1. **Splay Tree Algorithm**

**Concept-**

Splay Tree Algorithm is basically the rearrangement of the elements in such a way that after the operation on the element, that element is placed at the root of the tree.

**Operation-**

1. Insertion- We will insert the element like a typical binary search tree at the leaf. After that we will perform splaying at each step. Splaying is rearranging the elements in such a way that the newly inserted element is made the root. We will run a loop for splaying until the newly inserted element is made the root.
2. Deletion- Deletion operation require searching of the element in the tree in the same way as binary search tree. After finding the element we splay it until it becomes the root of the tree. After making it the root, we delete that element, which gives us two subtrees. Then we find the greatest element from the left subtree and splay it until it becomes the root of that left subtree. At the end, we attach the right subtree as the right child of the previously splayed left subtree.
3. Search- In the search operation, we perform typical binary search tree operation for the element. After finding that element, we splay it until it becomes the root of the tree.

**Pseudocode-**

Splaying function is common in all the operation so we’ll have a common splay function,

Splay(T, n) // T is the tree and n and is the node which has to be splayed

While n.parent != null

If n.parent == T.root

If n == n.parent.right

Rotate\_left( T, n.parent)

Else

Rotate\_right(T, n.parent)

Else

p=n.parent

g=p.parent

If (n.parent.left ==n and p.parent.left ==p)

Rotate\_right(T,g)

Rotate\_right(T,p)

else if (n.parent.right ==n and p.parent.right ==p)

Rotate\_left(T,g)

Rotate\_left(T,p)

else if (n.parent.right ==n and p.parent.left ==p)

Rotate\_left(T,p)

Rotate\_right(T,g)

else

Rotate\_right(T,p)

Rotate\_left(T,g)

End While

1. Insertion-

insertion(T,n)

temp=T.root //temp is temporary variable

y=NULL

While(temp != NULL)

y=temp

if (n.data < temp.data)

temp=temp.left

else

temp=temp.right

n.parent=y

if (y==NULL)

T.root=n

else if (n.data < y.data)

y.left=n

else

y.right=n

splay(T,n) //splay

1. Deletion-

Deletion(T,n)

Left\_tree //create an empty subtree

Right\_tree //create an empty subtree

Left\_tree.root=T.root.left

Right\_tree = T.root.right

If (Left\_tree.root != NULL)

Left\_tree.root.parent = NULL

If (Right\_tree.root !=NULL)

Right\_tree.root.parent = NULL

If (Left\_tree.root != NULL)

m= MAX (Left\_tree , Left\_tree.root)

splay (Left\_tree, m) //splay

Left\_tree.root.right = Right\_tree.root

T.root =Left\_tree.root

else

T.root =Right\_tree.root

1. Searching

search (T, n , x) // T is the tree, n is the node , x is the element to be searched

If (x==n.data)

splay(T,n) //splay

return n

else if (x < n.data)

return search (T, n.left , x)

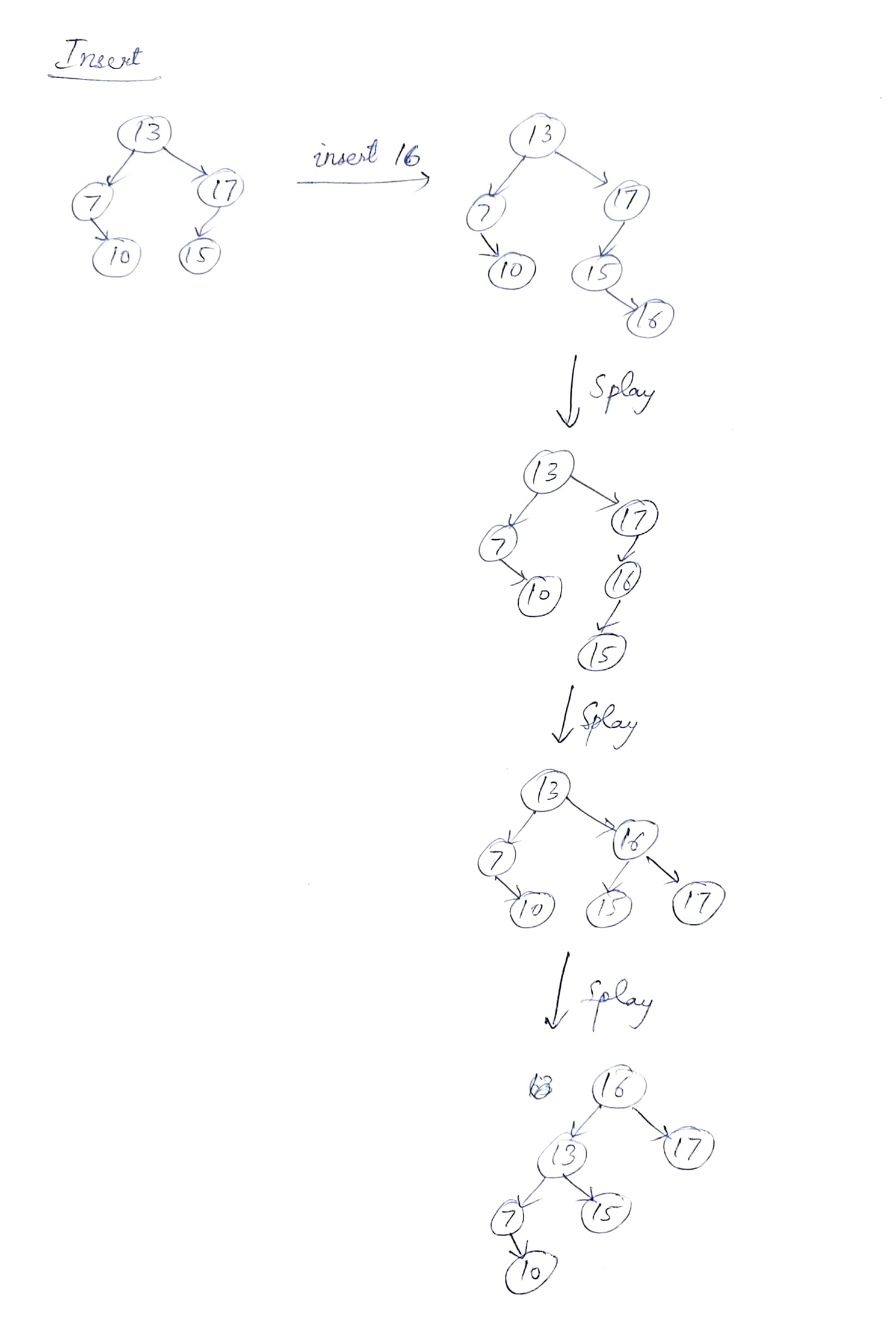
else if (x > n.data)

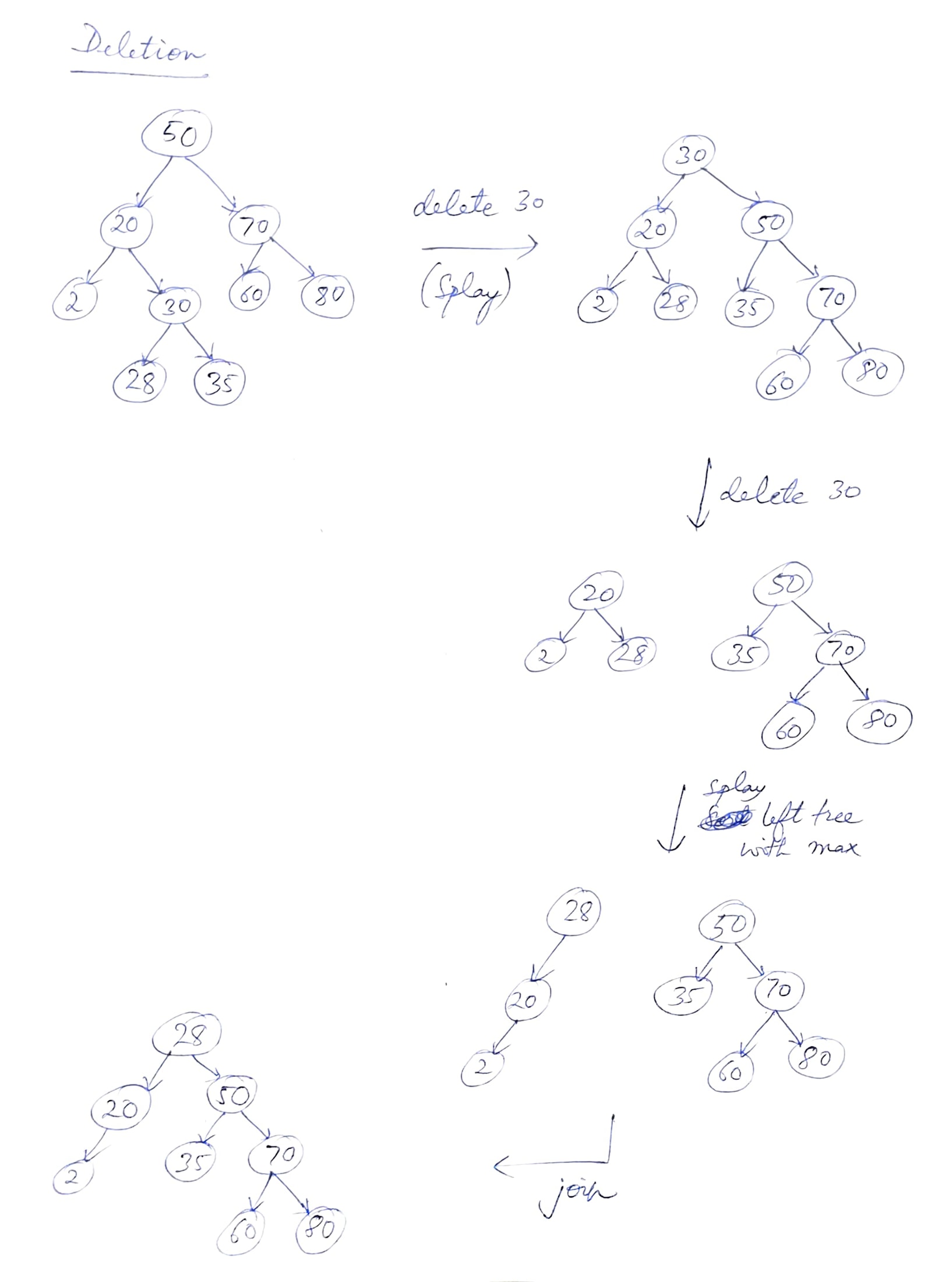
return search (T, n.right , x)

else

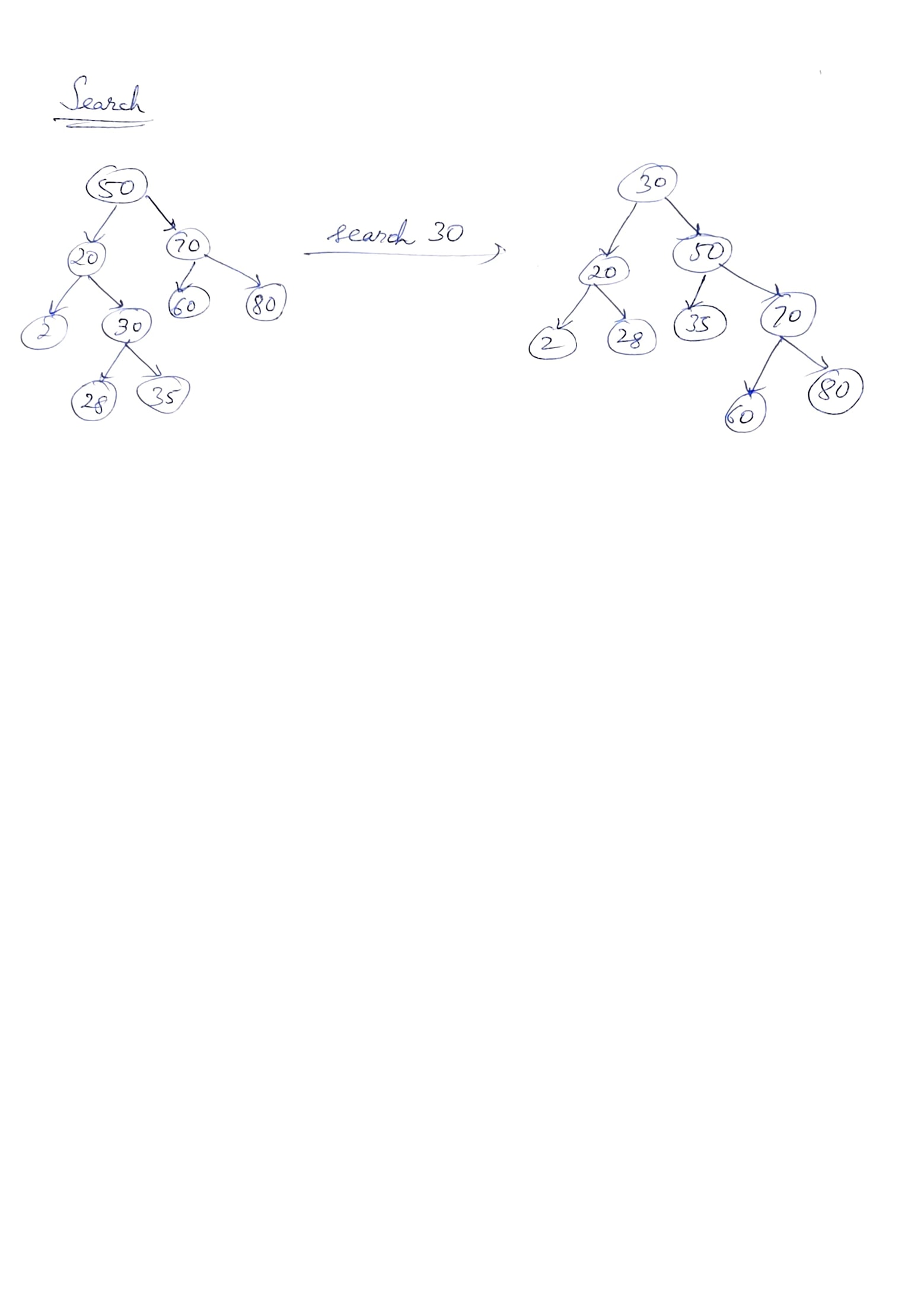
return NULL

**Example:**

1. Insertion-
2. Delete



1. Searching



**Time Complexity-**

Complexity of splay tree algorithm is O(log n)

We confirm this complexity through the idea of splay tree that each node is made the root after each operation, hence every insertion, deletion and search operation will have the average as well the worst time complexity of O(log n)

**Notable special properties-**

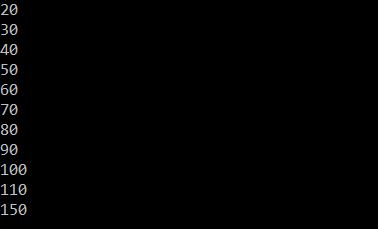
1. All the operation on splay tree takes O (Log n) average time.
2. After every operation on the splay tree, the element on which operation happens becomes the node which can be easily accessed. This property has multiple use cases in real life.
3. It has small memory footprint that is it does not need to store bookkeeping data.

**Practical application-**

1. Splay trees are used in places where recent data is required to be executed such as in the implementation of cache or data compression.
2. It can be used in the network router where data packets are transferred quickly to outgoing wire based on its IP address. So based on the previous transfers on IP address, using splay tree we can reduce the decision making of going through entire IP address and just transfer to the IP address if it comes again.
3. It is used in GCC compiler, Linux loadable kernel modules and many other applications.

**Java implementation-**

Output:



1. **AVL tree**

**Concept-**

AVL trees are self-balancing binary search tree. It uses a balancing factor to keep a check on its height in such a way that height of two child subtree of a node cannot differ by more than one.

**Operations-**

Insertion: We insert new element in the same way as the binary search tree. This may cause the balance factor of the node to exceed 1 or -1, so in that case we will rearrange the tree in such a way that the balance factor of each node does not exceed 1 or -1. For this we’ll have to check from the root of the tree to node we just inserted.

Deletion: We delete the element in the same way as we do in binary search tree. We search the element and then remove it. This may disturb the balancing factor of nodes in the tree, i.e (between 1 and -1). So for this we’ll have to go from the root to the node we just deleted to check for balancing factor and rearrange the tree accordingly to maintain the balance factor.

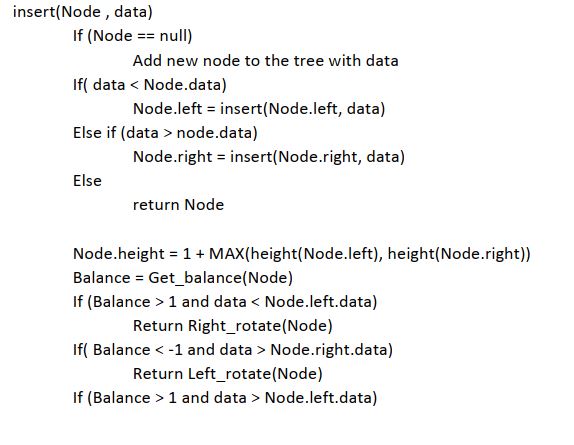
Searching- Searching is exactly same as in binary search tree. We just find it and return the element and no changes are done.

**Pseudocode-**

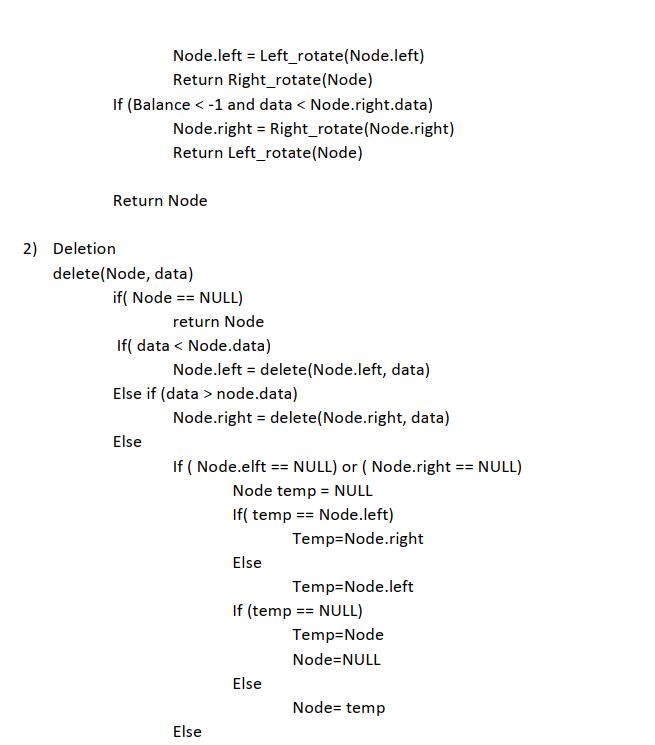
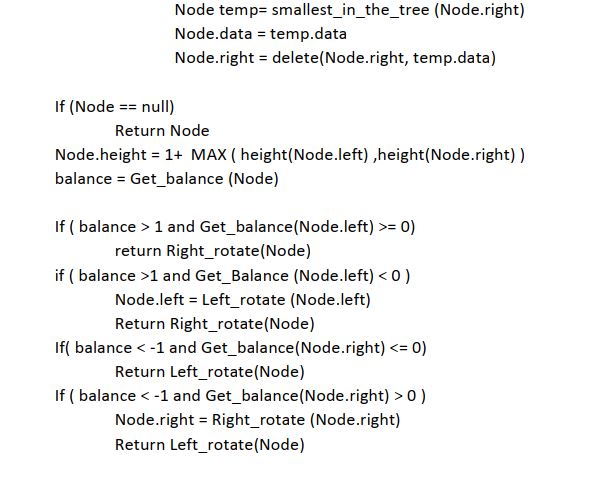
There are some helper functions required in insertion and deletion process for the rotation of the node.

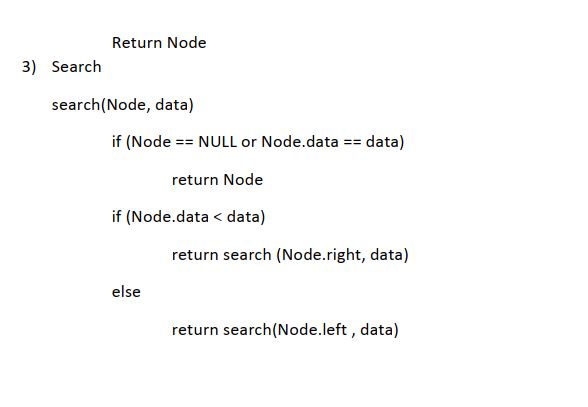
Right\_rotate(Node x) – this is done when the right node causes the imbalance to the AVL tree

Left\_rotate(Node x) -this is done when the left node causes the imbalance to the AVL tree

Get\_balance(Node x)- this function helps in measuring the balance factor of the node i.e difference between the height of the left and right subtree

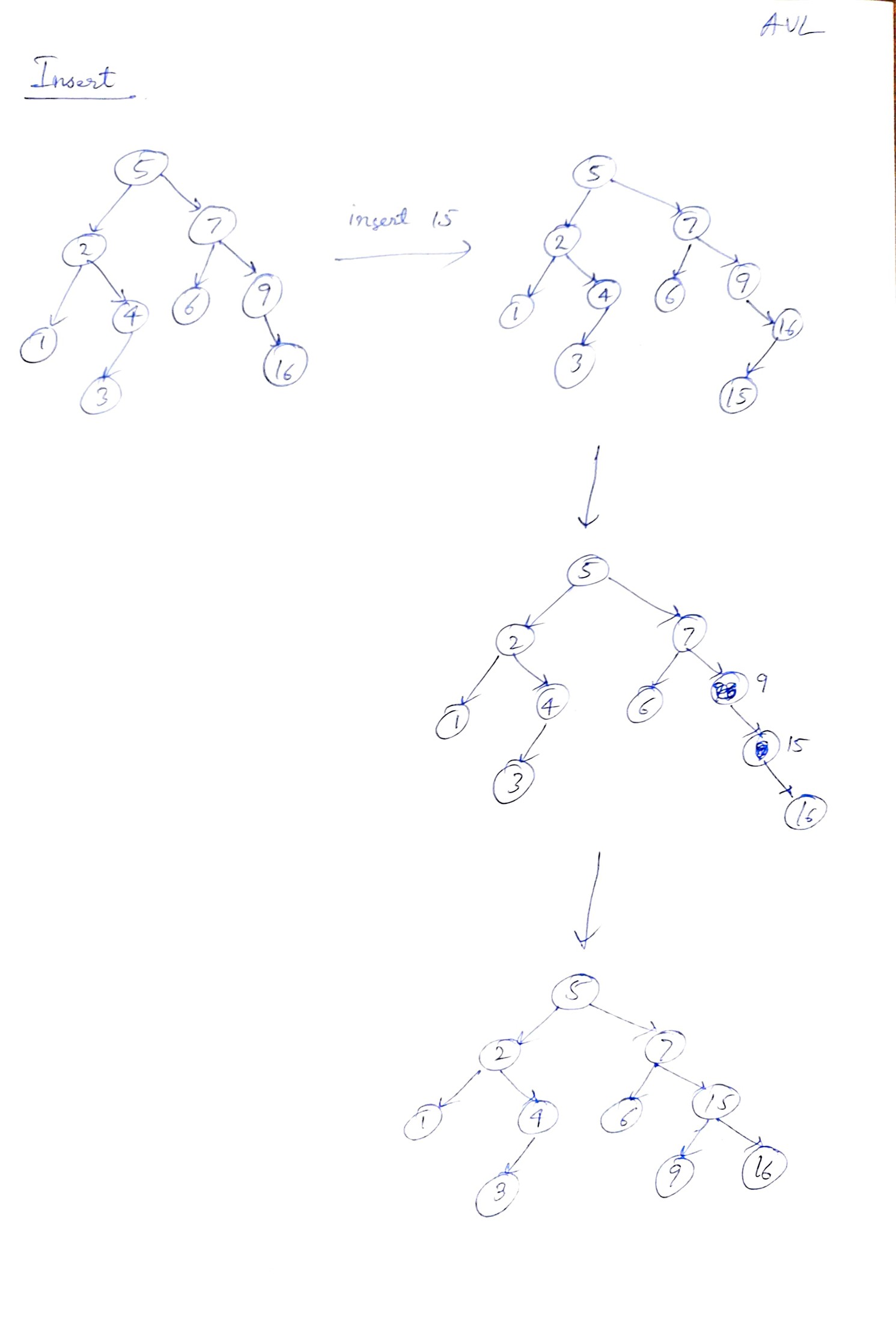
1. Insertion-



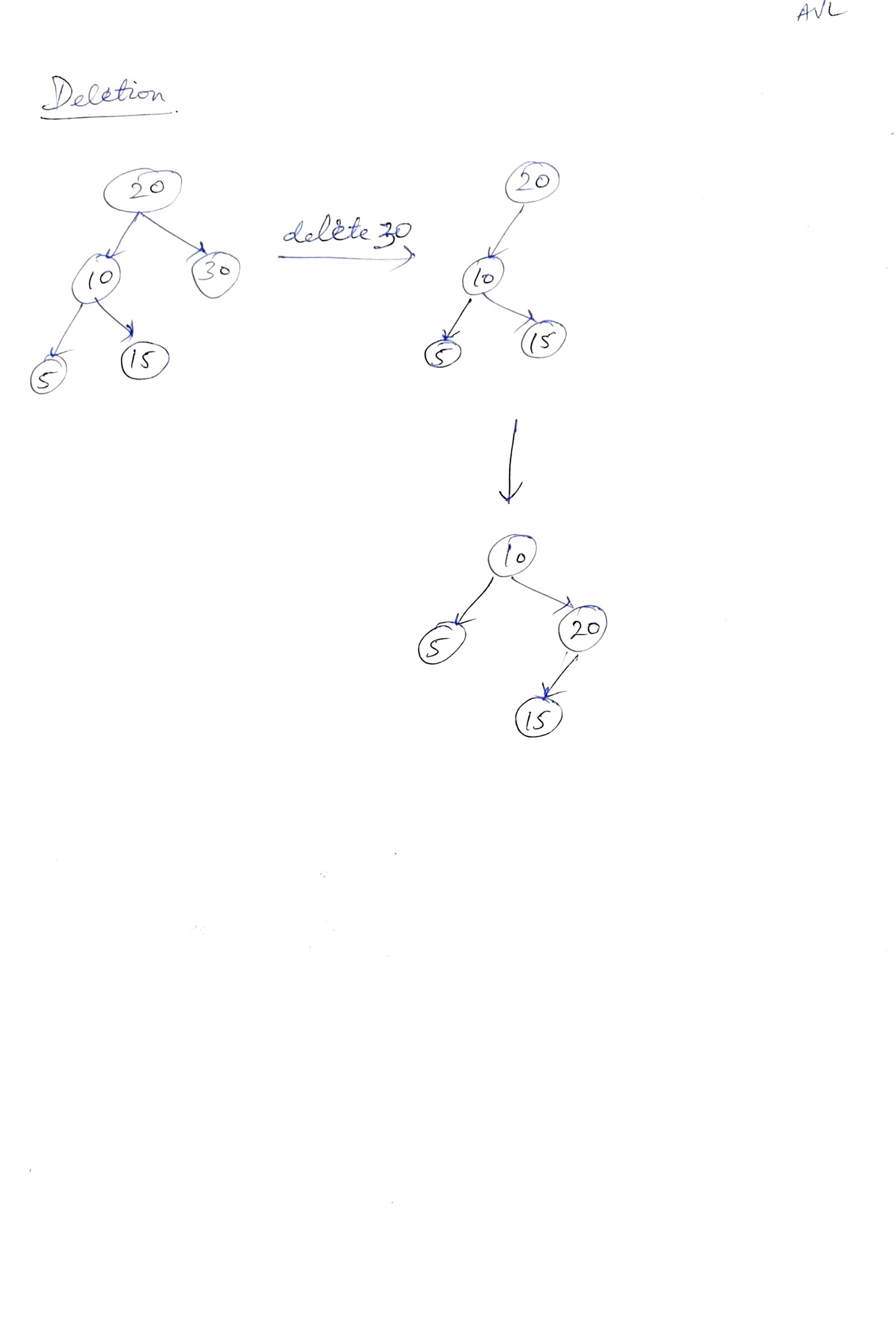


**Example –**

Insertion-



Deletion-



**Time complexity analysis-**

Insertion, Deletion and Searching all have the average and worst case time complexity is O (log n) as the AVL trees are balanced trees and traversal in average and worst case would be equal.

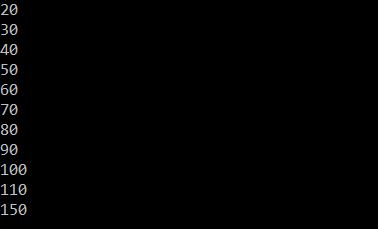
**Notable Special Properties**-

1. All the operation takes O(log n) time complexity in average and worst case.
2. Height of two child subtree cannot differ by more than 1

**Practical application**

1. It can be used in places where insertion and deletion isn’t done very often but searching of data is done always. Hence places like train ticket booking where new trains aren’t added daily but the same trains are accessed daily.
2. In the smart school system where same materials are accessed every day in different classrooms.

**Java Implementation-**



1. **B-tree**

**Concept-**

A B tree is a multi way tree which can be of ‘m’ order having m-1 keys and m children. The purpose of using it is to store many keys in a single node although keeping the height of the tree small.

**Operation-**

Insertion-

A new node is added to the leaf level of the tree. First we traverse the tree to find an appropriate leaf node. Then we check if the leaf node has less than m-1 keys, if yes then we insert the element in ascending order. Otherwise if leaf node has more than m-1 keys then we add the node in ascending order and split the node into two nodes at median and push the median upto its parent node. In case the parent node has m-1 keys then we split it too doing the same steps.

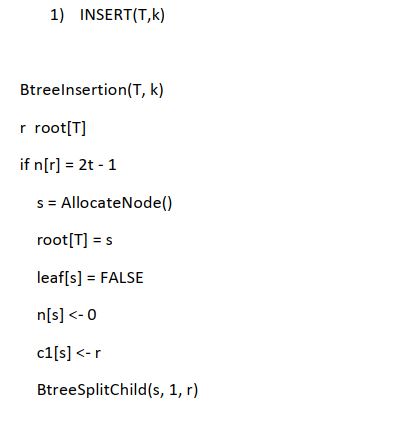
Deletion-

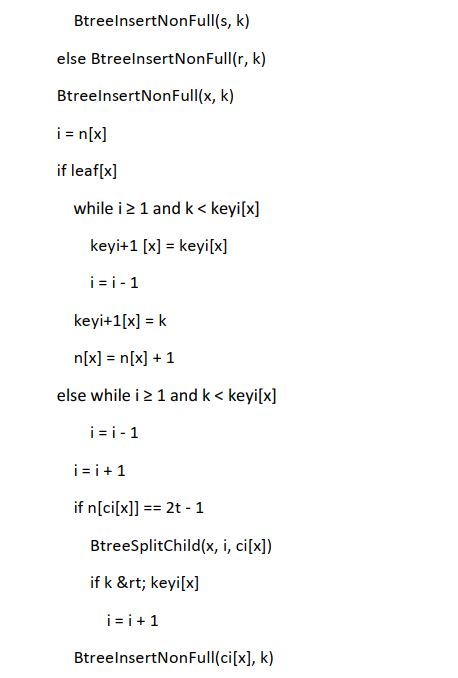
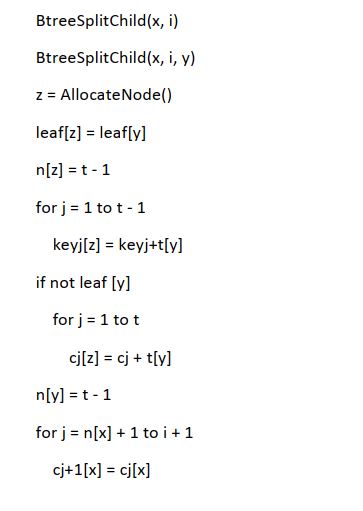
Deletion is done by first searching the element to be deleted. So it can be either a leaf node or a internal node. If it is a leaf node then locate it and if there are more than m/2 keys in the leaf node then delete that key from the node. Now if it doesn’t have m/2 keys then take the element from right or left sibling of the node. Do this until the parents are left with m.2 nodes an then apply the same process.

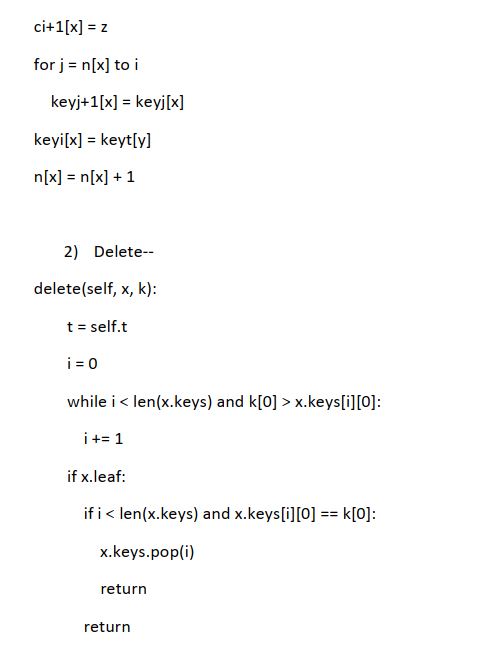
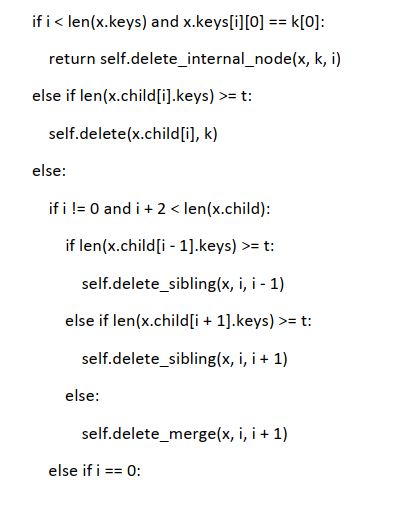
If the node to be deleted is an internal node then replace with its in-order successor or predecessor. As the successor or predecessor will be on leaf node so delete it using the previous steps of deleting from leaf node.

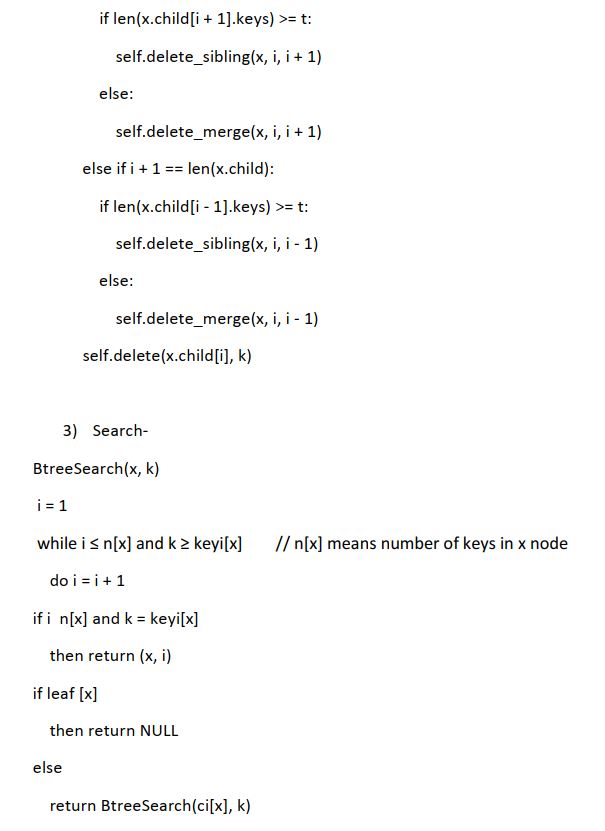
Search-

Search operation is typical to any search operation by traversing throughout the nodes and keys and when it matches return the value.

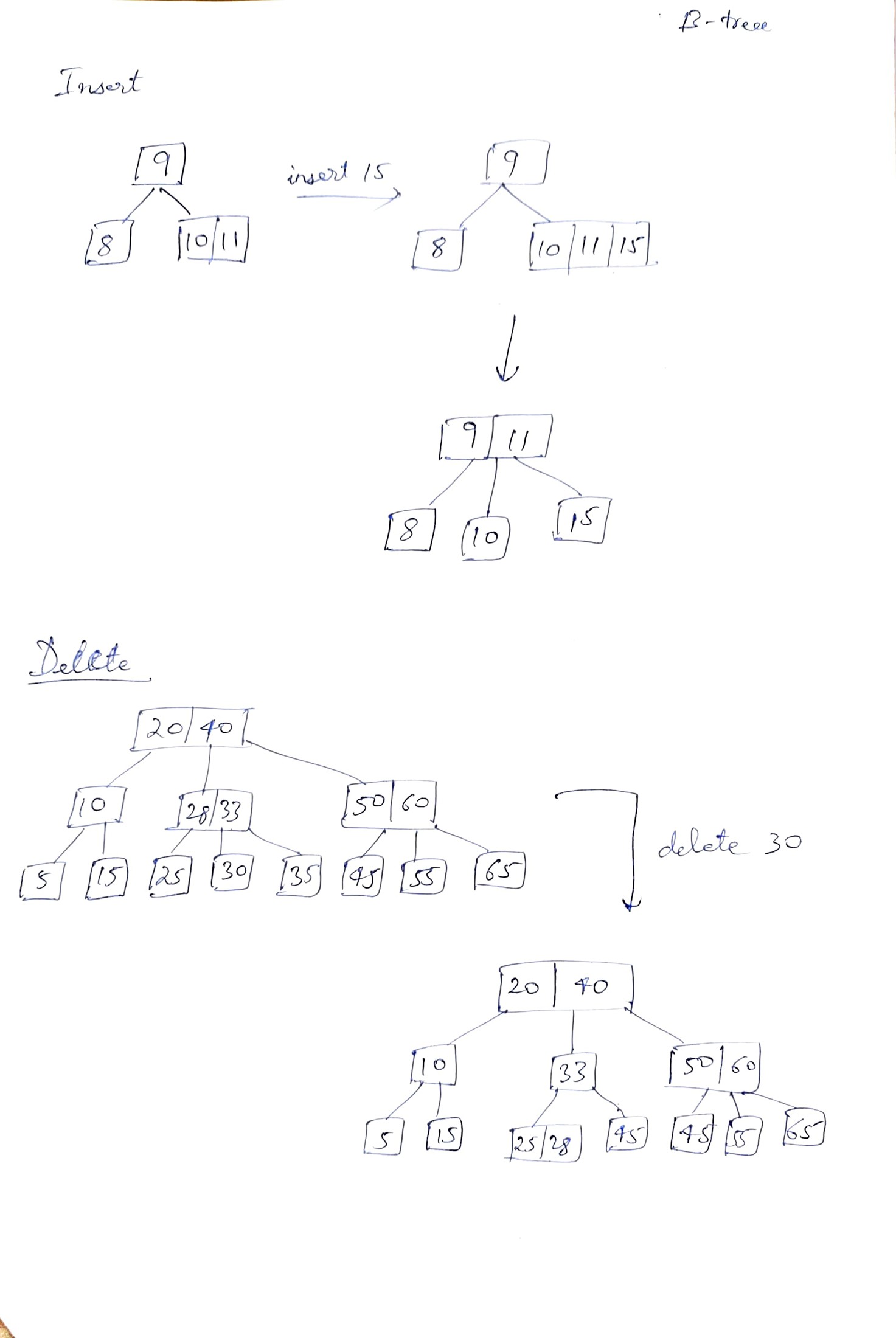
**Pseudo code-**

****

****

****

**Example-**



**Time Complexity-**

Time complexity of all the operation is O(log n) as ata each level, the search is optimised as if the key value is not present in the range of parent then the key is present in another branch. The algorithm traverse till the leaf hence the complexity comes out to be O(log n)

**Notable special Properties-**

1. All leaves are at same level
2. All nodes contain at most m-1 keys
3. Time complexity of all the operation on B-tree is O(log n )
4. All keys are of the node is sorted in ascending order

**Application**

1) It is used in file systems and database where there are lot of data and the need for quick access.

2) It is used as blocks of data in different storage spaces

3) We can process multilevel indexing with B-tree.

**Java Implementation-**

1. **2-3-4 trees**

**Concept –**

2-3-4 trees are balanced search tree. It can have different type of nodes such as 2-node with 1 key and 2 child nodes , 3 node with 2 keys and 3 child nodes and 4 node with 3 keys and 4 child nodes. The purpose of these type of nodes is to balance the tree and perform different operations while allowing multiple keys.

**Operations**

Insertion-

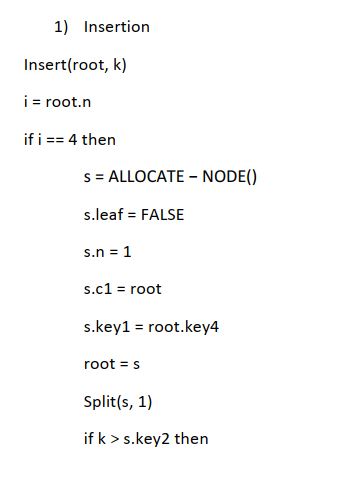
New node is inserted at the leaf irrespective of whether there is space in some internal node. In 2-node on addition it becomes 3-node and 3-node on addition become 4-node. When the node is already full then we split the nodes before adding the node to the smaller node. This splitting process can continue until we reach root.

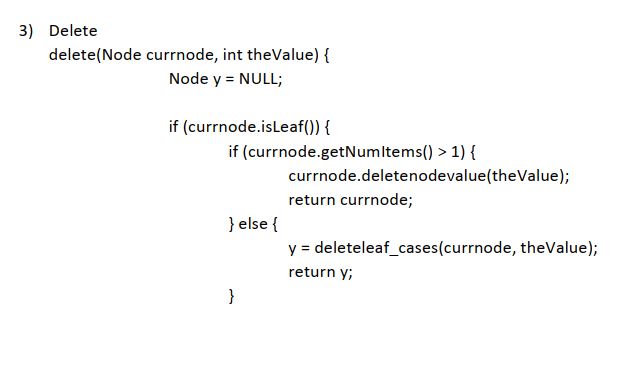
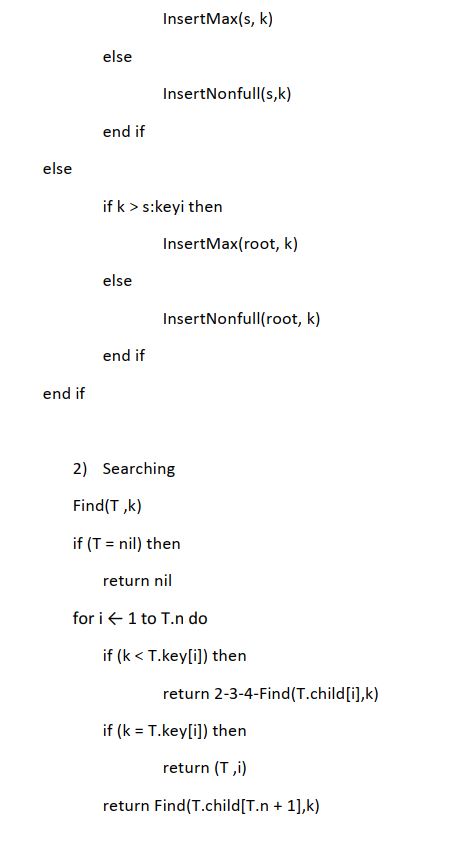
Deletion –

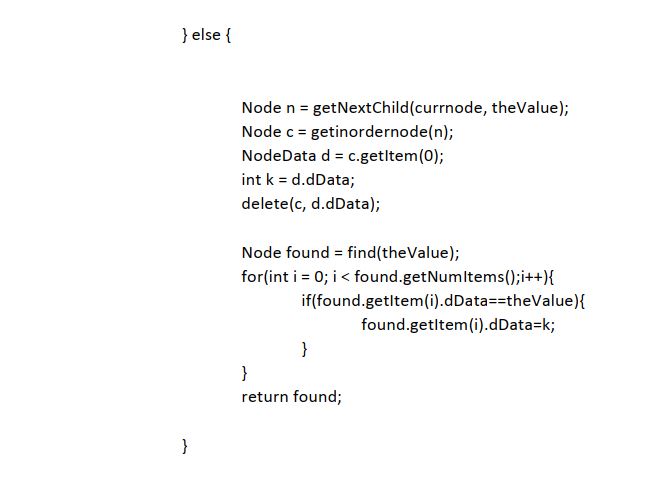
We have to search the element to be deleted. If the element is found in the leaf node then we delete it. If the element if not a leaf node then we just keep a pointer on that node and continue searching for the leaf that might contain element’s predecessor or successor. On finding them, we just swap those two elements.

Searching –

We recursively traverse every nodes to check for the key being searched. In case not found then we return it doesn’t exist.

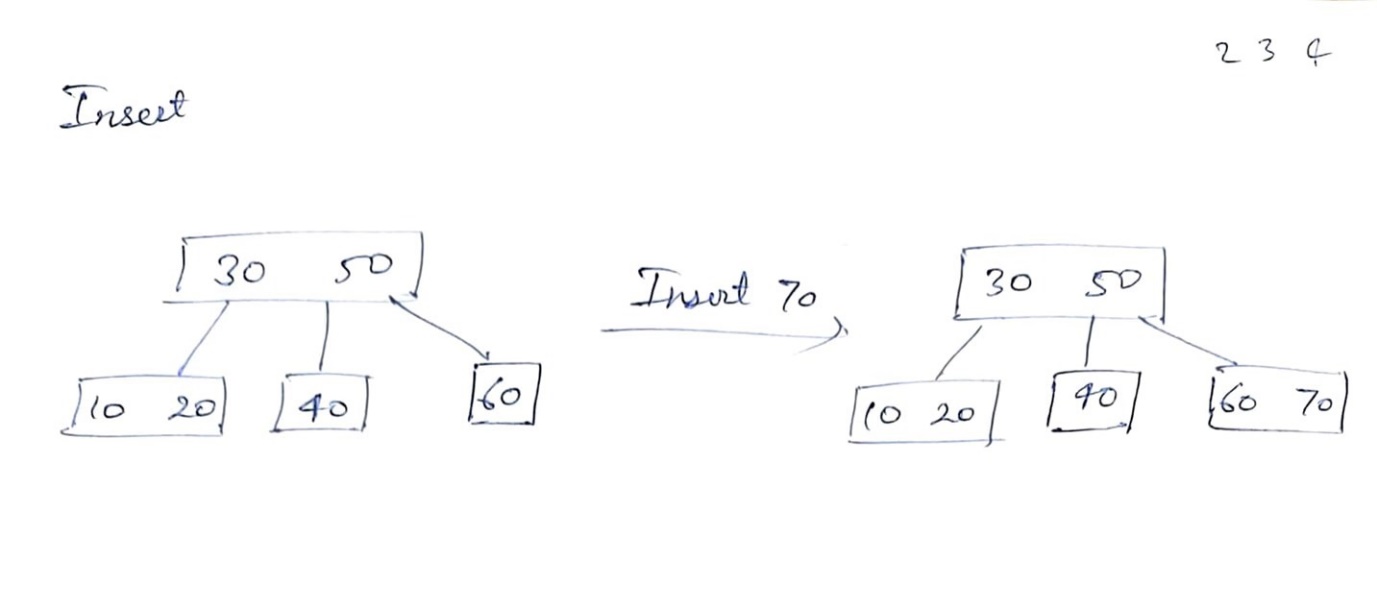
**Pseudocode-**



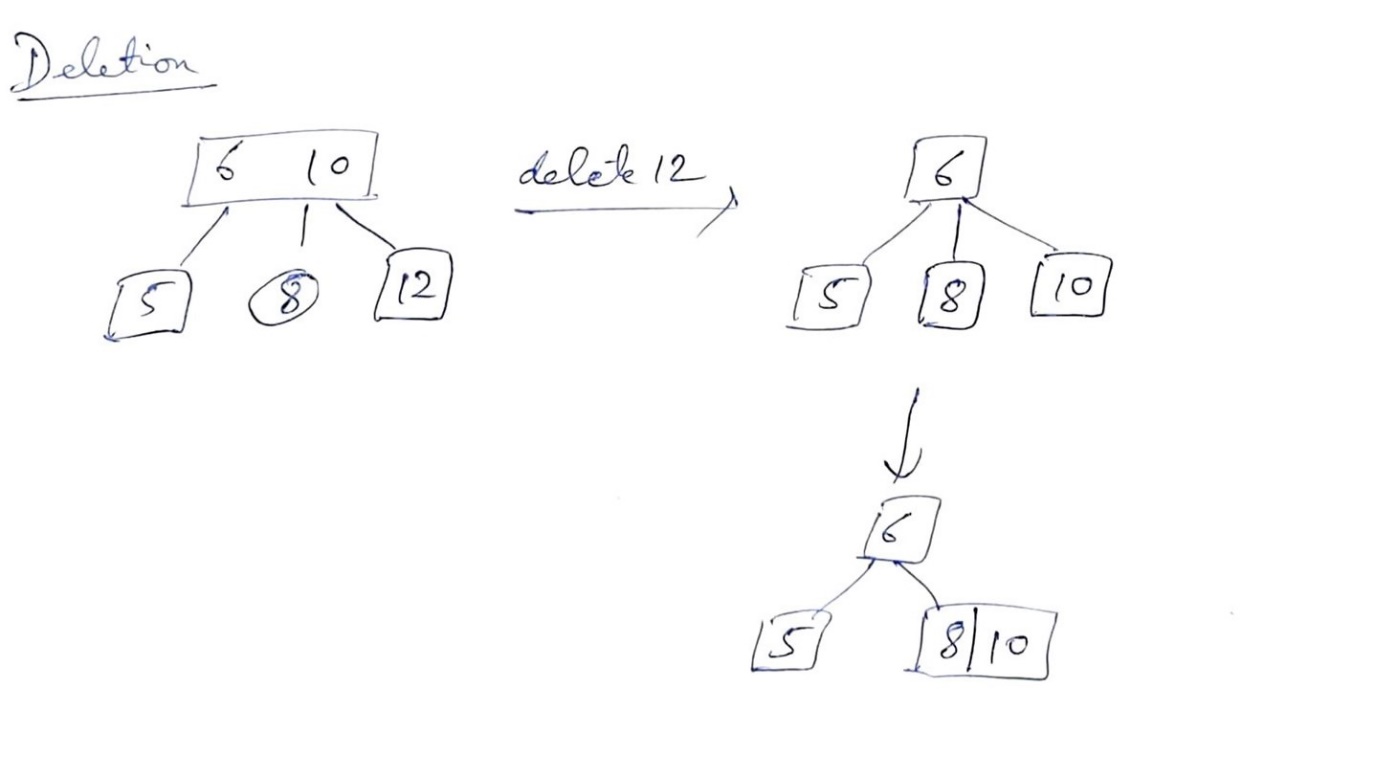
****

**Examples-**

Insertion-



Deletion-



**Time Complexity analysis-**

All the operation in 2-3-4 tree takes O(log n) time complexity as the tree is always balanced. Insertion and deletion process require addition and subtraction of node in the child tree.

**Notable special properties**

1. Each node can store at most 3 values arrange in ascending order.
2. The tree is balanced so leaf nodes are at equal level.
3. The time complexity for the all operations is O(log n)

**Practical application**

1. It can be used for implementing dictionary
2. To keep track of Virtual Memory Segments of a Process.
3. It is used in Linux kernel in different operation where large memory is used.

**Java Implementation-**



1. **Red- Black trees**

**Concept-**

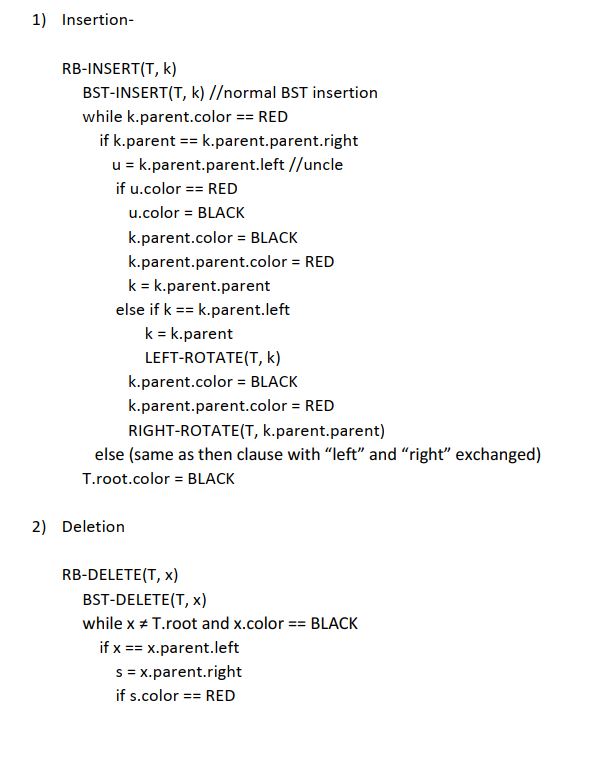
Red Black trees are basically binary search tree with property of self balancing after every operation. This self balancing is done using color scheme which are red and black. A colour code is added to each node so that using that balancing can be done.

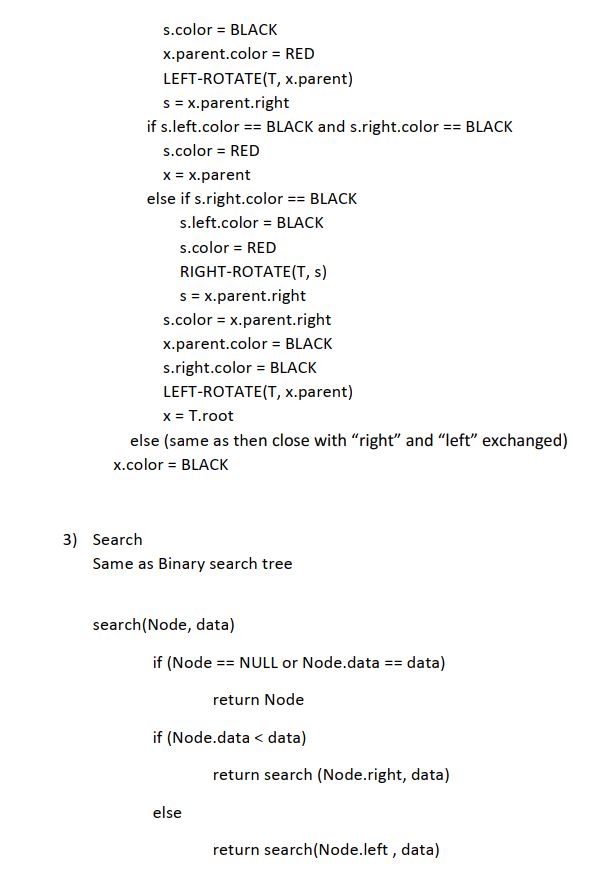
**Operations-**

Insertion- First we insert node like insertion in usual binary search tree. We use color code on the nodes to help self balance it. Every new node is color coded as red. We do rotation based on properties of red black trees such that the tree is balanced.

Deletion- In Deletion, we first search the node like usual binary search tree. After finding the node, we delete it but then we also have to balance the red black tree based on its colour constraints. We do rotation based on properties of red black tree such that tree is balanced.

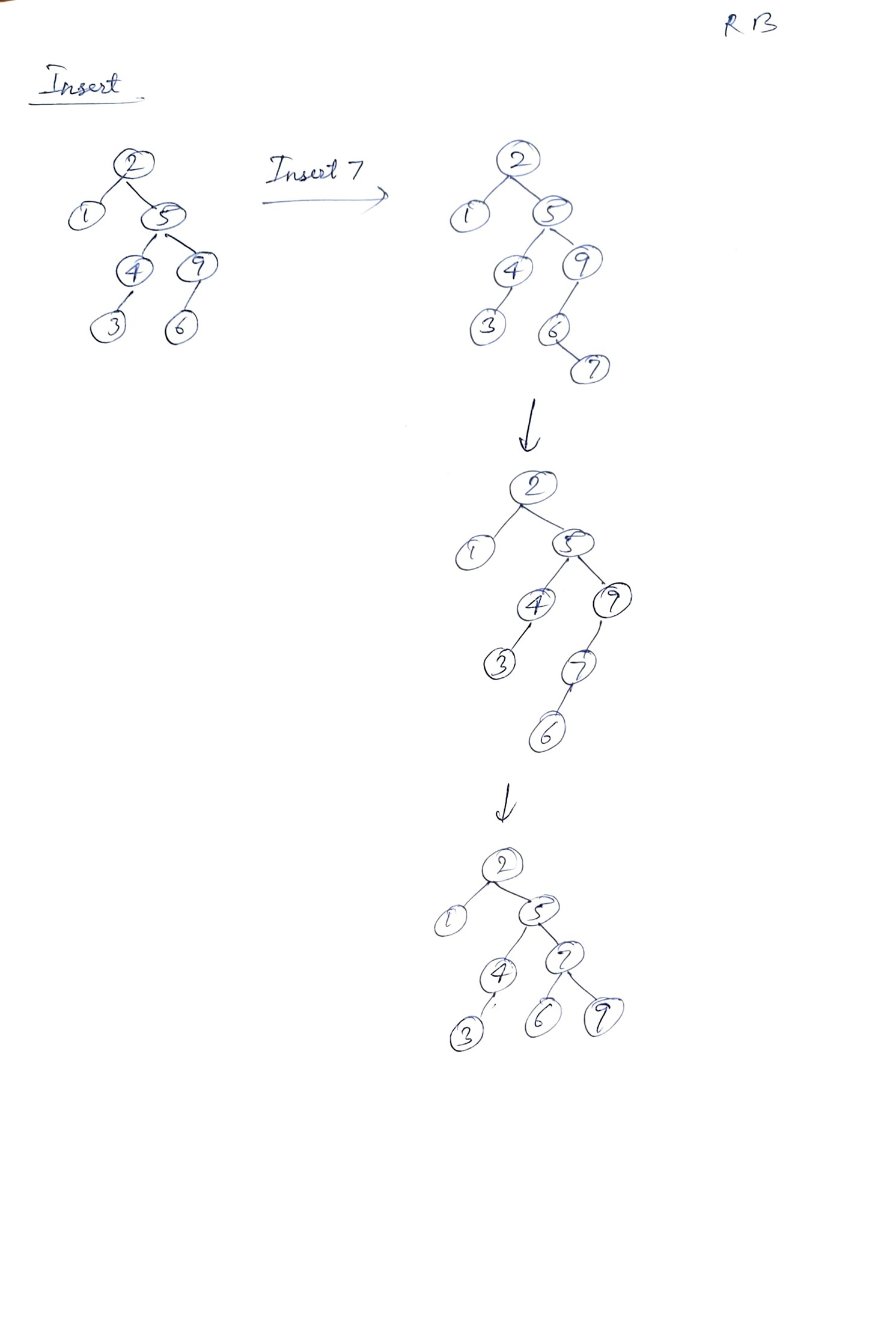
Search- Searching is typical to search operation on binary search tree

**Pseudocode-**

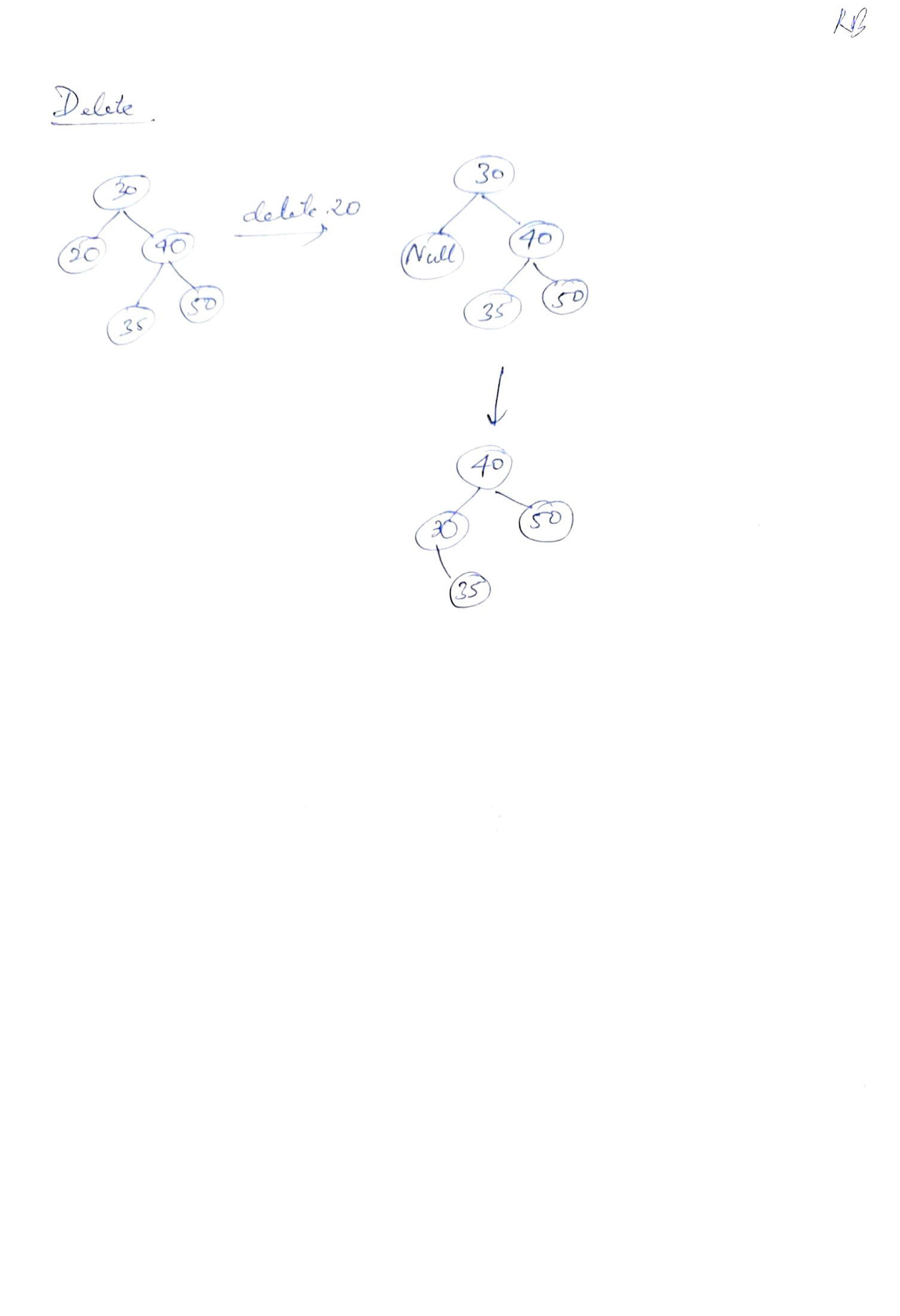
****

**Example-**

Insertion-

****

Deletion-

****

**Time Complexity analysis-**

All the operation of insertion, deletion and searching is of the time complexity O(log n). The red, black tree self-balances itself after every operation which causes the rearrangement of the nodes in the tree to be balanced. the path from the root to the farthest leaf is no more than twice as long as the path from the root to the nearest leaf. Hence it is height balanced making the operation of time complexity of O(log n).

**Notable special properties-**

The time complexity of all the operations is O(log n)

Every node of the tree is either red or black in colour

The main root node should always be black

A red node can’t have red node as parent or child

The path from node to any leaf has same number of black nodes

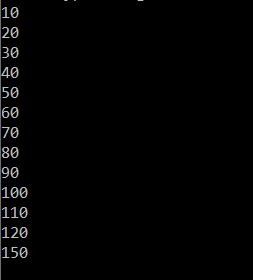
**Practical application-**

It is used in the linux kernels where virtual runtimes serve as the key.

It can be used in maps where routes are to be determined.

It can be used in K mean clustering algo to reduce time complexity.

It is used in MySQL for indexing the tables

**Java Implementation**

1. **Scapegoat tree**

**Concept-**

Scapegoat trees are self balancing binary search tree with the concept of maintaining size balance in the nodes. The concept is that when the tree is unbalanced, we identify the “scapegoat” node which has caused it and rebalance it.

**Operations-**

Insertion-

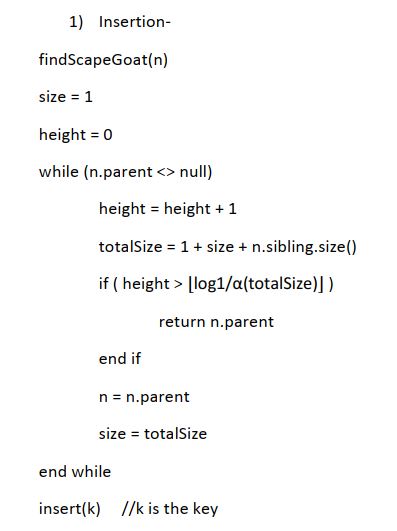
We insert the node like insertion in binary search tree. Once it is inserted, it checks if the tree is balanced or not. If not, then it does it by checking the node’s ancestry until it finds the subtree which is not balanced. It is checked with the weighing property of the scapegoat algorithm. With this it finally rebalanced the whole tree.

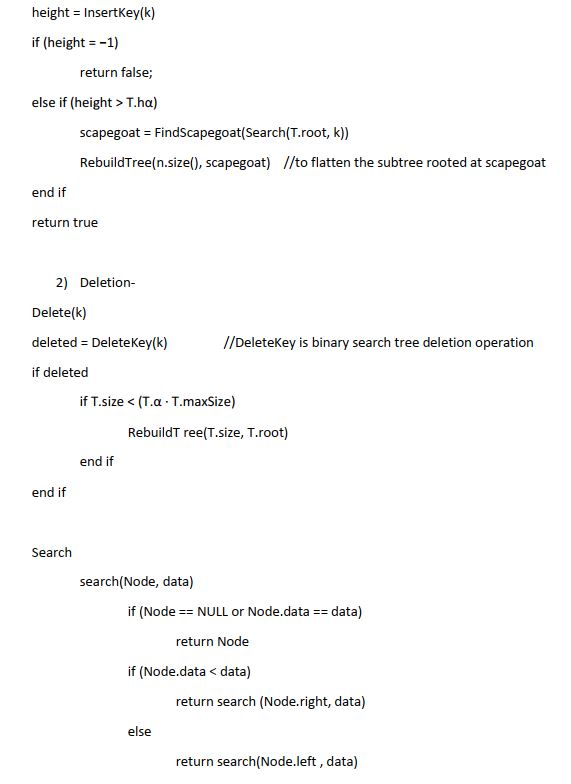
Deletion-

Deletion is also similar to deletion in binary search tree. It searches for the node and then deletes. After deleting it look up through ancestry to find unbalanced subtree and using the weighing property of the scapegoat algorithm it rebalanced the the tree.

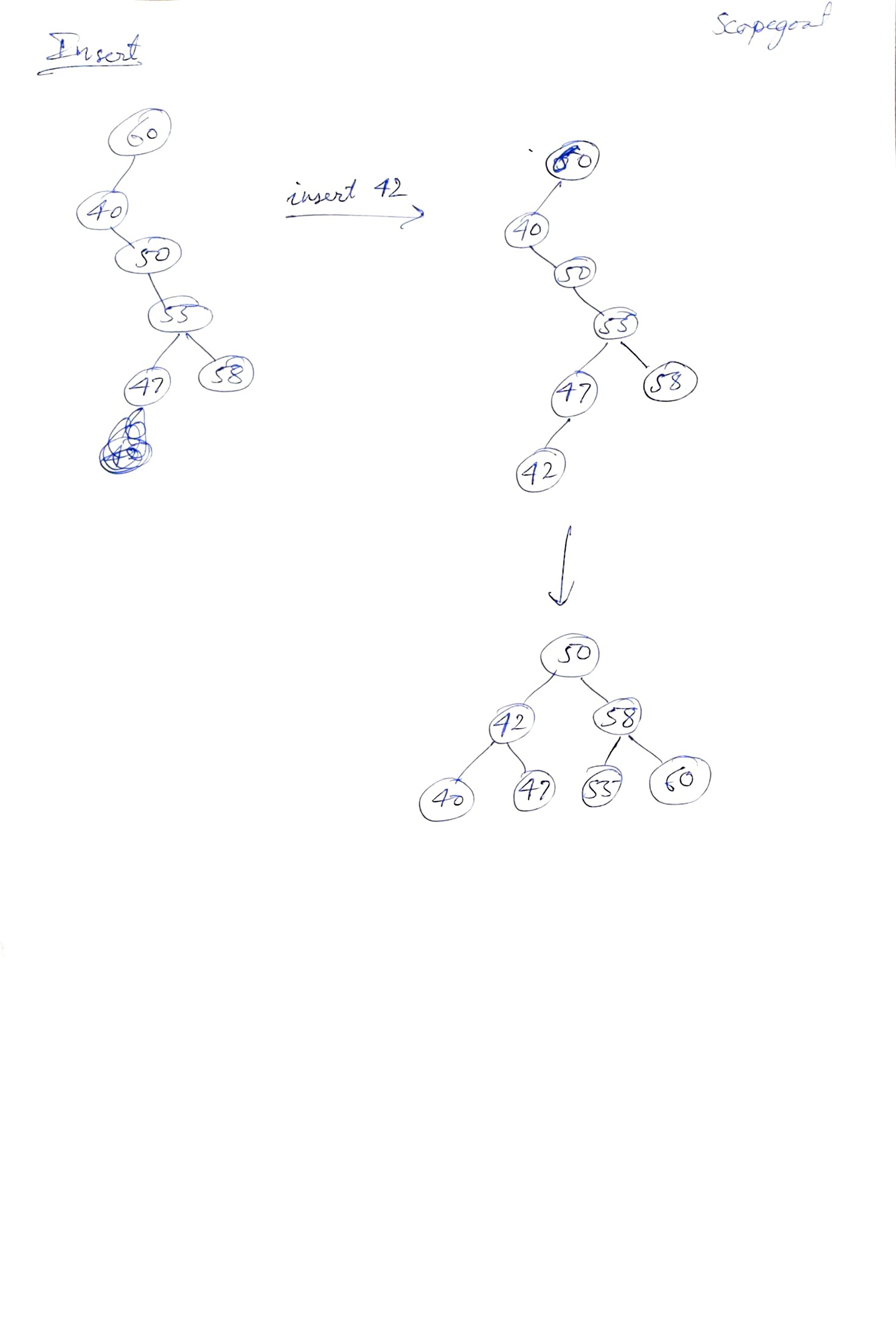
Searching-

It is similar to the searching operation in binary search tree.

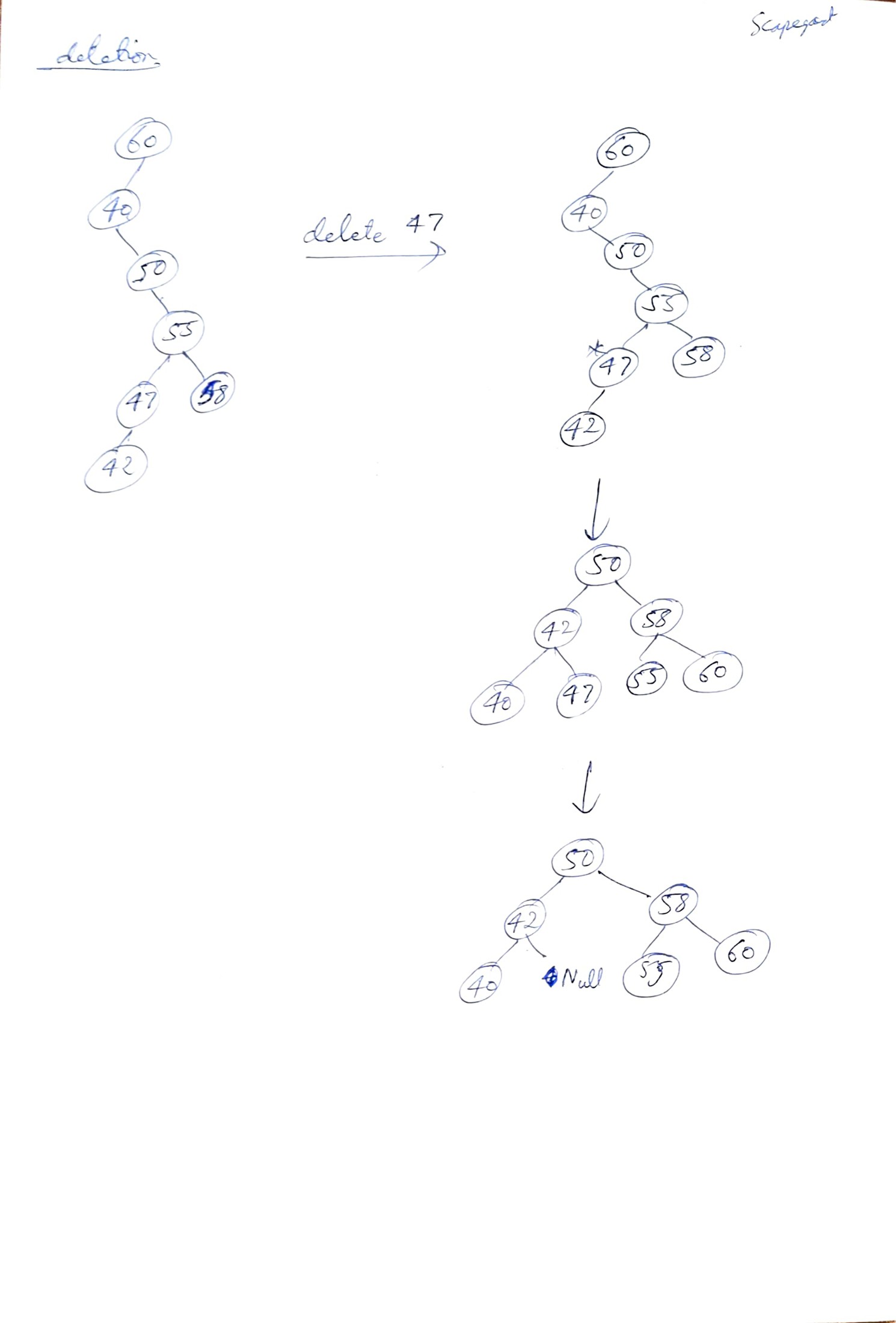
**Pseudocode-**



**Example-**

Insertion

Deletion-



**Time Complexity analysis-**

The time complexity of all the operations i.e. insertion, deletion and searching is O(log n) as climbing up and finding the scapegoat node and balancing the tree causes the complexity to be O (log n).

**Notable special properties-**

1. Time complexity is O (log n) for all the operation
2. Scapegoat tree does not require extra data in each node for self-balancing
3. It identifies a scapegoat node and then rebalances the tree

**Practical uses**

1. It can be used in systems with low memory as searching and other operation consume less space
2. It can be used in book acceptance systems in libraries as it is efficient insertion operation and consumes less memory.

**Java Implementation-**

