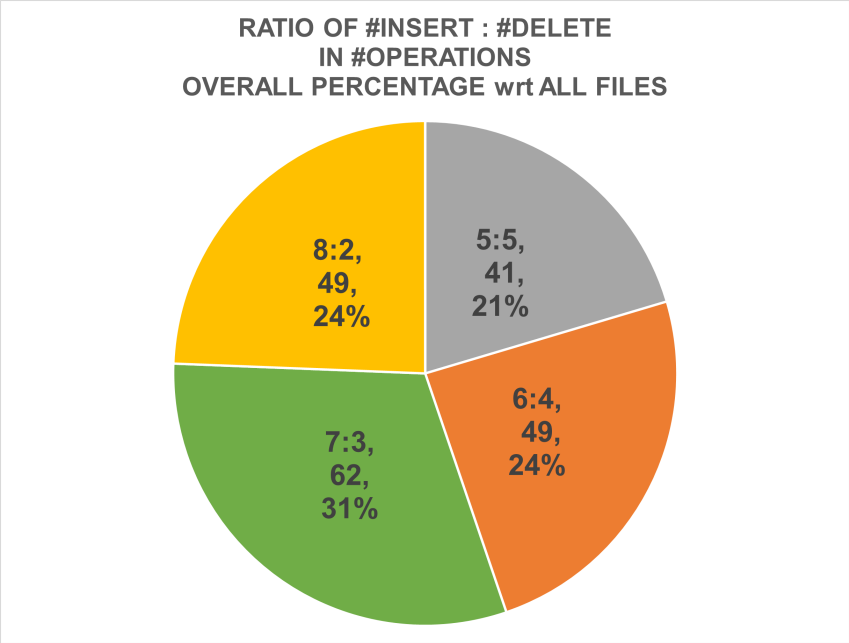


Figure 12: Analysis: Insert:Delete Ratio Overall Count

- This figure (13) shows the percentage of insert:delete ratio with respect to operations, that is what ratio is used at which operation count.

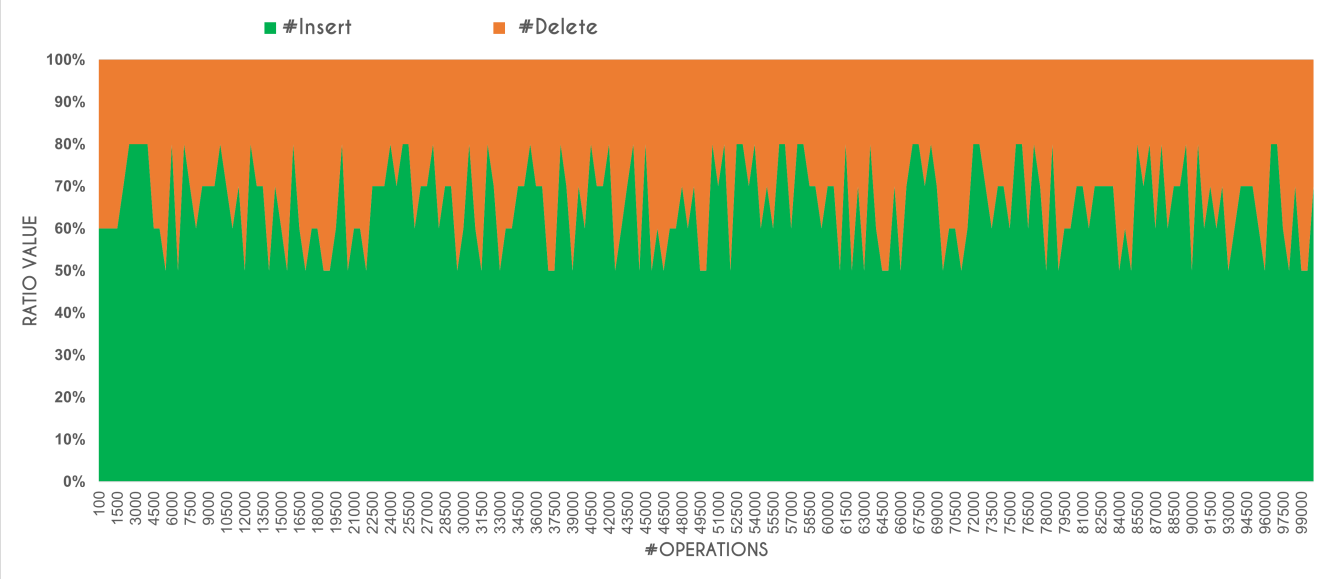


Figure 13: Analysis: Insert:Delete Ratio used with Operations

■ **Analysis:**

1. The percentage of insert:delete ratio is uniformly distributed over all the operations.

6.2 Number of Nodes vs Operations

- The figure (14) shows the number of nodes with respect to the operations.

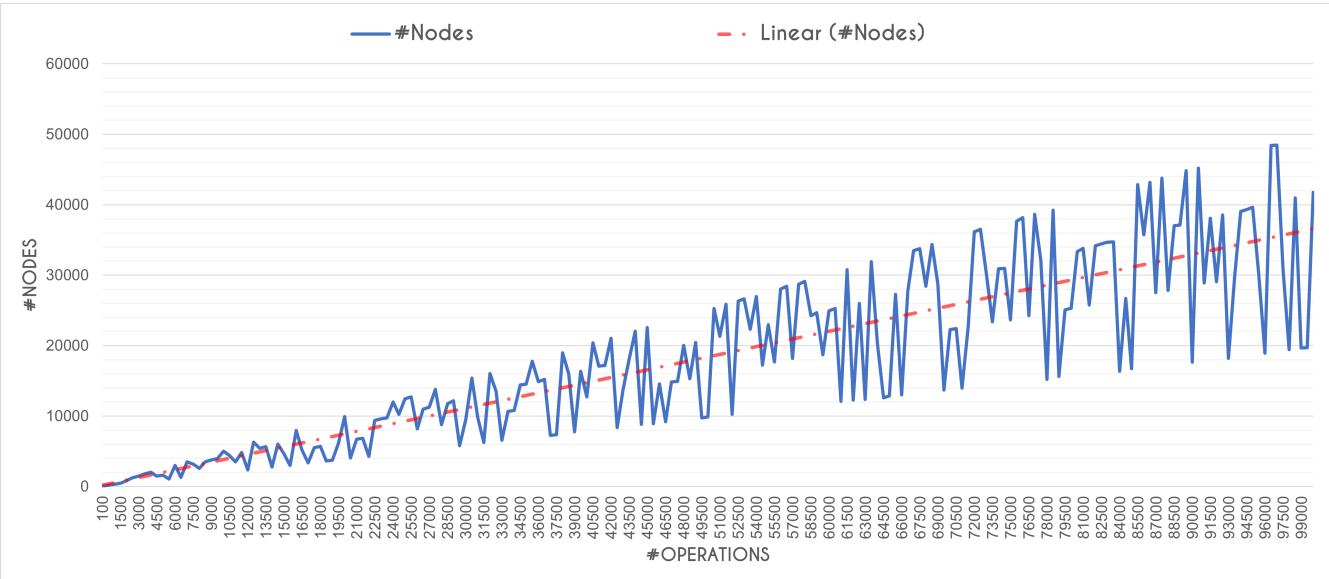


Figure 14: Analysis: Number of Nodes with Operations

■ **Analysis:**

1. We can see from the figure (13) and figure (14) that *increasing the insert ratio increases the number of nodes* in the tree. Whenever the insert ratio decreases from previously increased values, then number of nodes also dropped.

2. Number of nodes follows a *linear increase with operations* as shown by red dotted line in the graph (14).

6.3 Height of Tree vs Operations

- The figure (15) shows the final height of the AVL, BST, Treap with respect to the small number of the operations.

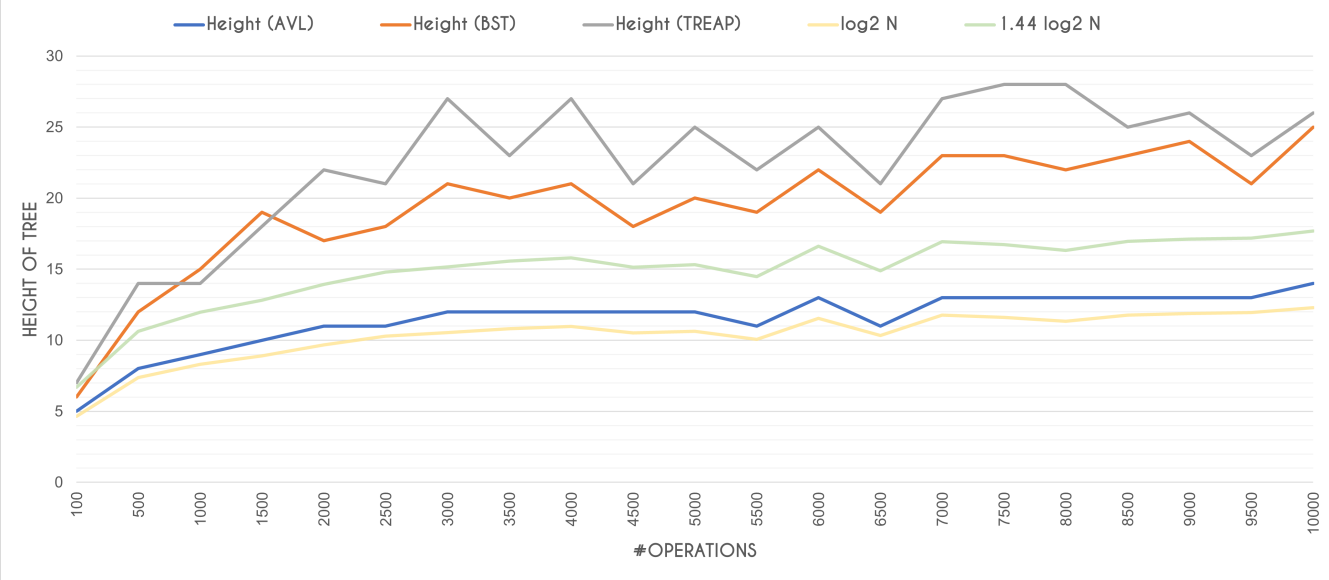


Figure 15: Analysis: Height of Tree with Max 10k Operations

- The figure (16) shows the graph of the final height of the AVL, BST, Treap with respect to the large operations.

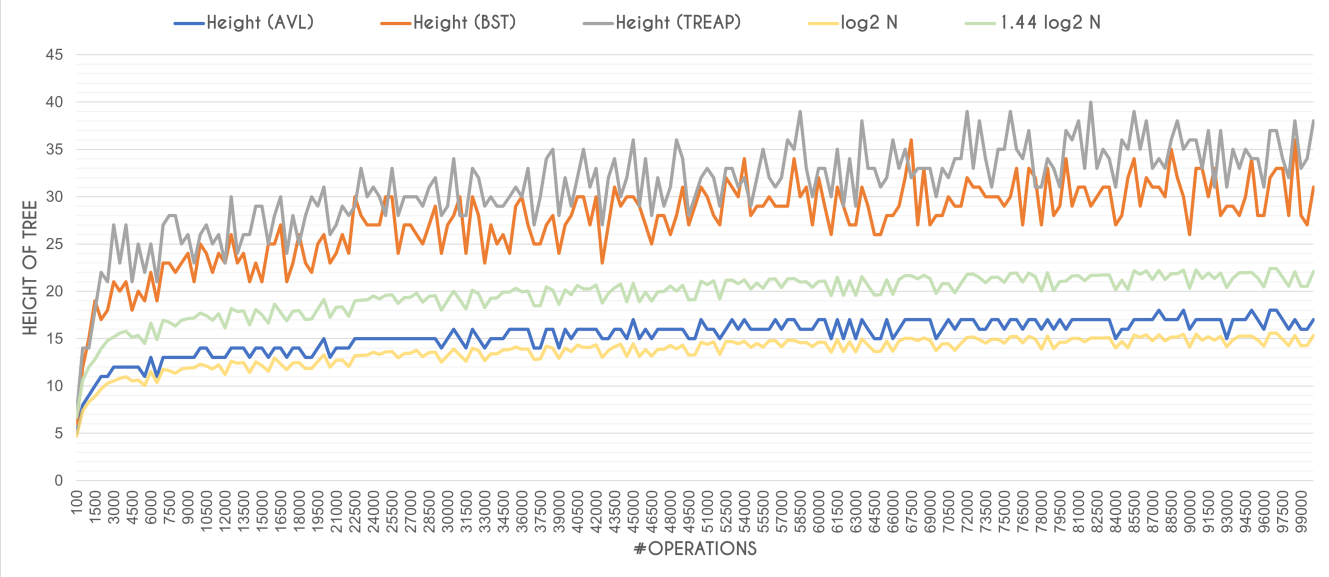


Figure 16: Analysis: Height of Tree with Operations

- The figure (17) shows the logarithmic trend of the final height of the AVL, BST, Treap with respect to the Number of the Nodes.

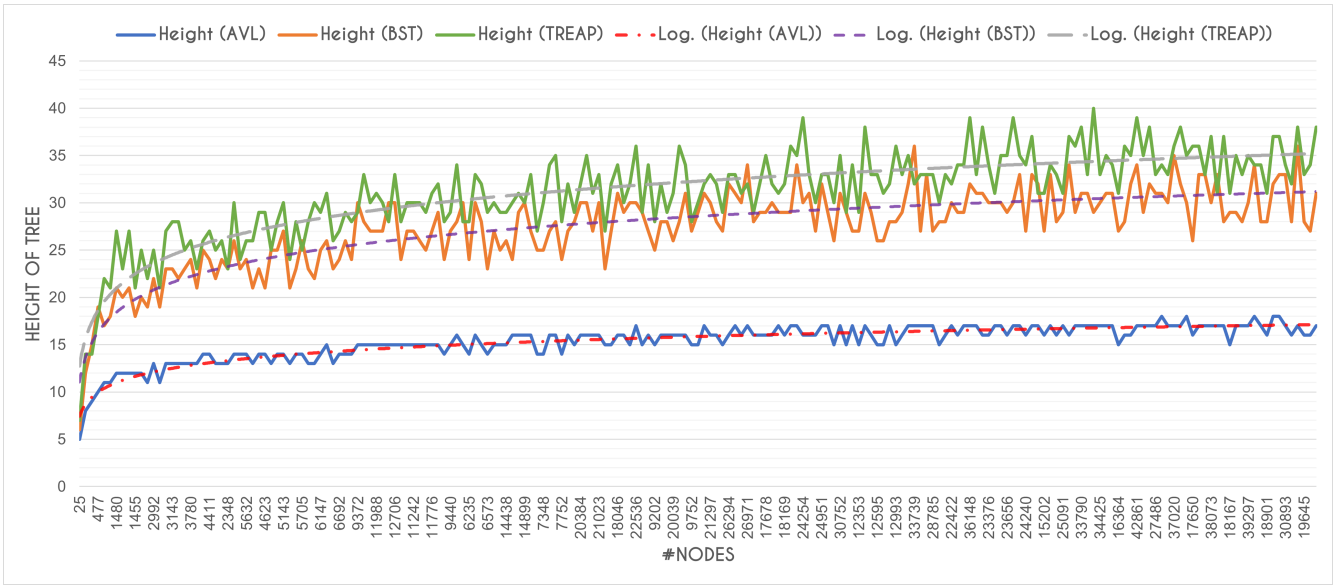


Figure 17: Analysis: Logarithmic Trend of The Height of Tree

■ **Analysis:**

1. We can see from the figure (16) that **Height of the AVL Tree** is bounded within theoretical range of minimum $O(\log N)$ and maximum of $O(1.44 \times \log N)$ where N is the number of nodes. The AVL Tree is balanced.
2. The figure (16) shows that **Height of the BST and Treap** being greater than maximum of $O(1.44 \times \log N)$ but both tree height are comparable to each other, with height of the Treap slightly more than the BST. They are not as much balanced as compared to AVL Tree, but with random insertion and deletion they are not unbalanced too.
3. The figure (17) shows the *logarithmic trend of height* of the AVL(red dotted line), BST(purple dotted line), Treap(grey dotted line) with respect to number of nodes.

6.4 Average Height of Node vs Operations

- The figure (18) shows the average height of the node with respect to small number of operations for AVL, BST, Treap.

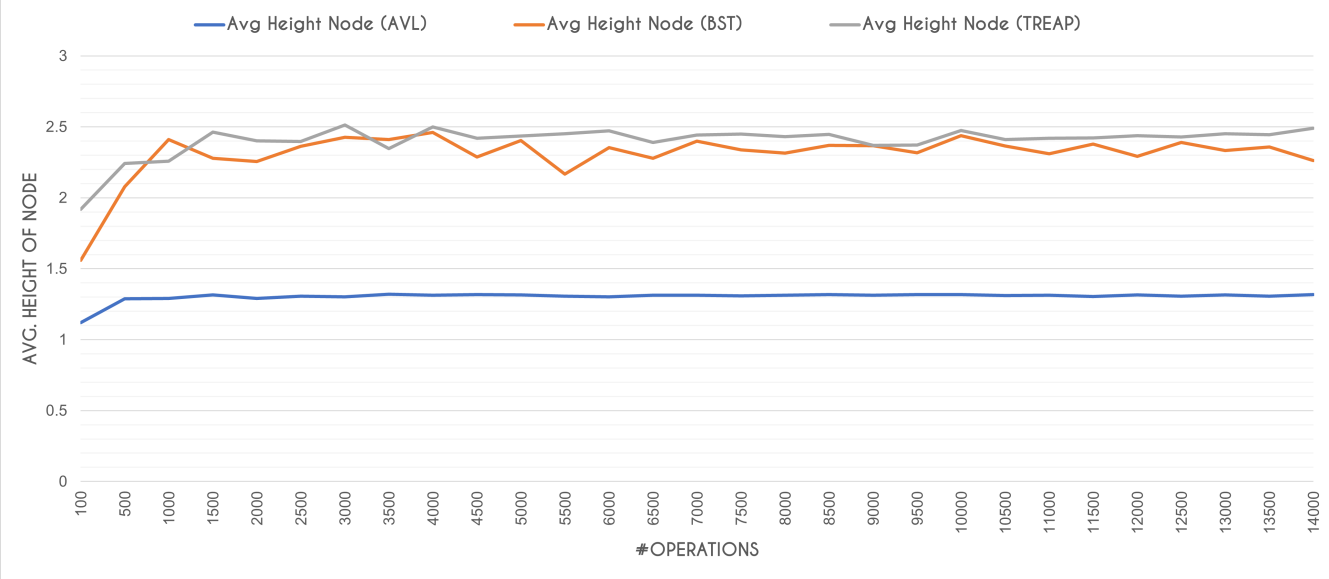


Figure 18: Analysis: Average Height of Node with Max 14K Operations

- The figure (19) shows the average height of the node with respect to large number of operations for AVL, BST, Treap.

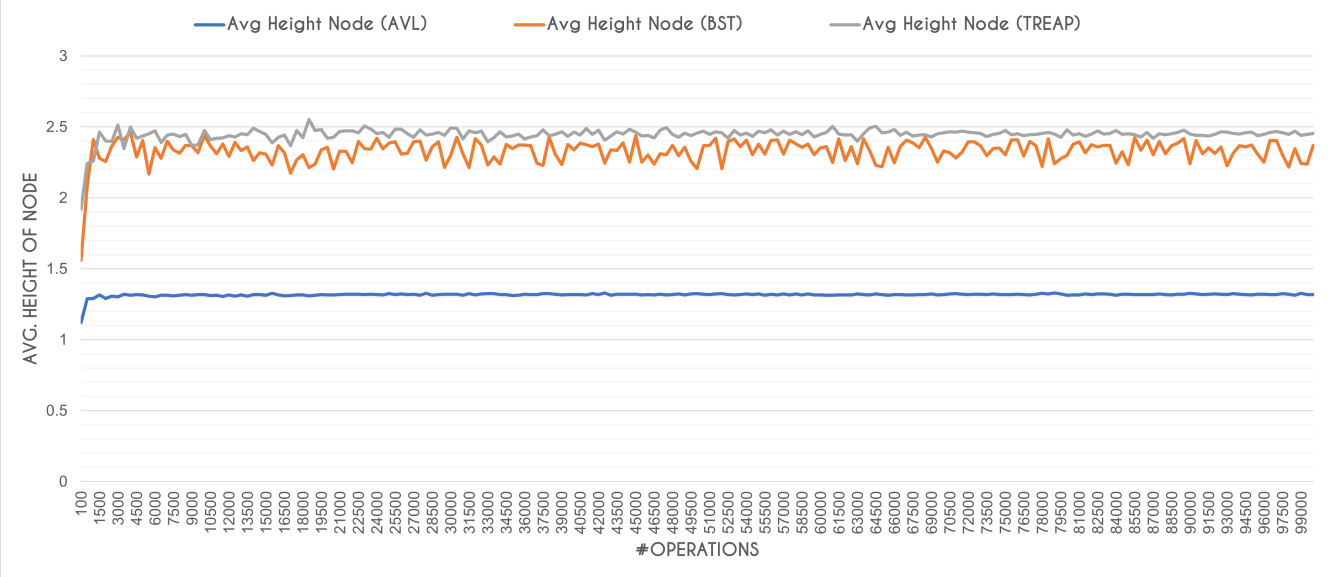


Figure 19: Analysis: Average Height of Node with Operations

■ **Analysis:**

1. The figure (18) and (19) shows that average height of the node increases with increase in the number of operations up to limit of max 2000 operations.
2. After 2000 operations, **Average Height of Node for AVL** converges and remains almost same for all the operations.
3. After 2000 operations, **Average Height of Node for BST and Treap** also converges and remains almost within same bound.

4. Average Height of Node of AVL is smaller than BST and Treap, while between BST and Treap, Average Height of Node for Treap is slightly more than BST but both comparable.

6.5 Number of Comparisons vs Operations

- The figure (20) shows the number of nodes comparisons for AVL, BST, Treap during insertion and deletion with respect to the small number of operations.

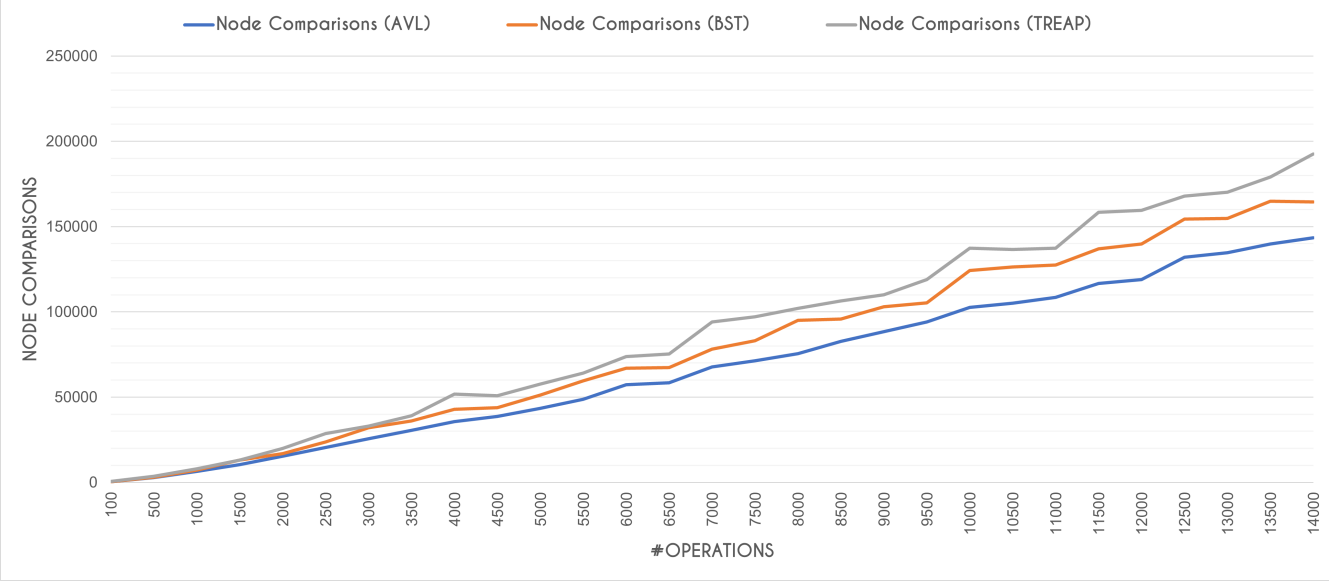


Figure 20: Analysis: Number of Node Comparisons with max 14K Operations

- The figure (21) shows the number of nodes comparisons for AVL, BST, Treap during insertion and deletion with respect to the large number of operations.

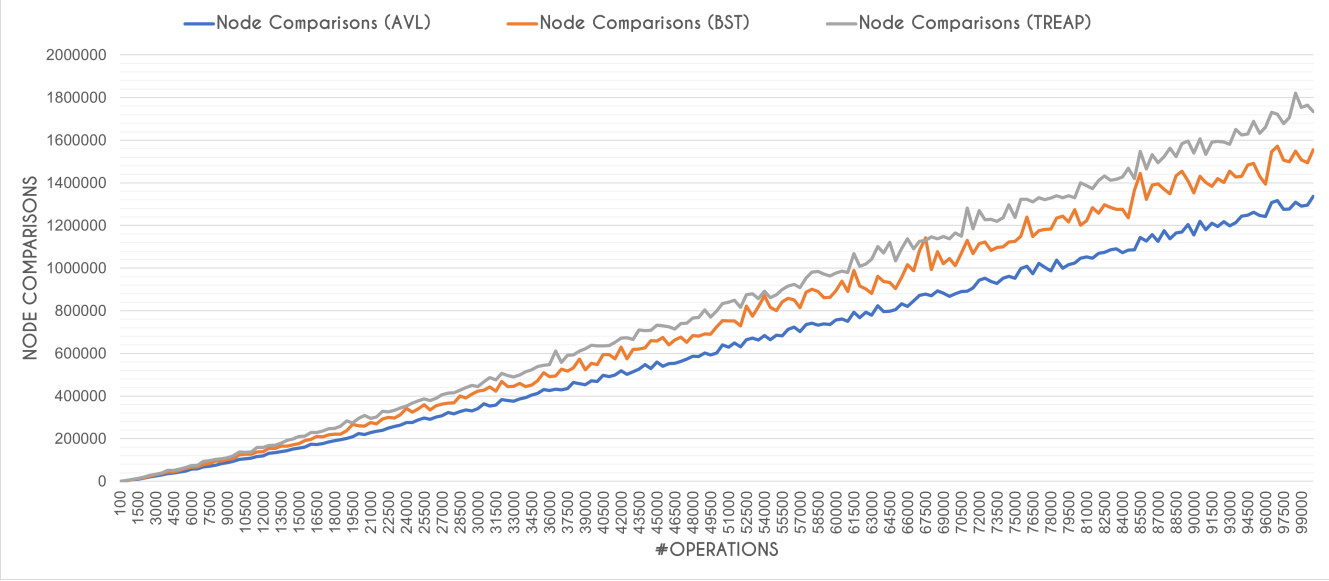


Figure 21: Analysis: Number of Node Comparisons with Operations

- The figure (22) shows the linear increasing trend of the number of nodes comparisons for AVL, BST, Treap with respect to operations.

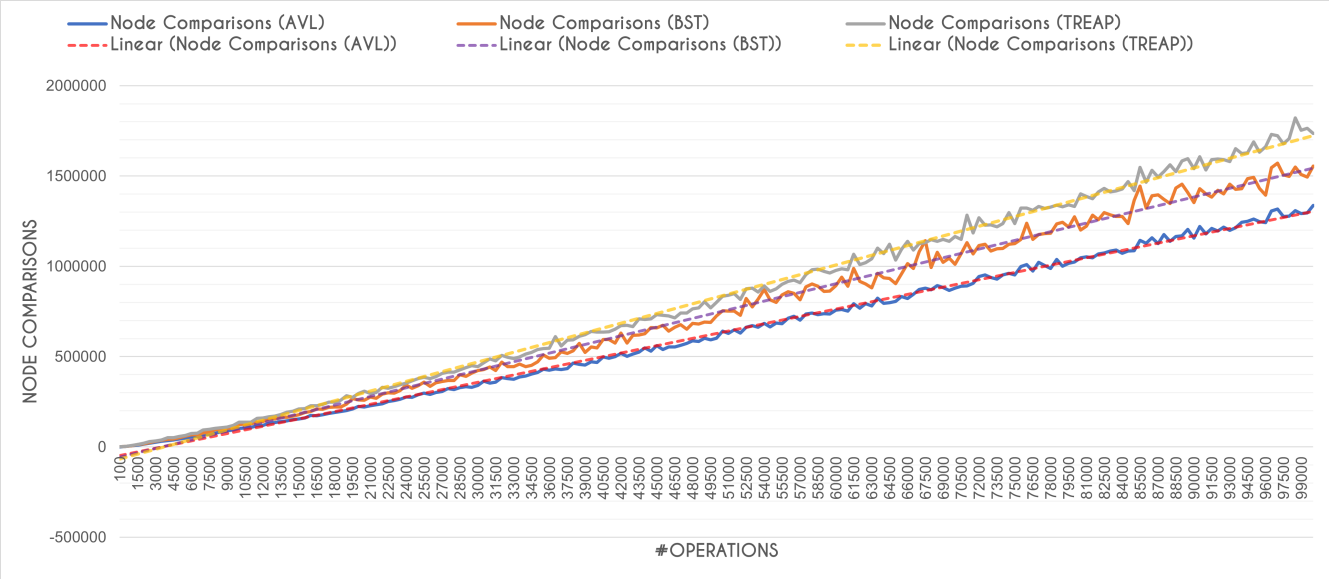


Figure 22: Analysis: Linear Trend of the Number of the Comparisons.

■ **Analysis:**

1. The figure (22) shows the **linear increasing trend** for the node comparisons with increase in number of operations for AVL(red dotted line), BST(purple dotted line), Treap(yellow dotted line)
2. **Node Comparisons of AVL is smaller** than BST and Treap, while between BST and Treap, Node Comparisons for Treap is more than BST with gap increasing with increasing number of operations.

6.6 Number of Rotations vs Operations

- The figure (23) shows the number of rotations in AVL and Treap during insertion and deletion with respect to the small number of the operations.

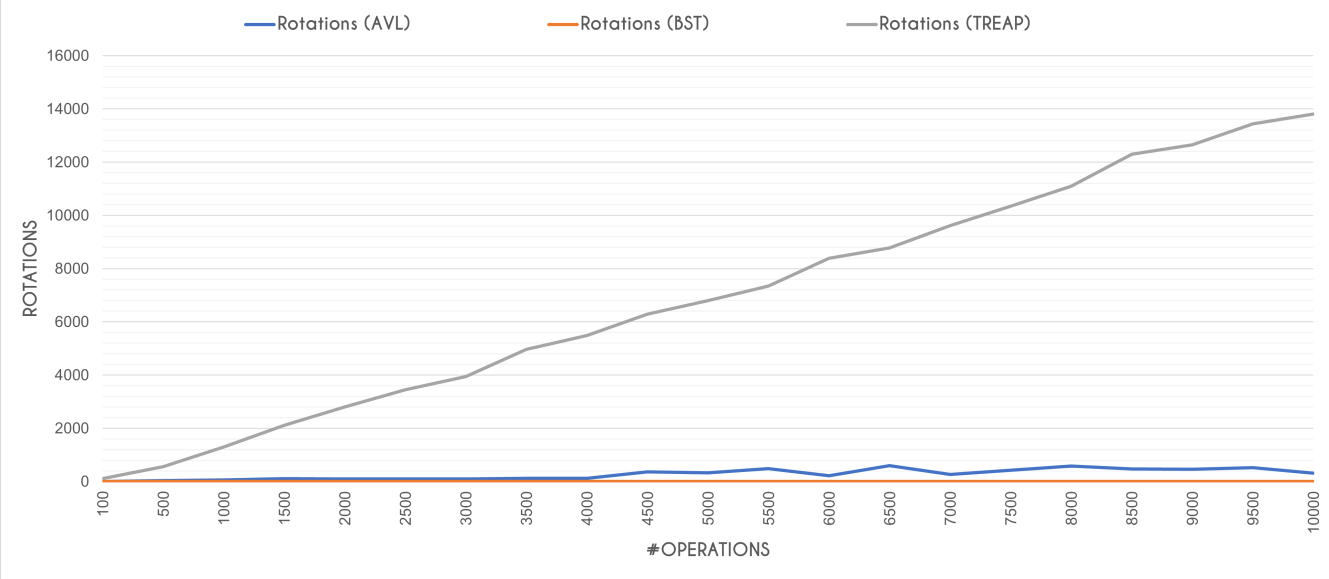


Figure 23: Analysis: Number of Rotations with max 10K Operations

- The figure (24) shows the number of rotations performed in AVL and Treap during insertion and deletion with respect to the large number of the operations.

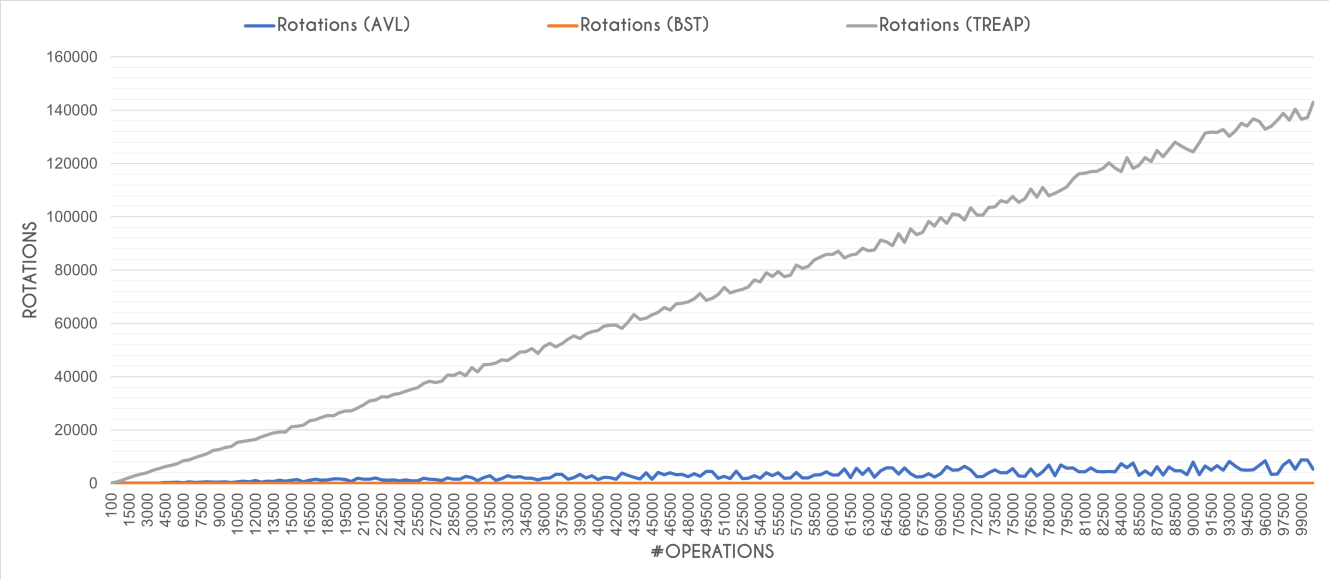


Figure 24: Analysis: Number of Rotations with Operations

■ **Analysis:**

1. The number of the rotations are clearly **less for the AVL** as compared to Treap. Rotations in AVL increases very slowly even for large number of operations.
2. Their is *linear increase in the number of rotations for Treap* and gap between AVL rotations and Treap increases with number of operations.

6.7 Conclusions

Final conclusions after the analysis are:

- For all the parameters (Height, Avg Height, Node Comparisons, Rotations) **performance of the AVL is quite better** (with large gap) than the BST and Treap. With large number of operations height, average height of the node, node comparisons and rotations for AVL are less.
- **Height of the AVL is between its theoretical bounds** (Figure 16). *BST and Treap are not completely balanced but Height of them follows a logarithmic trend* (Figure 17).
- For the parameters (Height, Avg Height, Node Comparisons) **performance of the BST is slightly better** (with smaller gap) as compared to Treap. While in some cases, they are almost comparable due to randomized nature of insertion and deletion with random priority.
- For the parameter (Rotations) **performance of the AVL is quite better** (with large gap) as compared to Treap. Treap rotations are more as we have to repeatedly rotate the node to make it the leaf.
- For the Parameter (Node Comparisons) AVL, BST and Treap *follows a Linear increase trend* with respect to increase in operations (Figure 22).