**SubTree-balanced Splay Tree (STbST)**

**Author:**

* Kowsikan Siva ([kowsikan.siva@gmail.com](mailto:kowsikan.siva@gmail.com))

**Technologies and Concepts Used:**

* Python,
* Data structures,
* Tree data structures – Binary search tree, Avl tree and Splay tree.

**Introduction:**

In this paper, I explained a novel approach to balance the height of right and left subtrees of the Splay Tree (ST).

The problem in splay tree is, the tree may be become skew tree which means height of the tree is almost equal or equal to the number of nodes in the tree. This will increase the timing of insertion, deletion and search operations.

After balancing left and right subtrees of Splay Tree (ST), that tree is called SubTree-balanced Splay Tree (STbST). This STbST structure will reduce the timing of insertion, deletion and search operations. And this won’t affect any characteristics of Splay Tree (ST).

Before getting into ideology explanation we need to know some basic concepts of data structures which will helps us to understand SubTree-balanced Splay Tree (STbST) very well.

**Basic Concepts:**

I am explaining only the necessary concepts in data structures which is related to SubTree-balanced Splay Tree (STbST), because some data structures concepts are not necessary to understand SubTree-balanced Splay Tree (STbST).

**Data Structures:**

Data structures is a method of organizing, storing and retrieving of data in a computer memory.

Types: 1) Linear data structures,

2) Non-linear data structures.

**Non-linear Data Structures:**

The data structures where data items are not organized sequentially. In non-linear data structures single data could be connected to more than one elements (data) to reflect special relationship among them.

Ex: Tree data structures, Hash map, etc.

**Tree Data Structures:**

Tree data structures is a collection of data (node) which is organized in hierarchical structure and this is recursive definition.

**Node:**

Organized memory area for storing actual data and storage location address of next node it connected with.

**Edge:**

Link between two nodes is called as edge. If ‘n’ number of nodes available in tree means ‘n-1’ edge should be present.

**Root:**

First node of the tree and origin of tree structure is called root node. There should be only one root and tree must have root node.

**Parent:**

The node which has a branch from it to any other node is called parent node.

**Child:**

The node which is descendant of any node is called child node.

**Leaf:**

The node which does not have any node as its child node is called leaf node.

**Height:**

Total number of edges from leaf node to particular (parent) node in the largest path is called height.

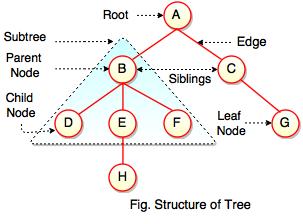
**Subtree:**

Every child node will form a subtree on its parent node.

**Skew Tree:**

Skew tree which means height of the tree is almost equal or equal to number of nodes in the tree.

These above explained concepts are given as a picture for clearer understanding. (Img1)



**Img1-Structure of tree**

**Related Trees:**

Following trees are closely related to SubTree-balanced Splay tree (STbST). So we are in need of understand those tree concepts.

1. Binary search tree (BST),
2. AVL tree (AVL),
3. Splay tree (ST).

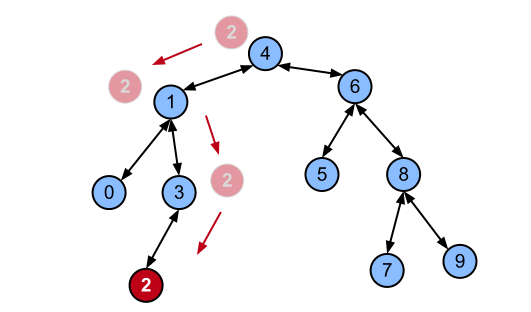
**Binary Search Tree (BST):**

In Binary search tree (BST) parent nods are only allowed to have two child at maximum. Any parent nodes left child value always lower than its parent node and right child value always lower than its parent node.

**Insertion:**

Insertion operation in BST means inserting node (values) based on previous inserted node (values). First ever inserted node (values) is root node and further inserted node (values) are positioned based on previous inserted node (values). Any parent nodes left child value always lower than its parent node and right child value always lower than its parent node.

Let us see how value 2 is inserted in Binary search tree (BST). From Img2 we can clearly understand how insertion operations performed.



**Img2-BST Insertion**

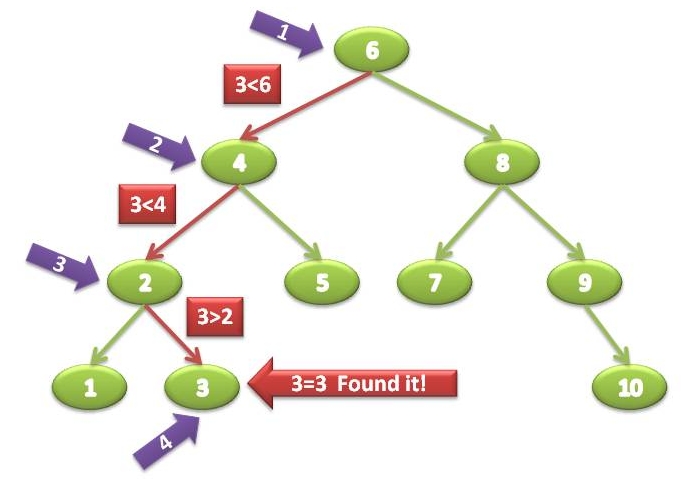
**Search:**

We can check the presence of node (value) in Binary search tree (BST).

As usual the search operation starts with root node. If search value equal to root then it will return element found. Otherwise it will check search value is lesser or greater than root. If it’s lesser then it will check in left sub tree.

If it’s greater then it will check in right sub tree. It will keep on do this comparison process till it reach value or leaf node. Even if leaf node also not match with search value then it will return search value not found.

Let us see how value 3 is searched in Binary search tree (BST). From Img3 we can clearly understand how search operations performed.



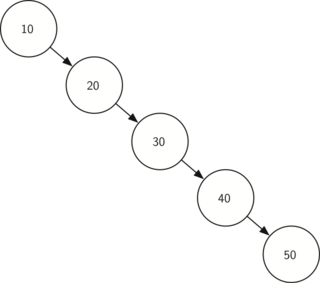
**Img3-BST Search**

**Skew tree in BST:**

From previous definition of skew tree we know how it will affect the tree operation timing.

Mostly if skew tree formed in BST or any tree it will create worst case operation timing. Worst case means, “Time taken to perform operations like insert or search or delete will be large”.

From the below image we can clearly understand skew tree and worst case.



**Img3-Skew tree**

**Worst case in Search operation:**

Let’s search value 50 in above tree. First it will check with 10, not matched and 50 is greater than 10 so it will go to its left child (20) step1.

It will check with 20 not matched and 50 is greater than 20 so it will go to its left child (30) step2.

Same operation will be performed till it find 50. At step5 it will find value 50. Here number of nodes (here 5 nodes) in tree is equal to number of steps (here 5 steps) to find value 50. This is known as worst case search operation in BST.

Similarly if we are searching value which is not present in tree also will leads to worst case. For example if we are searching 60 in the above tree it will check till 50 (step5). After that only it will say element not found.

**Worst case in Insert operation:**

Let’s insert 60 in above tree. It will check till 50 (step5). After that only it will insert 60 as a right child of 50. Here also number of nodes (here 5 nodes) in tree is equal to number of steps (here 5 steps) to insert value 60. Again if we insert 70. It will check till 60 (step6). After that only it will insert 70 as a right child of 70.

**Worst case in Delete operation:**

Let’s delete 50 in above tree. It will check till 50 (step5). After that only it will delete 50. Here also number of nodes (here 5 nodes) in tree is equal to number of steps (here 5 steps) to delete 50.

All worst case in BST mentioned as O(n) in time complexity. Here O(n) means all (insert or delete or search) operation will process all the nodes in that tree to do its respective operation. To manage the worst case in BST, AVL tree was created. AVL tree will balance the height of tree so no way to form skew tree in AVL tree. Let’s discuss AVL tree.

**AVL Tree (AVL):**

AVL tree is a self-balanced binary search tree which means tree nodes will auto rotate or auto arrange itself to balance height of tree. Whenever any parent node’s balance factor not equal to -1 or 0 or 1 then the tree will perform some rotation operation to balance itself.

Balance factor is calculated from the following formula.

**Balance Factor (BF) = [Height of left subtree] – [Height of right subtree]** (Should always equal to -1 or 0 or 1)

Let’s see the types of rotation which will make the AVL tree balanced. Then we will see how rotation will make the tree balanced.

**Rotations:**

Single Rotation: 1) Left Rotation

2) Right Rotation

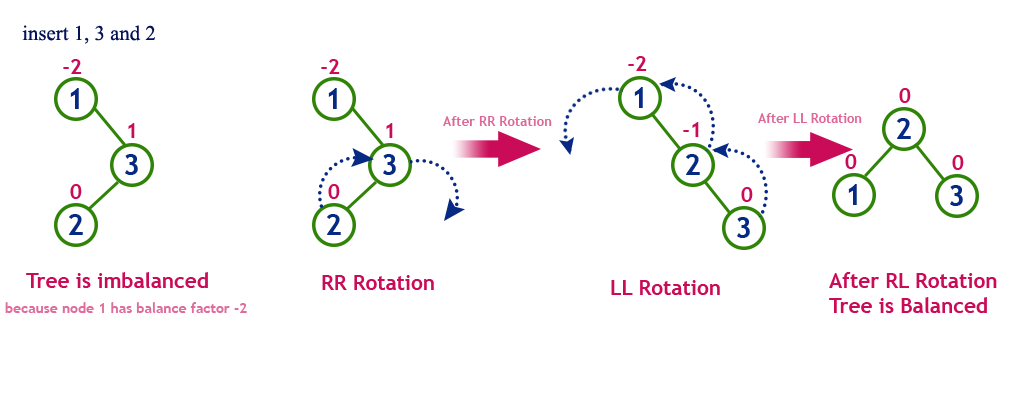
Double Rotation: 1) Left Right Rotation

2) Right Left Rotation

**Insertion:**

Insertion operation in AVL tree is similar to BST. But as mentioned above when the tree’s balance factor is affected it will perform respective rotation operations to make the tree balance.

Let’s see with an example for insertion and rotation. (Img4).

Insert 1, 3 and 2. Till inserting 1 and 3 balance factor is not affected. When inserting 2, balance factor will get affected. So rotation operation will be performed to make the AVL tree balanced.

**Img4-AVL Insertion and Rotation**

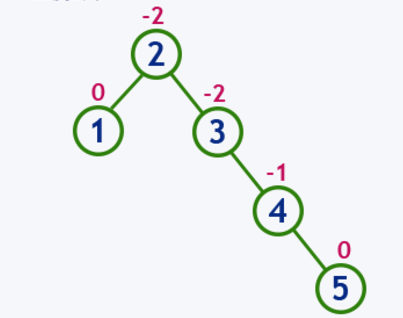
**Search:**

We can check the presence of node (value) in AVL tree.

As usual the search operation starts with root node. If search value equal to root then it will return element found. Otherwise it will check search value is lesser or greater than root. If it’s lesser then it will check in left sub tree.

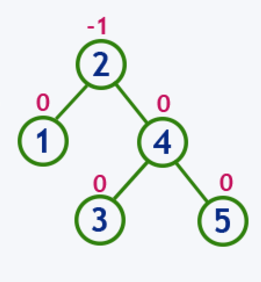
If it’s greater then it will check in right sub tree. It will keep on do this comparison process till it reach value or leaf node. Even if leaf node also not match with search value then it will return search value not found.

Let’s search node (value) 5 in Img5. It will take 4 steps to find. And the balance factor is not matched for some parent nodes.



**Img5-AVL Search before Balance**

Let’s search node (value) 5 in Img6 which is Img5’s balanced form. It will take 3 steps to find and BF satisfied.



**Img6-AVL Search after Balance**

**Delete:**

Delete operation in AVL tree is similar to BST. But in AVL tree after deleting any node it will check balance factor (BF) for every node. If any node not satisfy the balance factor (BF) condition then it will rotate tree nodes to make it balance.

**Skew tree in AVL:**

As we discussed before in BST about skew tree, in AVL there is no possibility for skew tree. The reason is whenever balance factor not satisfied the condition or skew tree formed at that time it will perform rotation operation to make the tree balance.

**Worst case in any operation:**

In AVL tree both average and worst case for any operation is O() where “n” is number of nodes in AVL tree. For example let’s take number of node value is 7 (n=7).

In BST, worst case is O(n). It need to process all nodes. For this example it need to process all 7 nodes.

But in AVL tree both average and worst case is O(). Here O()=2.80(approximate 3) which means, to perform any operation it need to process at max 3 nodes only (3 out of 7 nodes).

AVL tree is invented to overcome the skew tree formation in BST. On Splay Tree only we implemented our concept. Let us discuss Splay Tree (ST) next.

**Splay Tree (ST):**

Splay tree (ST) is a self-adjusted binary search tree in which every operation on an element rearrange the tree so that the element is placed at the root position of the tree. All the operations on Splay Tree (ST) involved with common operation called “Splaying”. Here I mentioned self-adjusted not as a self-balanced.

**Splaying:**

Splaying an element is the process of bringing it to the root position by performing suitable rotation operation. In a Splay Tree (ST) splaying an element rearrange all the elements in the tree so that the splayed element is placed at root of the tree. Splaying operation automatically brings more frequently used elements closer to the root of the tree.

**Rotations:**

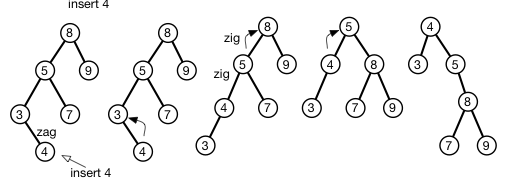
Splay tree (ST) has following rotation operations. Splaying will use these rotation operations to move element to root. These rotations are like rotations in AVL tree but here used to move elements to root.

1. Zig Rotation
2. Zag Rotation
3. Zig-Zig Rotation
4. Zag-Zag Rotation
5. Zig-Zag Rotation
6. Zag-Zig Rotation

**Insertion:**

Splay tree (ST) insertion operation is similar to BST. But as mentioned above after insert an element, tree will be rearranged to move recently inserted value to the root.

Let’s insert node 4 (value) in the tree. Then splaying concepts will be performed. Using some splay tree rotation operation node 4 (value) will become root. Img7



**Img7-ST Insertion and Splaying**

**Search:**

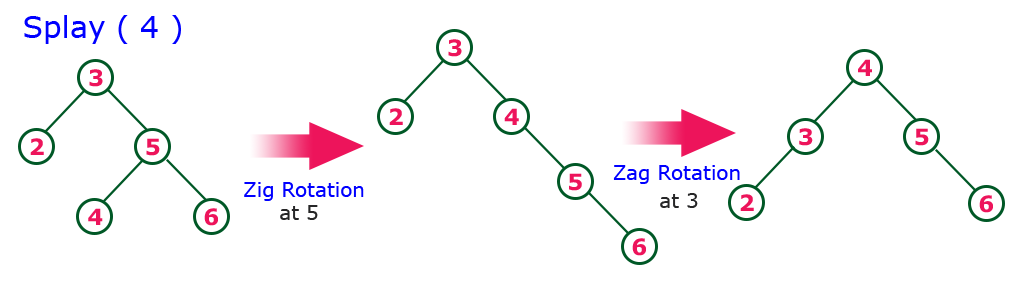
We can check the presence of node (value) in Splay tree (ST).

As usual the search operation starts with root node. If search value equal to root then it will return element found. Otherwise it will check search value is lesser or greater than root. If it’s lesser then it will check in left sub tree.

If it’s greater then it will check in right sub tree. It will keep on do this comparison process till it reach value or leaf node. Even if leaf node also not match with search value then it will return search value not found.

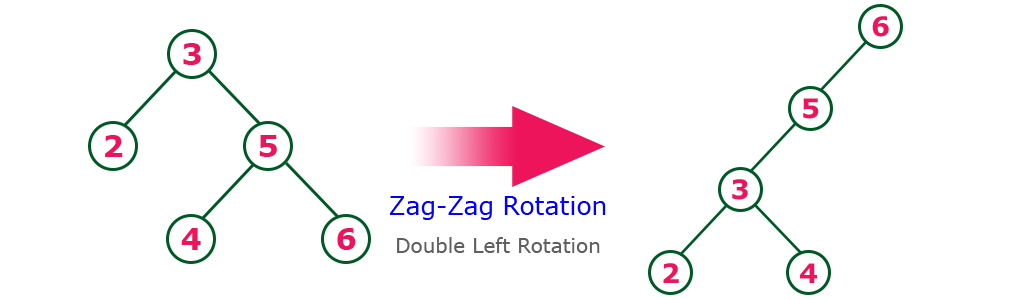
But as mentioned above after search (if found) an element, tree will be rearranged to move recently searched (if found) value to the root. If element not found it will change smaller or larger value in the tree as a root. If searched value is larger than largest element in tree means largest value in the tree will become root. If searched value is lesser than smallest element in tree means smallest value in the tree will become root.

Let’s see examples in Img8, Img9 and Img10.



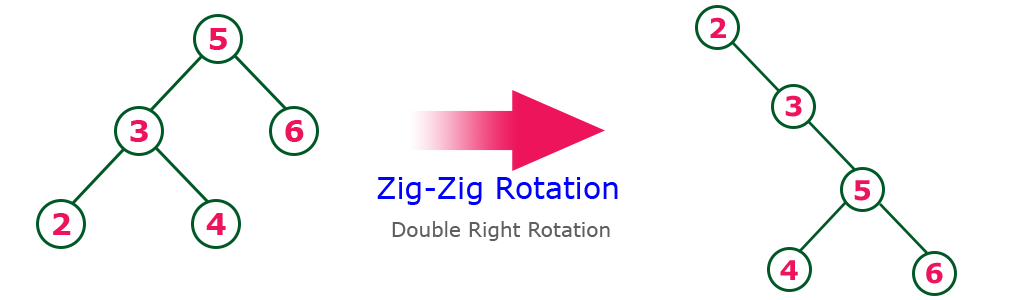
**Img8-ST Search and Element Found**

Let’s search 7 in Img9. Element not present so Largest 6 value changed as root.



**Img9-ST Search and Element Not Found and Larger**

Let’s search 1 in Img10. Element not present so smallest 2 value changed as root.

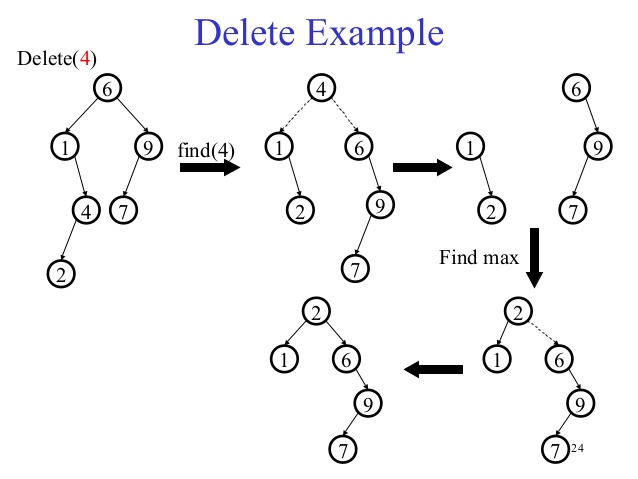


**Img10-ST Search and Element Not Found and Lesser**

**Delete:**

Delete operation in ST will make the element as a root first and it will delete that element. Then splited trees will be combined if when deleting root if the tree splited. Let’s see delete 4value in Img11.

It will find 4 and will make it as root. Then it will delete 4. Now tree split into 2 trees. Now it will find larger value in left sub-tree. Here 2 is the largest value in left sub-tree. Then 2 will become a root and trees will be combined.

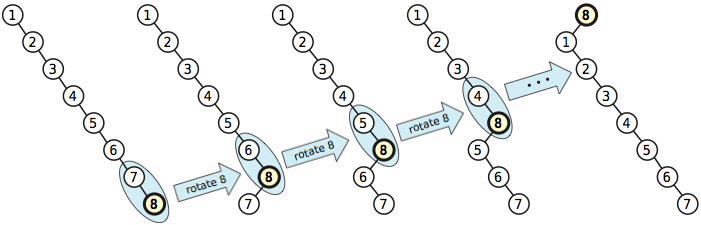


**Img11-ST Delete 4**

**Skew Tree:**

In the Splay Tree (ST) definition, mentioned that Splay Tree is a “self-adjusted” tree and not a “self-balanced” tree. Which means that nodes will not balance as like AVL tree. When making particular node as root it will arrange other nodes to make particular node as a root node.

In Splay Tree (ST) while adjusting nodes to make particular node as a root node there is possibilities for formation of skew tree. Let us see with an example. In Img12, insert value 8. Then the tree will make node 8 as a root. While adjusting other node to make node 8 as a root, it form skew tree.



**Img12-Skew Tree in Splay Tree (ST)**

**SubTree balanced Splay Tree (STbST):**

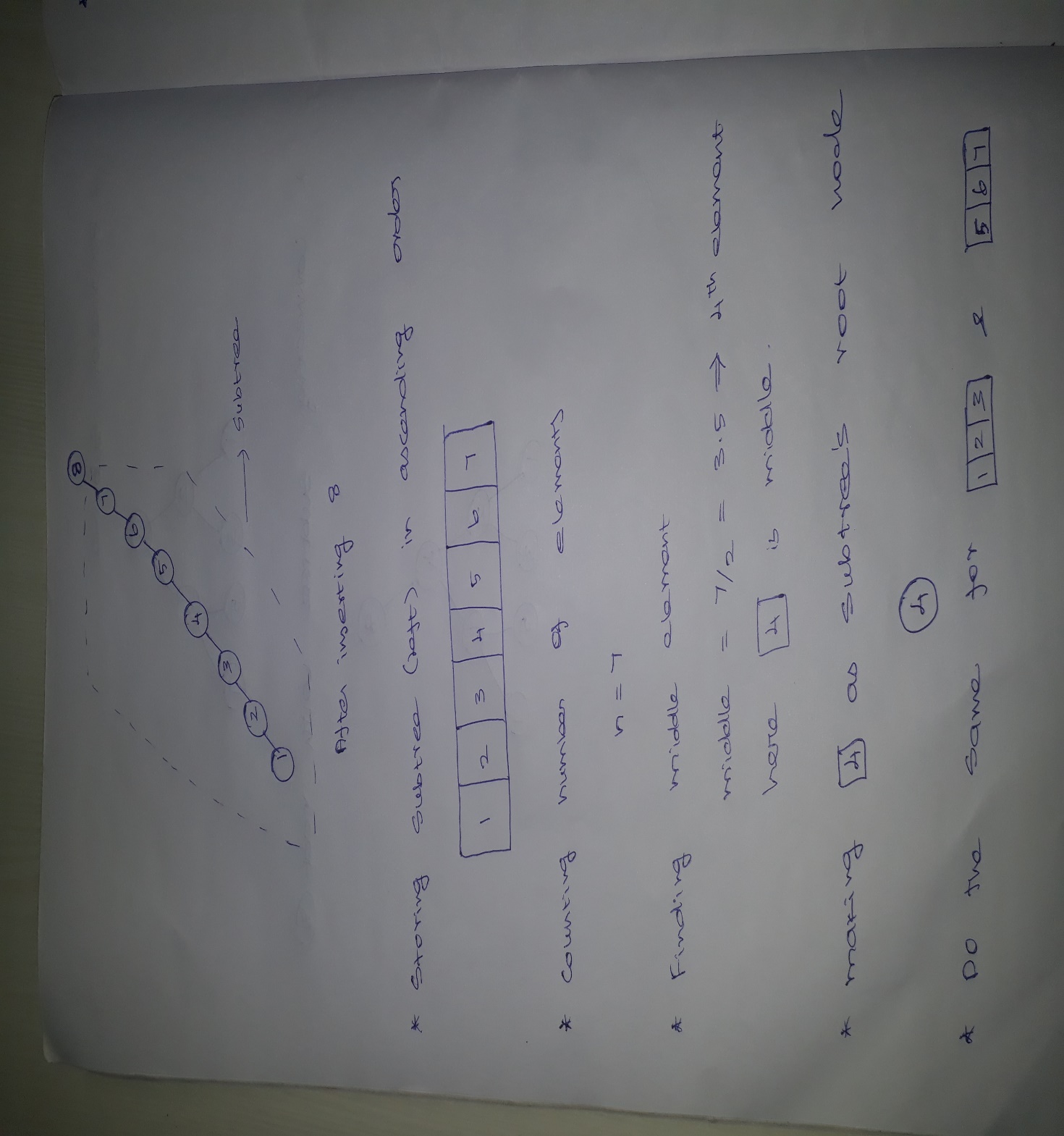
Here comes our SubTree balanced Splay Tree (STbST) to overcome the skew tree formation in Splay Tree (ST).

**Ideology:**

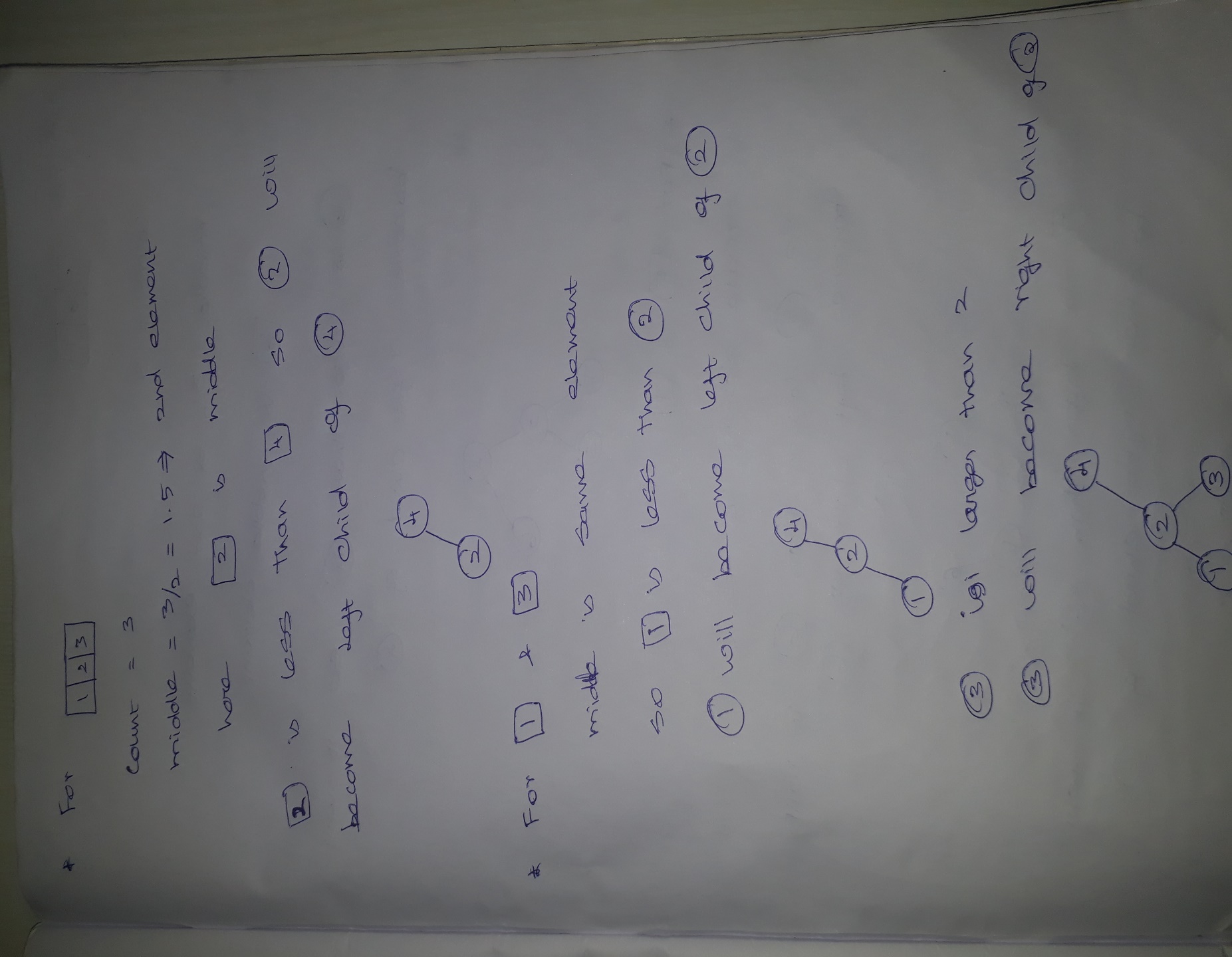
Idea behind this concept is “Taking right and left subtree of the splay tree root node and convert those subtrees as a balanced subtree and link those balanced subtrees to the root again.” This operations will be performed for all the operations like insert, delete and search. So the insertion, deletion and search operation time complexity will be reduced.

**Explanation:**

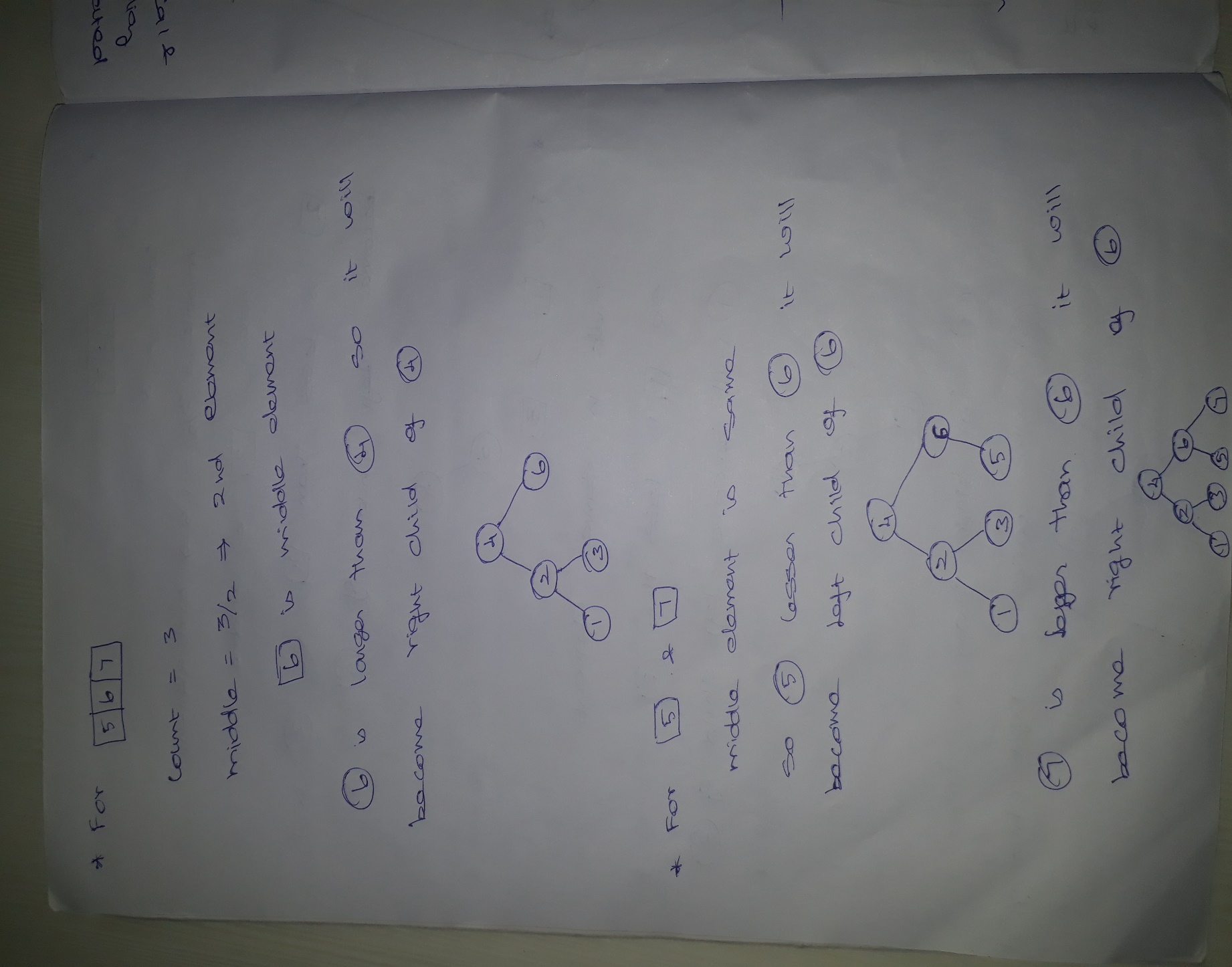
1. In Img13 explained with picture. Here I explained how subtree balanced. After inserting 1, 2, 3, 4, 5, 6, 7 and 8 tree will look like this (Img13). We will take left subtree of the root node (Here only left subtree available) those are 7, 6, 5, 4, 3, 2 and 1.
2. We will store subtree node values in ascending order. After performing ascending operation value will be like this. 1, 2, 3, 4, 5, 6 and 7.
3. We will calculate number of nodes in subtree. Here 7 nodes in subtree. To make the subtree balanced we will find the middle value of the available nodes. Here 4 is the middle node. And 4 will be fixed as a root node of the balanced subtree.
4. Then we will get the nodes before and after 4. Before values are 1, 2 and 3. And after values are 5, 6 and 7.
5. We will perform the step 3 again and again till all nodes arranged in balanced form.
6. For example number of nodes in before values of 4 is 3. Middle value in that before value is 2. And that 2 is fixed as a left child of 4.
7. Now before and after value of 2 is taken and same operation is performed. 2’s left child is 1 and right child will be 3.
8. Similarly after values of 4 is 5, 6 and 7. Number of nodes is 3. Middle value is 6. And 6 will be placed as a right child of 4.
9. Now before and after value of 6 is taken and same operation is performed. 5’s left child is 6 and right child will be 7.
10. Now this balanced subtree will be linked with root 8.
11. Thus the skew tree formation is removed using our idea.



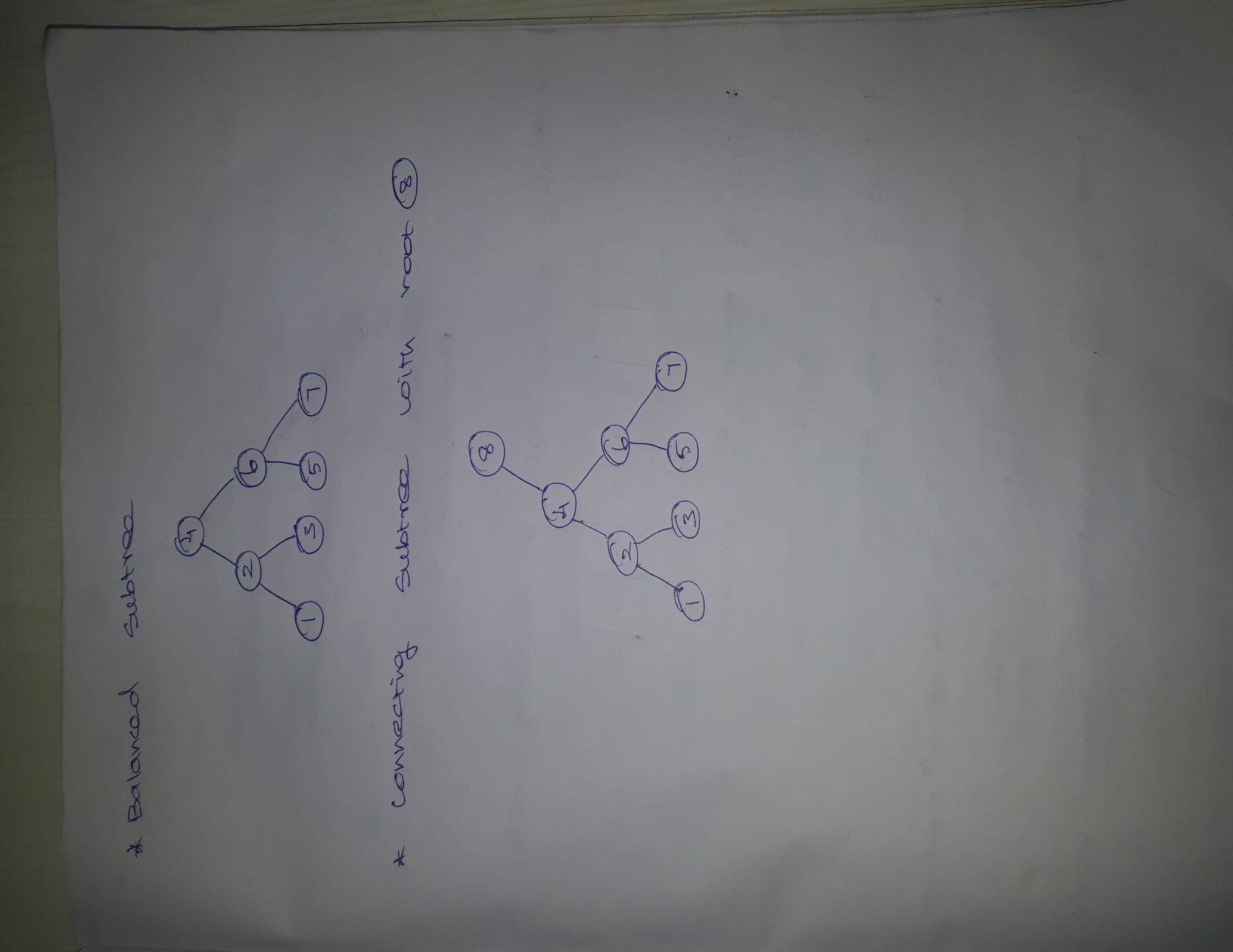
**Img13A**



**Img13B**



**Img13C**



**Img13D**

**Insertion:**

Insertion operation in SubTree balanced Splay Tree (STbST) is similar to Splay Tree (ST). But as mentioned above right and left subtree will be converted into balanced subtree and linked with root.

**Search:**

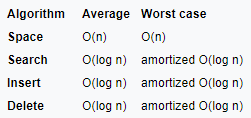
Search operation in SubTree balanced Splay Tree (STbST) is similar to Splay Tree (ST). But as mentioned above right and left subtree will be converted into balanced subtree and linked with root.

**Insertion:**

Delete operation in SubTree balanced Splay Tree (STbST) is similar to Splay Tree (ST). But as mentioned above right and left subtree will be converted into balanced subtree and linked with root.

**Time complexity:**

Time complexity of Splay Tree (ST) is

****

In our SubTree Balanced Splay Tree (STbST) also have same time complexity as Splay Tree (ST). But because of removal of skew tree formation the operations like insert, delete and search will be faster than Splay Tree (ST).

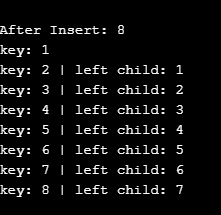
**Real Time Applications:**

We can use SubTree Balanced Splay Tree (STbST) where ever we are using Splay Tree. Some examples areas are given below.

1. Network router to route the packages
2. Memory caches
3. Lexicographic search tree
4. Data compression
5. Encryption
6. Optimize a text editor by avoiding large string copies

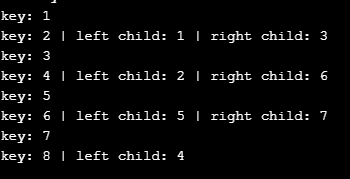
**Program output screen comparison between ST vs STbST:**

Img14 is output of C++ code of Splay tree for nodes 1, 2, 3, 4, 5, 6, 7 and 8.



**Img14**

Img15 is output of C++ code of SubTree balanced Splay Tree for nodes 1, 2, 3, 4, 5, 6, 7 and 8.



**Img15**