Dalhousie University
Master Of Applied Computer Science
CSCI-6057 - Advanced Data Structures
Responsible: Meng He

Optimizing search operation worst time complexity of Splay Tree:

Using AVL Tree and Splay Tree Rotation Properties

Thursday 14th April, 2022
Manraj Singh - B00877934

Contents

1 Git Repository: **1**

2 Abstract: **1**

3 Introduction: **1**

3.1 Binary Search Tree: 1

3.1.1 Children per node: 1

3.1.2 Nodes distribution: 2

3.1.3 Acyclic Dependency: 2

3.1.4 Sorted Order: 2

3.2 AVL Tree: 2

3.2.1 Left-Left Unbalanced Node: 2

3.2.2 Right-Right Unbalanced Node: 3

3.2.3 Right-Left Unbalanced Node: 3

3.2.4 Left-Right Unbalanced Node: 4

3.3 Splay Tree: 4

3.3.1 Zig Splay: 5

3.3.2 Zig-Zig Splay: 5

3.3.3 Zig-Zag Splay: 5

4 Problem Discussion: **7**

5 Research Paper Survey: **7**

6 Further Improvement: **9**

7 Comparison: **11**

8 Conclusion: **19**

List of Figures

1 Binary Tree nodes distribution 2

2 Balancing Left-Left Unbalanced node with right rotation: 3

3 Balancing Right-Right Unbalanced node with left rotation: 3

4 Balancing Right-Left Unbalanced node with right and left rotation: 4

5 Balancing Left-Right Unbalanced node with left and right rotation: 4

6 Zig Splay: 5

7 Zig-Zig Splay: 6

8 Zig-Zag Splay: 6
9 Left-Skewed Splay Tree after searching elements in strictly increasing order: . 7
10 Converting Skewed tree into balanced binary search tree: 8
11 Dynamic Programming – inside node variables: 8
12 Formation of left skewed tree and while searching nodes in a sorted order . . . 10
13 Figure 13: Balancing the left skewed tree using rotation operations like AVL tree 10
14 Balancing Left-Left unbalanced node (7) with right rotation balancing technique: 10
15 Balanced Splay Tree on the left sub tree of the root in strictly ascending search operations: 11
16 Comparison Parameters of Binary Search Tree: 12
17 Comparison Parameters of AVL Tree: 13
18 Comparison Parameters of Splay Tree: 13
19 Comparison Parameters of Hybrid Splay Tree: 13
20 Right skewed Binary Search tree after inserting elements in sorted order: 14
21 Balanced AVL tree after all the search operations in sorted order: 14
22 Left Skewed Splay Tree after search operations in sorted order: 14
23 Balanced Hybrid Splay Tree after search operations in sorted order: 15

List of Tables

1 Comparison Parameters in Nano Seconds with no. of elements = 10: 15
2 Comparison Parameters in Nano Seconds with no. of elements = 100 16
3 Comparison Parameters in Nano Seconds with no. of elements = 1000 16
4 Comparison Parameters in Nano Seconds with no. of elements = 3000 17
5 Comparison Parameters in Nano Seconds with no. of elements = 5000 17
6 Comparison Parameters in Nano Seconds with no. of elements = 50000 18

1 Git Repository:

`https://git.cs.dal.ca/manraj/hybridspaytree_ads_6057`

2 Abstract:

Splay tree is closely related to the binary search tree with a few changes in the insert and search operations. The worst time complexity of insert or and search operation in a Splay tree is same as that of a binary search tree, that is O(n). AVL tree is a data structure that maintains a balance of weights on every node. This helps in maintain the height of the tree to be less than or equals to 1.44*$O(\log_2(n))$. So, the worst time complexity to search and insert an element to the AVL tree becomes $O(\log_2(n))$ which is way better than the worst time complexity of a splay and a binary search tree.

The property of Splay Tree of moving the recently searched element to the top of the tree using rotation operations like the AVL tree, makes the amortised complexity of search and insert of elements less than $O(\log_2(n))$. Even though the most expensive operation in a splay tree is more expensive than any operation in an AVL tree but it is compensated by the other operations which are way less expensive than the AVL tree.

3 Introduction:

Main idea of this project is to reduce the worst time complexity of a Splay tree search operations using AVL tree properties of balancing the nodes and reducing the max height the Splay Tree can reach. Splay tree is more related to Binary search tree with the properties of moving the most recently touched node to the top of the tree. This helps in reducing the time complexity of finding the same element by increasing its probability. Before moving to the solution, we will have to understand the working on different binary trees that we will use to improve the splay tree.

3.1 Binary Search Tree:

Binary search tree was introduced to ease the search and insert operations on a large dataset. The properties of binary search trees are:

3.1.1 Children per node:

Each node can have minimum of zero and maximum of 2 child nodes.

3.1.2 Nodes distribution:

Left child nodes to the parent nodes will always be smaller than the parent node and the right child nodes will always be bigger in value.

3.1.3 Acyclic Dependency:

There cannot be any cyclic dependency, which means that any node cannot have any of the ancestor nodes as its child.

3.1.4 Sorted Order:

In-order traverser gives the sorted order of the elements.

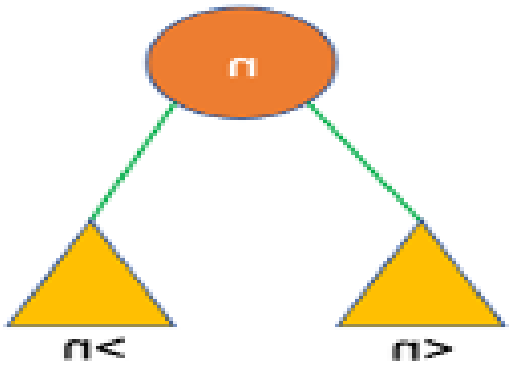

Figure 1: Binary Tree nodes distribution

3.2 AVL Tree:

AVL tree is an extension of Binary Search Tree which adheres all the properties of BST. It has some properties of its own which maintains the height of the tree and makes it more balanced.

Unbalanced tree is when there exists a node where mod of the difference between max height of left children and max height of right children is greater than 2. To maintain the balance these unbalanced nodes are rotated in such a way that the weight gets balanced, and the difference reduce to less than 2. To balance these nodes, we use 4 rotation methods which are explained figure[2] with examples:

3.2.1 Left-Left Unbalanced Node:

As shown in the figure 2 we can see node 7 have max left height as 3 and max right height as 1. Moreover, the left child also has more weight on the left side. This makes the node 7 as left-left unbalanced and can be balanced using one right rotation as shown in figure 2.

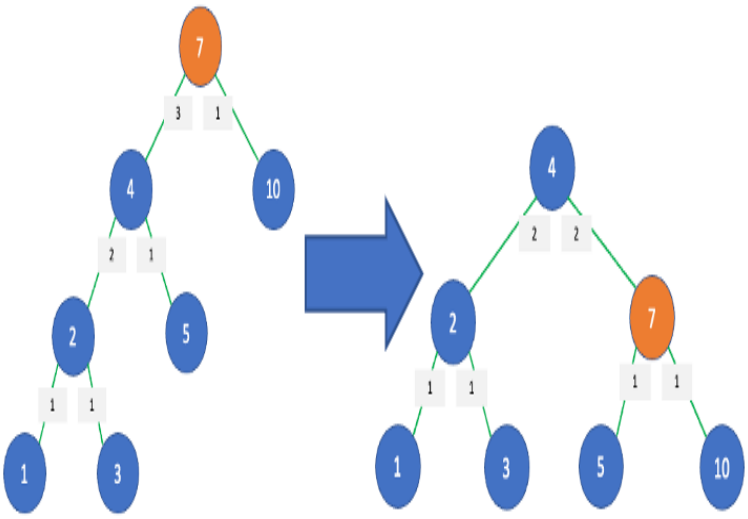

Figure 2: Balancing Left-Left Unbalanced node with right rotation:

3.2.2 Right-Right Unbalanced Node:

As shown in figure 3, right-right unbalanced node has more weight on right that left and same pattern is followed in the right child, which makes the node 7 as right-right unbalanced. This can be balanced using one single left rotation.

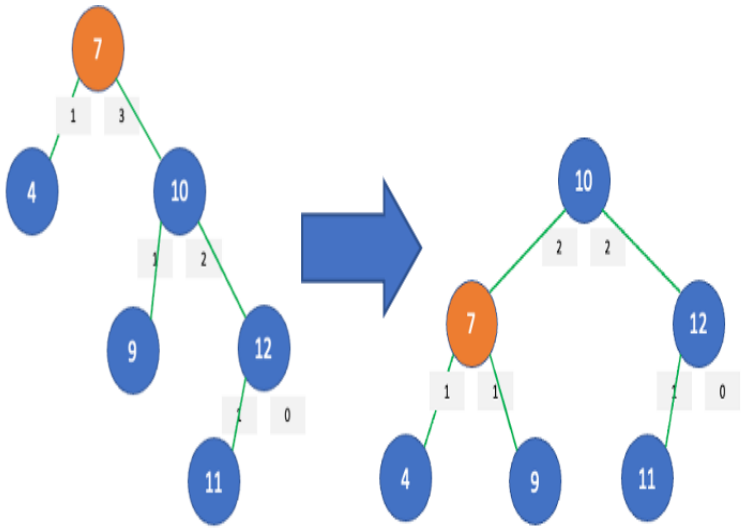

Figure 3: Balancing Right-Right Unbalanced node with left rotation:

3.2.3 Right-Left Unbalanced Node:

As shown in figure 4, the main unbalanced node has more weight on right and the right child node has more weight on left which makes it Right-Left unbalanced node (7). To balance it first we need to rotate the right node of the unbalanced node with right rotation and then perform left rotation on the main unbalanced node (7) as shown figure 4:

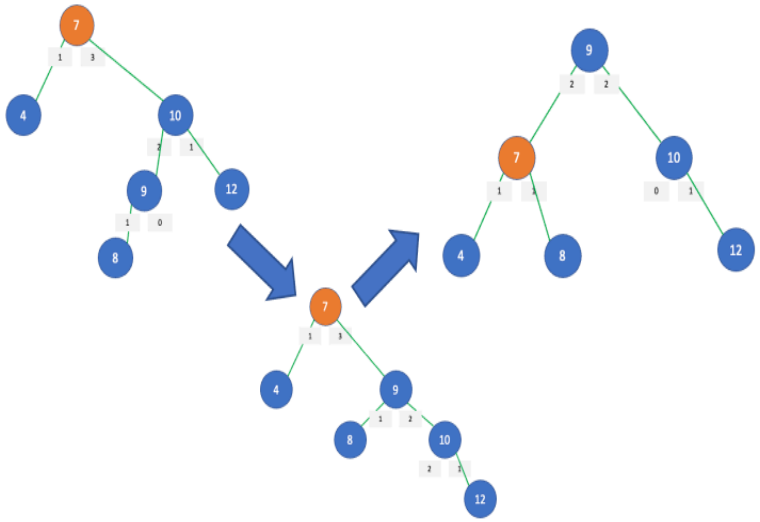

Figure 4: Balancing Right-Left Unbalanced node with right and left rotation:

3.2.4 Left-Right Unbalanced Node:

As shown in figure 5, the main unbalanced node has more weight on left and the left child node has more weight on right which makes it Left-Right unbalanced node (7). To balance it first we need to rotate the left node of the unbalanced node with left rotation and then perform right rotation on the main unbalanced node (7) as shown figure[5]:

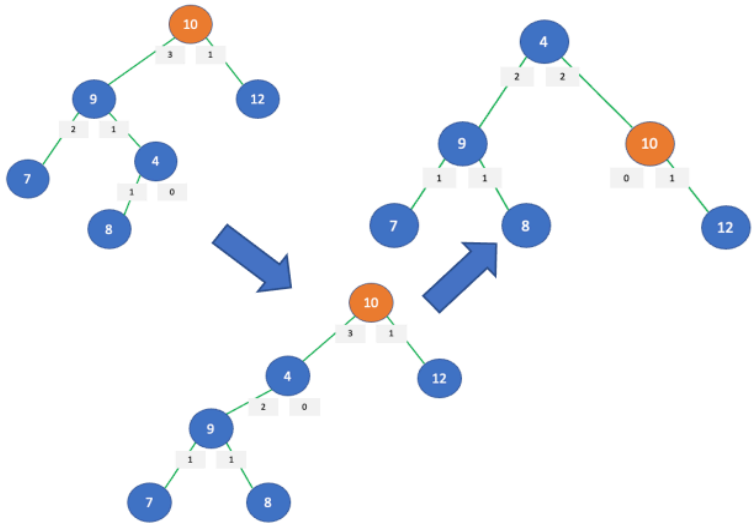

Figure 5: Balancing Left-Right Unbalanced node with left and right rotation:

3.3 Splay Tree:

Splay Trees are an extension of Binary Search tree. It's the most used basic data structures in last 30 years. It's the fastest binary search tree and used in many software like GCC compiler, GNU C++ library, malloc for unix kernals, etc. The main idea of a splay tree is to make the

recently visited nodes of the tree more easily accessible. To achieve this, after every search or insert operation the target node is moved to the root of the tree using 3 different splay operations as discussed below. This makes the time complexity of re-searching the same node, which is recently searched for or inserted, as constant. These 3 splaying techniques are as shown figure[6]:

3.3.1 Zig Splay:

This is the rotation when the searched node is direct child of the root node. Only one rotation is required in this case to move the node to the root. In the figure[6] example the searched node is “c”.

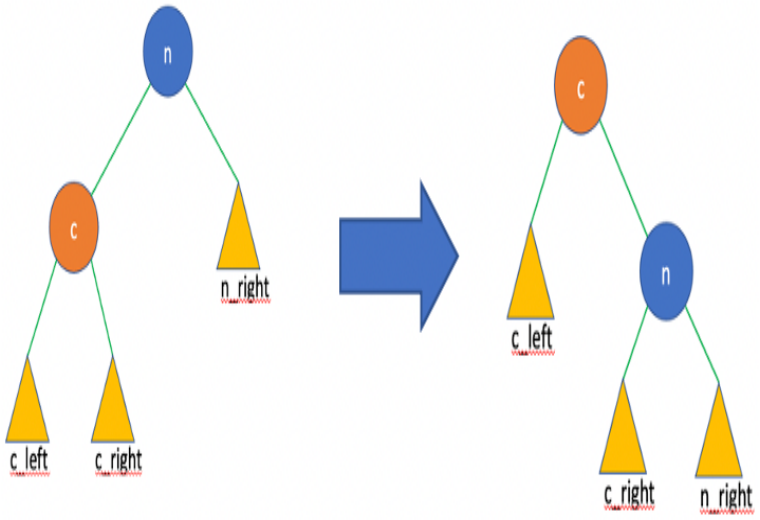

Figure 6: Zig Splay:

3.3.2 Zig-Zig Splay:

In this operation the searched node is moved using left or right rotation depending on the original position with respect to the parent and super parent node. These positions are Left-Left or Right-Right with respect to the parent and the super parent. as shown in the images figure[7] with the searched node as “p”:

3.3.3 Zig-Zag Splay:

In this operation the searched node is either on left-right position or right-left position w.r.t parent and super parent node as shown in the figure figure[8]. In the given figure the node “p” is the searched node.

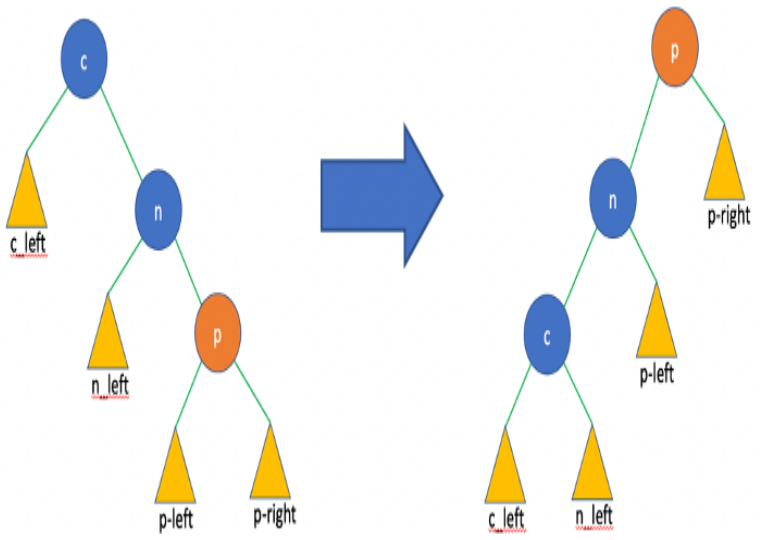

Figure 7: Zig-Zig Splay:

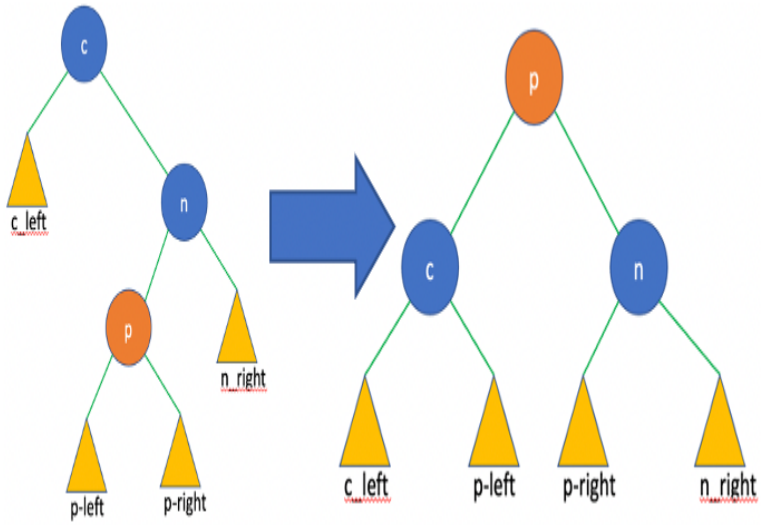

Figure 8: Zig-Zag Splay:

4 Problem Discussion:

As we discussed above that the splay tree is an extension of Binary Search Tree with the Amortized time complexity of search less than $O(\log_2(n))$. But there are some cases when the worst time complexity of this tree increases to O(n). This happens when the tree is either left skewed or right skewed as shown in the figure9:

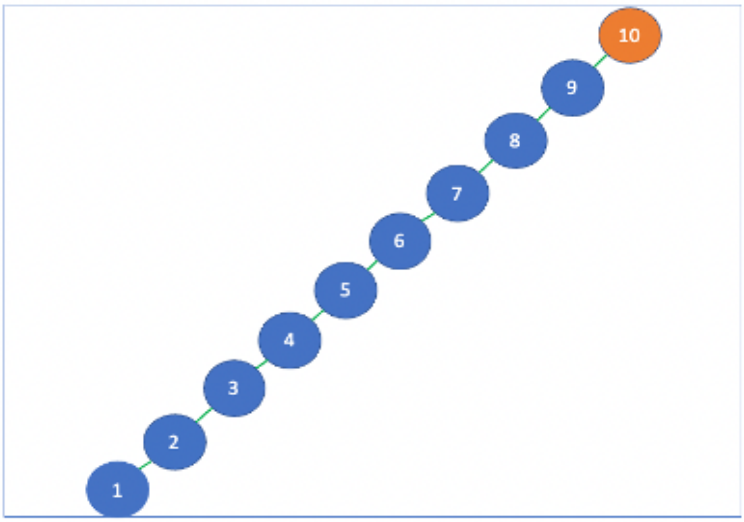

Figure 9: Left-Skewed Splay Tree after searching elements in strictly increasing order:

This can happen when the items are inserted or searched form the tree in a strictly increasing or decreasing order. When a tree is skewed, the search operation takes O(n) worst time to search the leaf node data or the data which is less than the leaf node in case of left skewed tree or greater than the leaf node in case of right skewed tree.

5 Research Paper Survey:

I read three research papers [1, 2, 3] to find a solution to mitigate this issue. In the first [1] research paper author talks about balancing a skewed tree by putting all the nodes of the tree in one array and then finding the center of that array. This center node will act as the root node of this tree with left sub tree will be the left side array of the node and right sub tree will be the right-side array of the same node. The center nodes of these sub trees will be the root nodes of these sub trees which will be directly connected to the parent node. This way, recursively all the sub arrays will be split into further sub arrays until there is only one element left in the sub array. This way we can create a balanced tree out of a skewed tree as shown in the figure 10.

According to the author this operation can be performed after formation of complete skewed splay tree. The time complexity to balance the tree will become O(n). Any search operation after that will require worst time complexity of $O(\log_2(n))$.

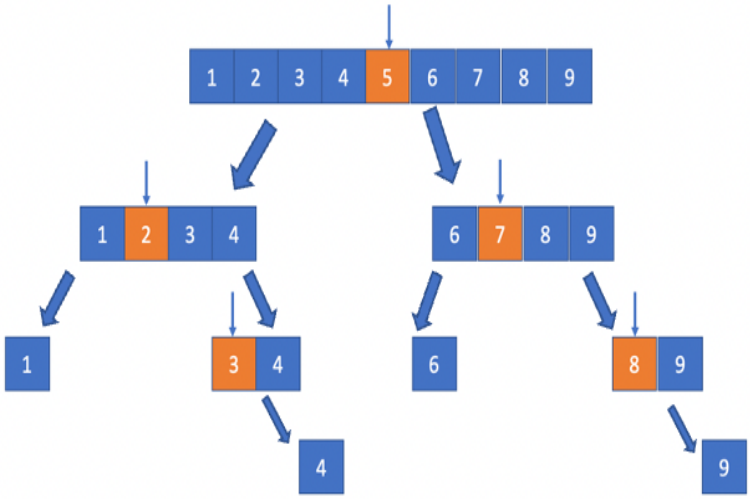

Figure 10: Converting Skewed tree into balanced binary search tree:

In the second and third research papers [2, 3] the author talks about the balancing operations using AVL tree techniques. These balancing operations are performed on the tree immediately after an unbalanced node is detected as discussed above. These rotational operations take constant time but searching the unbalanced node may take up to O(n) time complexity. This increases the time complexity of every balancing operation. To reduce this time complexity, we can use Dynamic programming, where each node can store more data which can be used to easily find the unbalanced node with minimum computation as shown in figure 11.

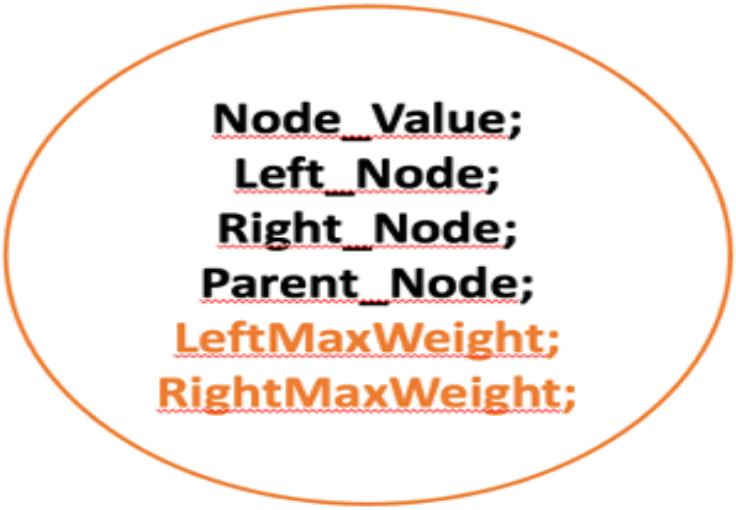

Figure 11: Dynamic Programming – inside node variables:

These variables will store required information which will be updated during every insert or search operation. According to the author, during any operation we don't have to update all the nodes every time, but we only need to update the nodes which are touched during these operations. These touched nodes are all recursive parent of the current searched or the inserted

node. After every search operation we must update the LeftMaxWeight and RightMaxWeight variables to the root. During updating the weights, we need to calculate the difference of these variables and if the |difference|>1 then that node is unbalanced. After finding the unbalanced node we can perform the suitable rotation operation as discussed above, to balance the node. While rotating we also must update the LeftMaxWeight and RightMaxWeight variables. We don't need to update the weights of sub trees of the searched and inserted nodes because their weights do not depend on the parent nodes.

Total Time complexity of insert operations (AVL tree) = complexity of finding node + complexity of balancing node Complexity of finding node = $O(\log_2(n))$ //As discussed above Complexity of balancing node = O(1)

Total Time Complexity of AVL tree for insert operation = $O(\log_2(n))$

6 Further Improvement:

For improvement I merged my findings from the above research papers to create a hybrid splay tree data structure which partially adheres AVL tree properties. Instead of balancing whole tree together when it is converted to a skewed tree as done in the first research paper, I introduced to perform the balancing after splaying the searched node to the top of the tree. Moreover, only the sub tree which is forming skewed tree is balanced instead of balancing the whole tree. In the example shared in figure 12 the left subtree of the root node is forming a skewed tree. This way we can maintain the time complexity of balancing the nodes under time complexity of $O(\log_2(n))$.

Total time complexity of search operation in a hybrid splay tree *= Time complexity of splaying the node to the top + ReccursiveOperation(Finding the unbalanced node in the sub tree of the root node * Time complexity of balancing the skewed side of the splay tree using AVL rotation properties) + Updating the weights of all the nodes.*

- Time complexity of splaying the node to the top: $O(\log_2(n))$ // same as the splay tree
- Time complexity of finding the unbalanced node: After performing a splay operation a new node is added to the top of a balanced tree as shown in figure 12 and figure 13. Therefore, if the tree is unbalanced then this new inserted node is the unbalanced node. We don't have to search for the unbalanced node.
- Time complexity of balancing the skewed side of the splay tree: The complexity of balancing the unbalanced node remains constant as it is in AVL trees. But there is a small change in finding the type of rotation required to balance the node.

  As we can see form the images [figure 13, 14], we cannot tell if the node 7 is left-left unbalanced or left-right unbalanced as the left child 4 has equal left and right weights. So, we must prioritise left-left or right-right over left-right or right-left. The reason is to

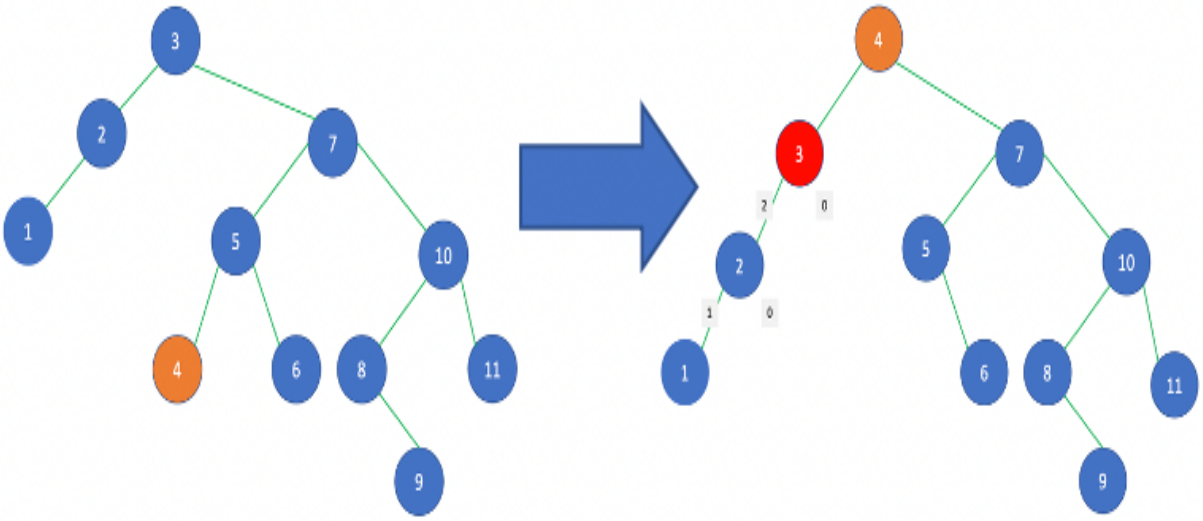

Figure 12: Formation of left skewed tree and while searching nodes in a sorted order

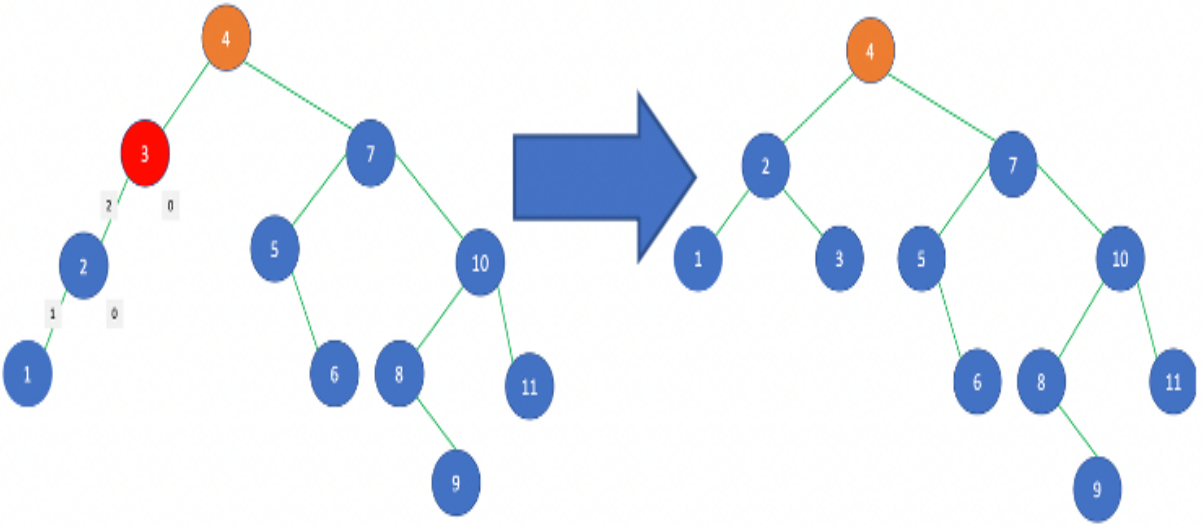

Figure 13: Figure 13: Balancing the left skewed tree using rotation operations like AVL tree

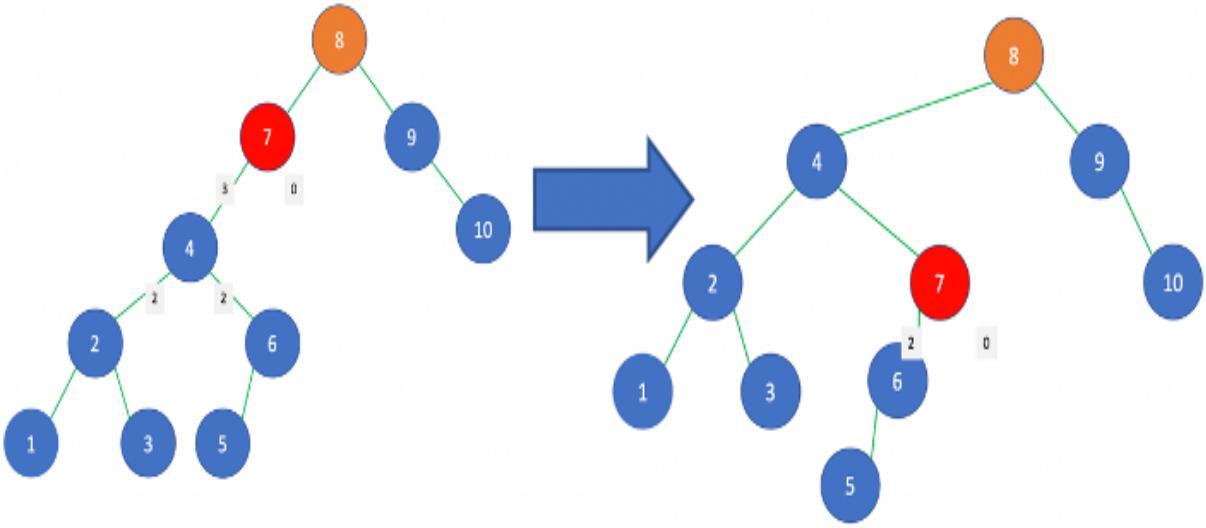

Figure 14: Balancing Left-Left unbalanced node (7) with right rotation balancing technique:

maintain the balance of all the child nodes of the unbalanced node. Let us assume that the left and right weights of any unbalanced node's child is "n". If we perform right rotation as done in figure 14 the right weight of that child node will become "n+1" whereas the left weight will remain "n". So that node will remain balanced as it was earlier. But the unbalanced node can remain unbalanced which is fine. This way the cost of finding the unbalanced node will remain constant. This unbalanced node may require recursive action to balance out, but all the operations' cost will remain $O(\log_2(n))$ including recursion as shown in figure 15.

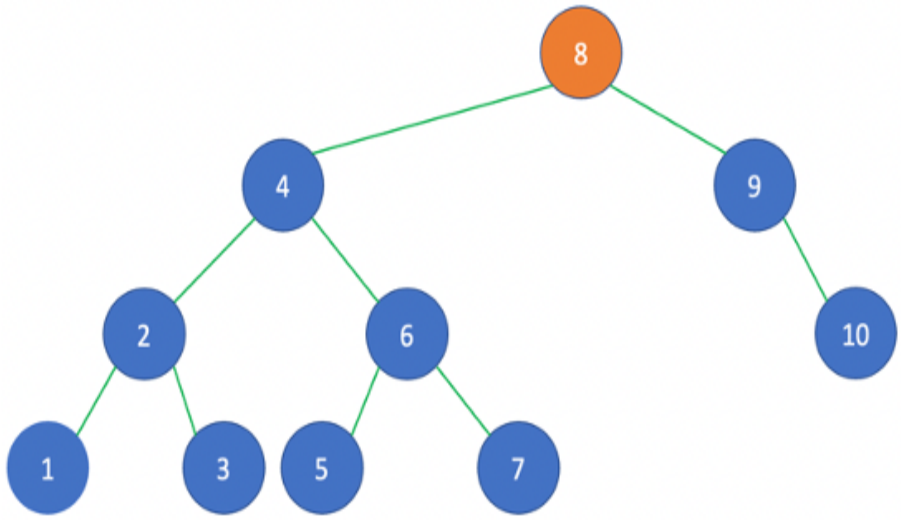

Figure 15: Balanced Splay Tree on the left sub tree of the root in strictly ascending search operations:

Time complexity of Updating the weights of all the nodes: Updating the weights will take $O(\log_2(n))$ like AVL tree as we need to update the weights of all the recursive parents which gets impacted while moving the node to the top during splay operations.

Therefore, the total time complexity of search operation in the hybrid splay tree = $O(\log_2(n))$ + $O(\log_2(n))$ + $O(\log_2(n))$

Total time complexity of search operation in hybrid splay tree = $O(\log_2(n))$

7 Comparison:

In this report we are comparing only search operations of all the trees with different number of elements in it. For Spay Tree and proposed Hybrid Splay Tree we are assuming the starting structure as AVL tree to start the operations from same point.

For comparison we are using 4 parameters:

- **Max Height:** For this parameter we check the max height of the tree after searching all the nodes in a strict ascending order. This gives us a proper idea of the skewed tree length.

- **Average Time complexity of searching all the nodes in Nano Seconds:** For this parameter we take the sum of time taken by every search operation while searching all the nodes in strict ascending order and then divide with the total no. of items in the tree. This gives us an average of the time taken by every search operation in the tree.
- **Search time complexity of last searched item in Nano Seconds:** For this parameter we search the last searched element again from the tree and note the time taken for this operation. This helps us to compare the trees. This will give us the best time complexity of search operation in splay tree.
- **Search time complexity of first searched item in Nano Seconds:** For this parameter we search the first searched element again after all the search operations. This will give us the worst time complexity of search operation in splay tree.

This comparison is done using different number of elements and the recorded the parameters as discussed above. The figure 16, 17, 18, 19 shows the parameter outputs for the different trees.

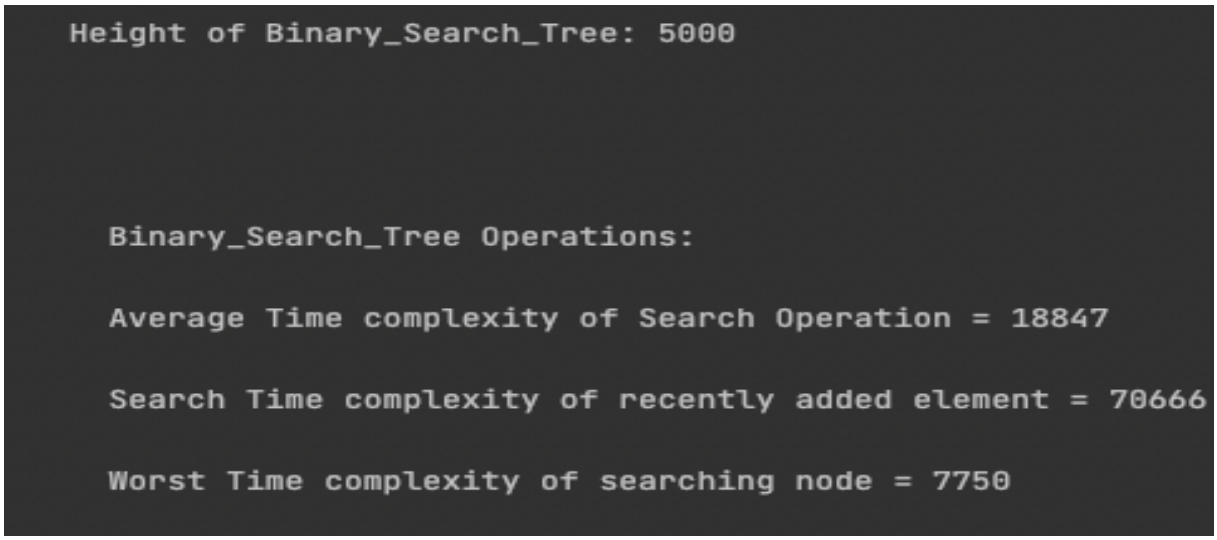

Figure 16: Comparison Parameters of Binary Search Tree:

- **Number of elements = 10:** When the number of elements are 10, we can observe the basic structure of the tree. The Binary Searched tree and Splay tree are left-skewed and right-skewed respectively. But the AVL tree and the hybrid splay tree are more balanced as shown in [figures 20, 21, 22, 23].

  After observing the tree structures in the [figures 20, 21, 22, 23], we can say that the Hybrid splay tree has partial properties of both Splay and AVL tree. As the recently searched node is on the top of the tree in both spay and hybrid splay trees. Moreover, the Hybrid splay tree is more balanced then splay tree, like a AVL tree. This can be better observed using the comparison parameters as shown in the table 1.

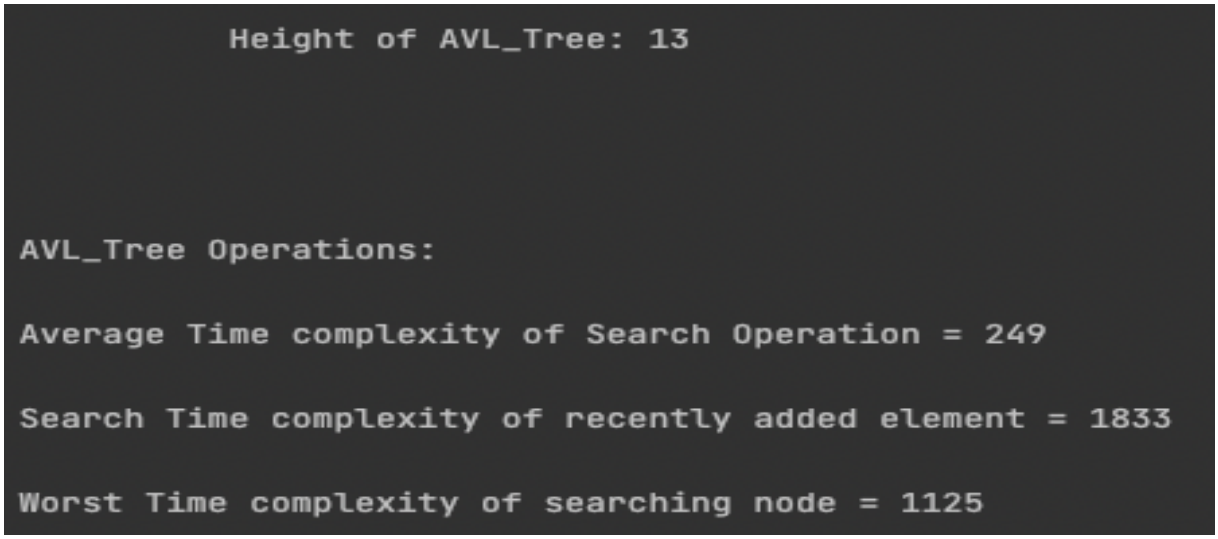

Figure 17: Comparison Parameters of AVL Tree:

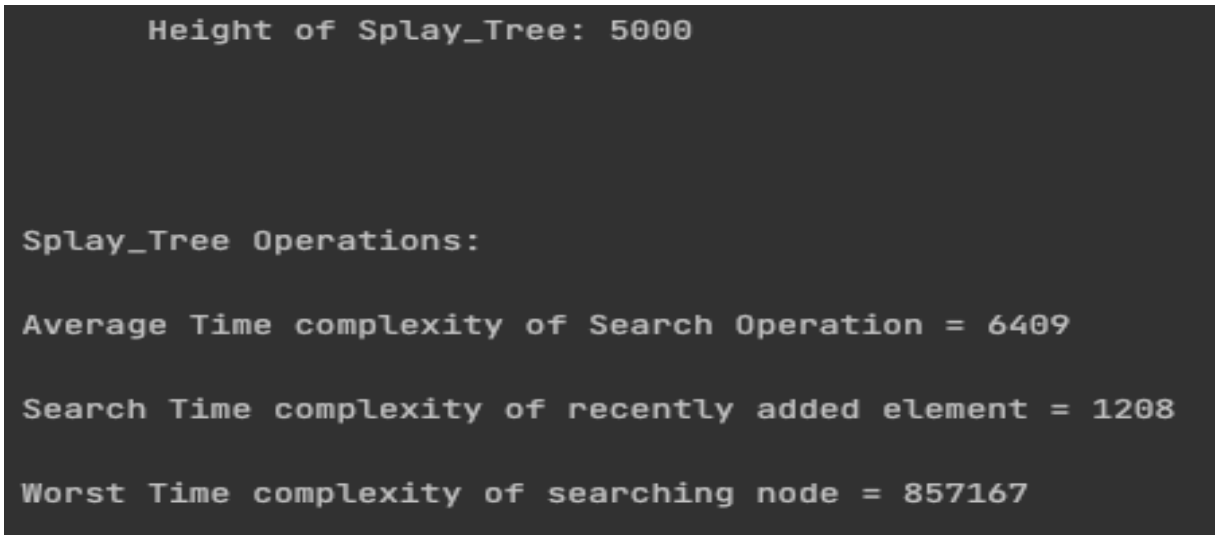

Figure 18: Comparison Parameters of Splay Tree:

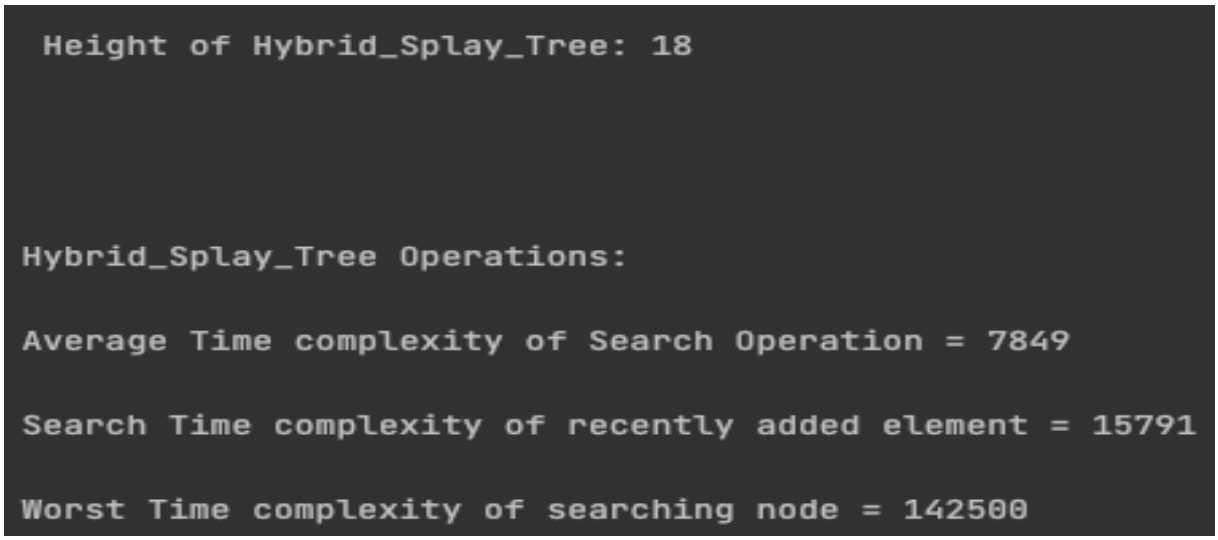

Figure 19: Comparison Parameters of Hybrid Splay Tree:

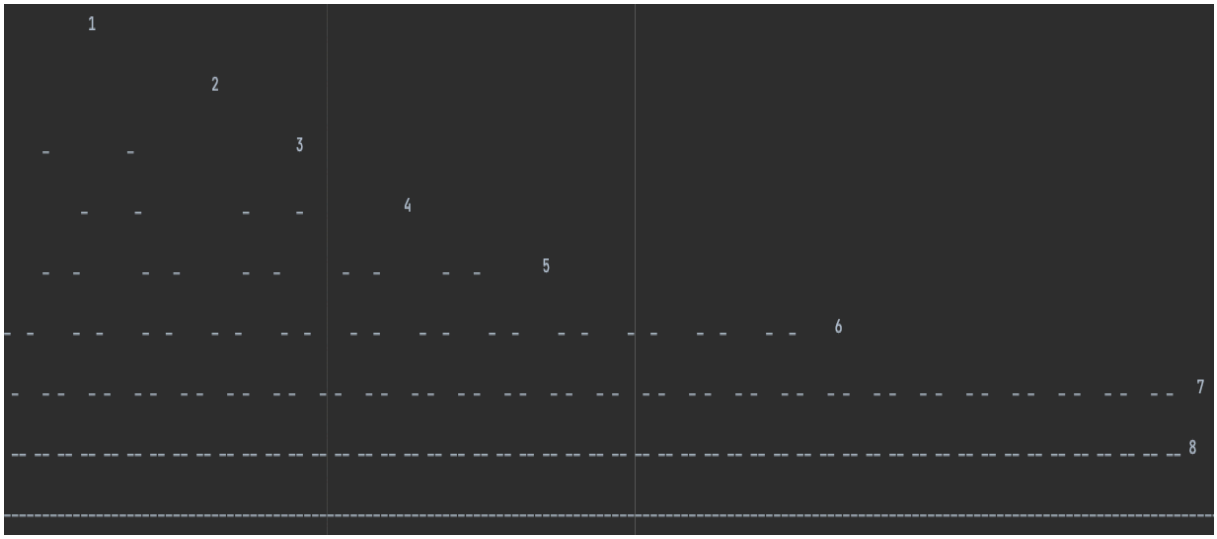

Figure 20: Right skewed Binary Search tree after inserting elements in sorted order:

Figure 21: Balanced AVL tree after all the search operations in sorted order:

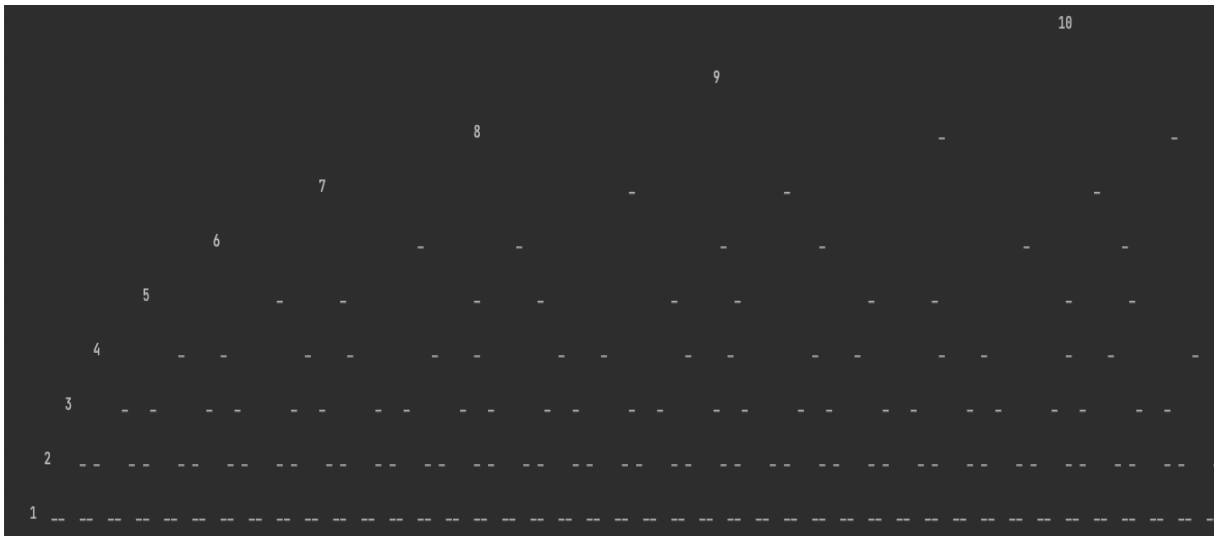

Figure 22: Left Skewed Splay Tree after search operations in sorted order:

Figure 23: Balanced Hybrid Splay Tree after search operations in sorted order:

TREE TYPES:				
Number of elements = 10	Binary Search Tree	AVL Tree	Splay Tree	Hybrid Splay Tree
Max Height of Tree	10	4	10	6
Average Time complexity of Search Operation	4608	704	438587	45391
Search Time complexity of recently added element	22833	1084	750	875
Worst Time complexity of searching node	5667	666	277709	8875

Table 1: Comparison Parameters in Nano Seconds with no. of elements = 10:

- **Number of elements = 100:** Table 2 shows the comparison parameter values for trees with 100 items in it.

TREE TYPES:				
Number of elements = 100	Binary Search Tree	AVL Tree	Splay Tree	Hybrid Splay Tree
Max Height of Tree	100	7	100	10
Average Time complexity of Search Operation	3723	2229	85774	77983
Search Time complexity of recently added element	35500	3208	1042	1875
Worst Time complexity of searching node	8417	1083	110708	55166

Table 2: Comparison Parameters in Nano Seconds with no. of elements = 100

- **Number of elements = 1000:** Table 3 shows the comparison parameter values for trees with 1000 items in it.

TREE TYPES:				
Number of elements = 1000	Binary Search Tree	AVL Tree	Splay Tree	Hybrid Splay Tree
Max Height of Tree	1000	10	1000	15
Average Time complexity of Search Operation	9226	434	13115	29845
Search Time complexity of recently added element	37834	1125	13666	1791
Worst Time complexity of searching node	7250	625	6775625	10291

Table 3: Comparison Parameters in Nano Seconds with no. of elements = 1000

- **Number of elements = 3000:** Table 3 shows the comparison parameter values for trees with 3000 items in it.
- **Number of elements = 5000:** Table 3 shows the comparison parameter values for trees with 5000 items in it.

TREE TYPES:				
Number of elements = 3000	Binary Search Tree	AVL Tree	Splay Tree	Hybrid Splay Tree
Max Height of Tree	3000	12	3000	17
Average Time complexity of Search Operation	12045	500	5676	8082
Search Time complexity of recently added element	29834	2167	10167	10917
Worst Time complexity of searching node	4667	2375	1447792	26250

Table 4: Comparison Parameters in Nano Seconds with no. of elements = 3000

TREE TYPES:				
Number of elements = 5000	Binary Search Tree	AVL Tree	Splay Tree	Hybrid Splay Tree
Max Height of Tree	5000	13	5000	18
Average Time complexity of Search Operation	18847	249	6409	7849
Search Time complexity of recently added element	70666	1833	1208	15791
Worst Time complexity of searching node	7750	1125	857167	142500

Table 5: Comparison Parameters in Nano Seconds with no. of elements = 5000

- **Number of elements = 50000:** Table 3 shows the comparison parameter values for trees with 50000 items in it.

TREE TYPES:				
Number of elements = 50000	Binary Search Tree	AVL Tree	Splay Tree	Hybrid Splay Tree
Max Height of Tree	50000	16	50000	23
Average Time complexity of Search Operation	*FAILED DUE TO MEMORY OF RECCURSIVE OPERATIONS*	162	*FAILED DUE TO MEMORY OF RECCURSIVE OPERATIONS*	2206
Search Time complexity of recently added element	*FAILED DUE TO MEMORY OF RECCURSIVE OPERATIONS*	197458	*FAILED DUE TO MEMORY OF RECCURSIVE OPERATIONS*	291167
Worst Time complexity of searching node	*FAILED DUE TO MEMORY OF RECCURSIVE OPERATIONS*	128625	*FAILED DUE TO MEMORY OF RECCURSIVE OPERATIONS*	479667

Table 6: Comparison Parameters in Nano Seconds with no. of elements = 50000

After observing the tables, we can see that the following trend on the comparison parameters:

- **Max Height:** The max height of the Hybrid tree is more balanced just like the AVL tree whereas, the splay tree's max height is increasing drastically with the increase in no. of elements.
- **Average Time complexity of searching all the nodes in Nano Seconds:** The average time complexity of search operations remains almost the same for AVL Tree, Splay Tree and Hybrid splay tree. But, for the binary search tree it is slightly higher, and the gap keeps on increasing with the increase in number of nodes.
- **Search time complexity of last searched item in Nano Seconds:** The search time complexity of last added element is same in case of Hybrid splay tree and Splay tree because both have that node at the root of the tree and this node is hit first whenever search operation is executed.
- **Search time complexity of first searched item in Nano Seconds:** The search operation of the first searched element is worst in case of Splay tree as it has this element at the end of the skewed tree. Searching this node needs us to traverse the whole tree which takes the max time which is O(n). When we search the same element in the AVL tree, it searched more easily because the tree is balanced with significantly less height. Hybrid splay tree's time is closer to AVL tree's time then splay tree's time, for this operation.

While searching the tree with 50000 items in it, the search operation failed due to the memory exceeded issue. For searching recursion was being used which requires stack memory. While searching the last element in the case of skewed splay tree, it crashed with the memory issue. Whereas the Hybrid splay tree and AVL tree were able to execute the operation without any issue as can be observed in table 6.

8 Conclusion:

AVL tree is always balanced and can maintain the items at their position without moving them during the search operations. Whereas the splay tree moves the recently searched item to the root which makes it easier to search for the next time. Hybrid splay tree performs both the operations of moving the recently searched node to the root of the tree and keep the sub trees balanced using AVL tree properties. This way we can reduce the worst time complexity of a splay tree. But there is a trade in this Hybrid splay tree. We can only keep only a few nodes close to the root node, which were recently searched. As if we try to balance the sub trees then these nodes will move down to further with every recursive balance operation. It is up to us that we can keep as many recently searched nodes close to the top as we want. After that the tree will be balanced to make it easier to access the leaf nodes.

References

[1] Siva, Kowsikan. (2017).SubTree balanced Splay Tree.

[2] Hills, Murray. (1985). Self-Adjusting Binary Search Trees.

[3] Jim Bell and Gopal K. Gupta. An evaluation of self-adjusting binary search tree techniques. Softw., Pract. Exper., 23(4):369–382, 1993.