gradeup

Data Structure Part-3

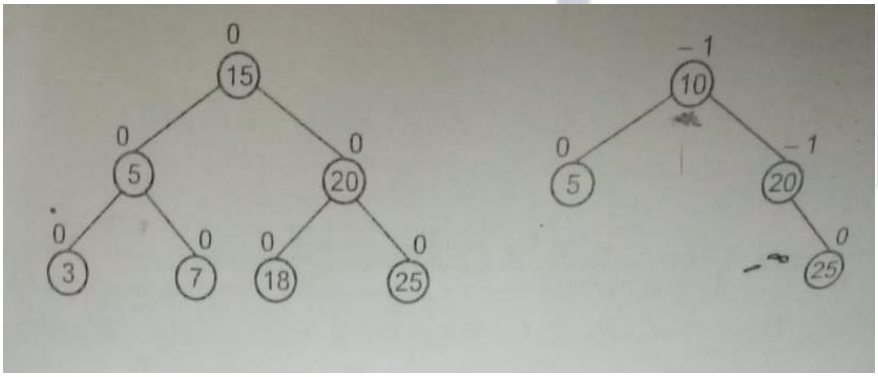
Data Structure Part-3

Content:

AVL Tree

- An empty binary tree is an AVL tree.
- If B is the non-empty binary tree with B_L and B_R are its left and right sub trees then B is an AVL tree if and only if
  i)B_L and B_R are AVL trees, and
  ii)$|h_L - h_R| < 1$, where h_L and h_R are the heights of B_L and B_R sub trees, respectively.
  Balance Factor: To implement an AVL tree each node must contain a balance factor , which indicates its states of balance relative to its sub trees. If balance is defined by
  (height of left sub trees)-(height of right sub tree)
  Then the balance factors in a balanced tree can only have values of -1, 0 or1. Thus AVL tree may have the following balance factor for a node{-1, 0, 1}. If a node 'x' has balance factor 1 then that means left sub tree is at greater height than the right sub tree, then ti is said to be left balanced. If a node'x' gas balance factor-1 then that means, right sub tree is at greater height than the left sub tree then it is said to be right balanced. For balance factor 0 the tree is said to be completely balanced.

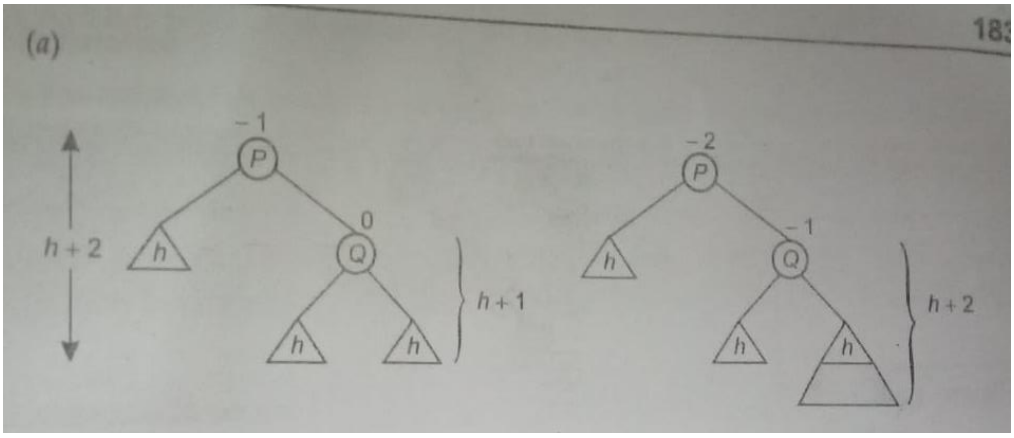
If the binary tree is the binary search then it is AVL search tree if and only if
1.Binary search tree B_L and B_R are AVL search tree where B_L and B_R are left and right sub tree respectively.
2.$|h_L - h_R| < 1$ where h_L and h_R are the heights of left and right sub trees, respectively.



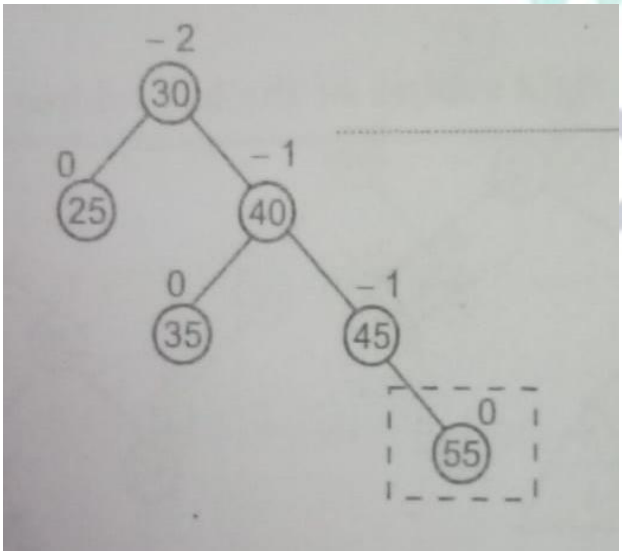
Insertion in an AVL search tree
1.If the data item key 'k' is inserted into an empty AVL search tree, then the node with key 'k' is set to be the root node. In this case, the tree is height balanced.

2.If tree contains only single node,i.e., root node, then the insertion of the node with key 'k' depends upon its value. If the value of 'k' is less than the key value of the root then it is inserted to the left of the root otherwise right of the root. In this case the tree is height balanced.

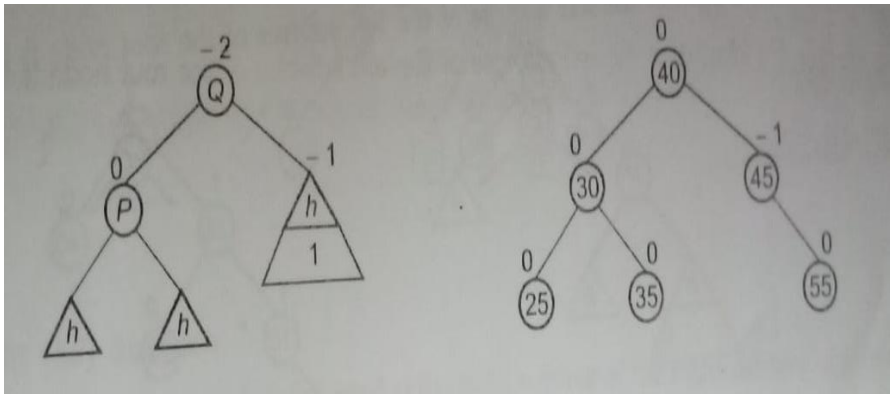
3.If on inserting the node with key 'k' the height of the right sub tree of the root has increased.

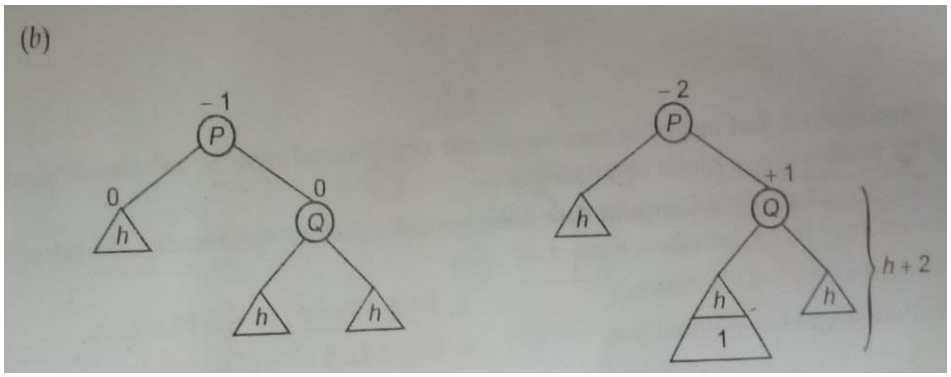


For example, inserting node 55



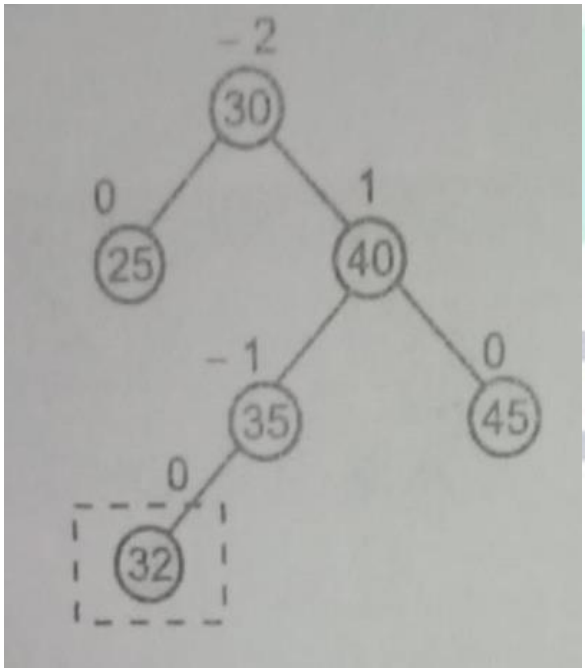
In this case, the problem can be easily rectified by rotating node Q about its parent p, so that the



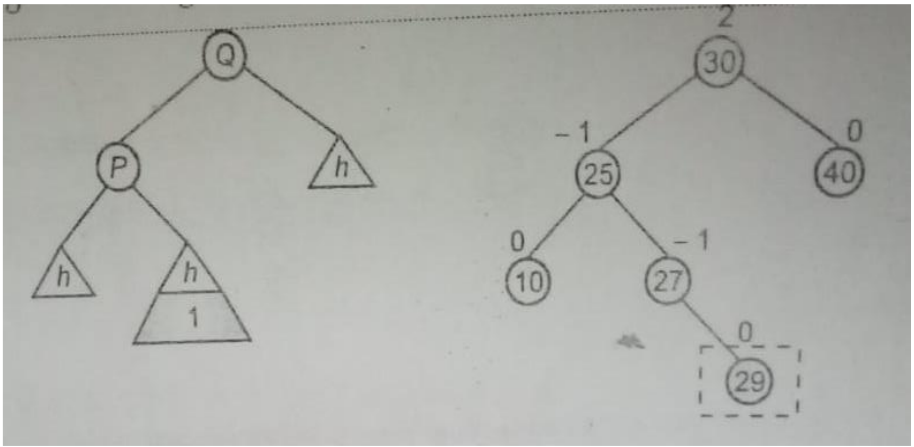


balance factor of both P and Q becomes zero, i.e.

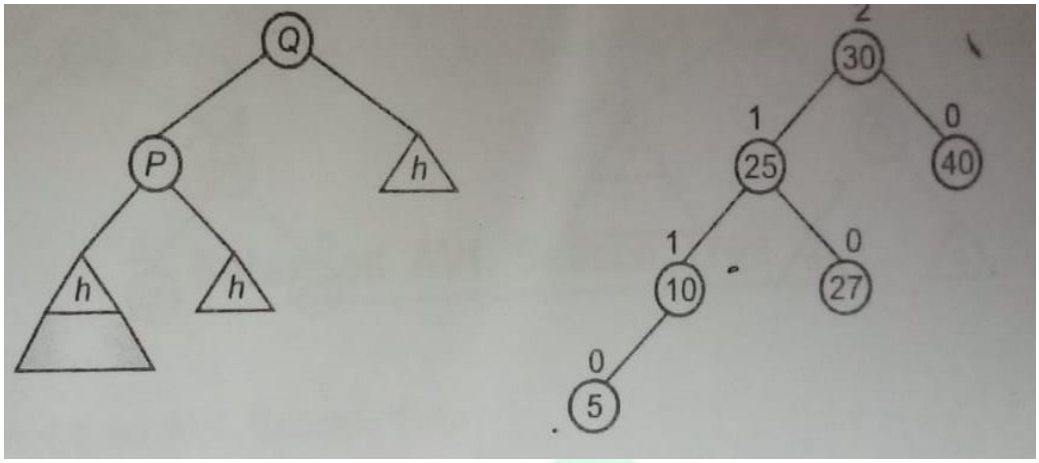
In this case, insert a node in the left sub tree of the right child i.e.



1.If an inserting a node with key 'k' the height of the left sub tree of the root is increased i.e.
a)Height of the right sub tree of the left sub tree of the root node is increased,i.e.



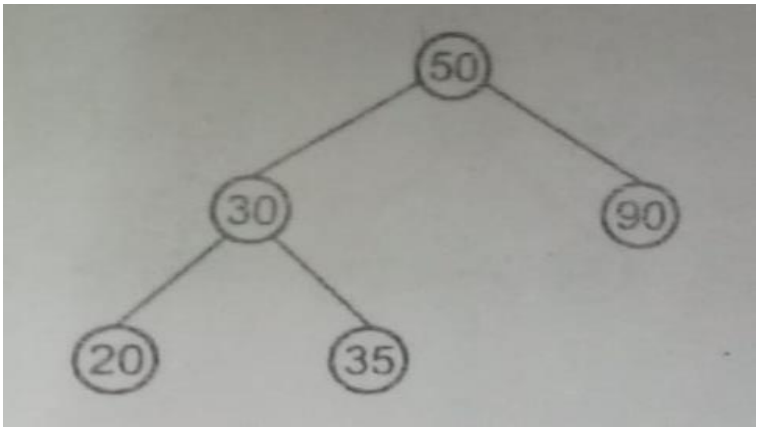
b)Height of the left sub tree of the left sub tree of the root node is increased,i.e.



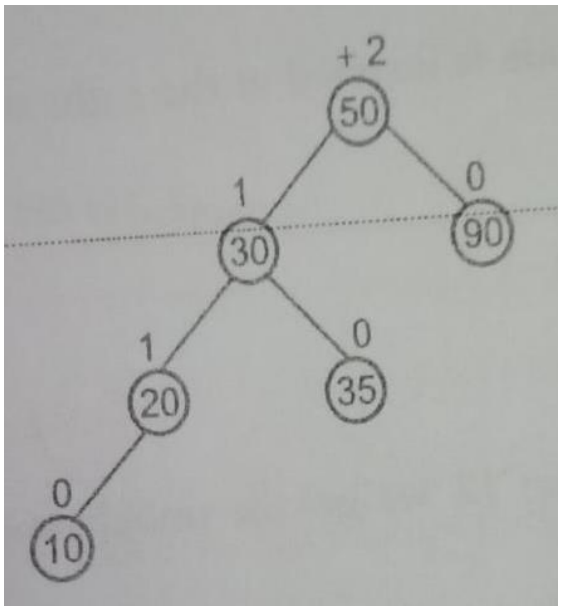
Rotations: The rotations for rebalancing the tree are characterized by the nearest ancestor of inserted node whose balance factor becomes +2

1. Left-left rotation
2. Left-right rotation
3. Right-right rotation
4. Right-left rotation

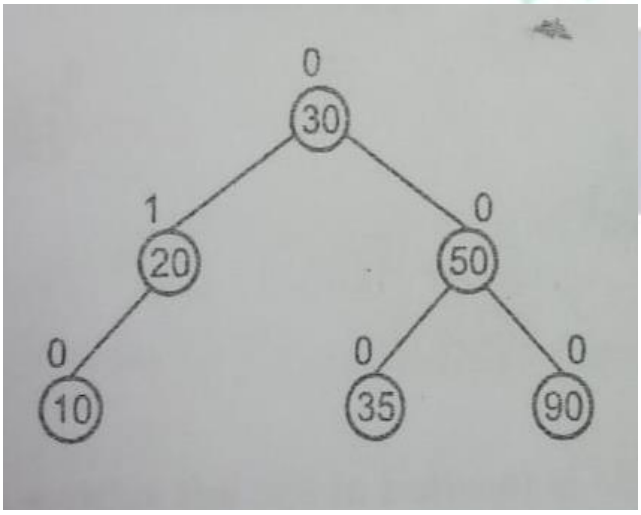
Left-left rotation
In this rotation new node is inserted at the left sub tree of the left sub tree of root.



Insert node 10, we get

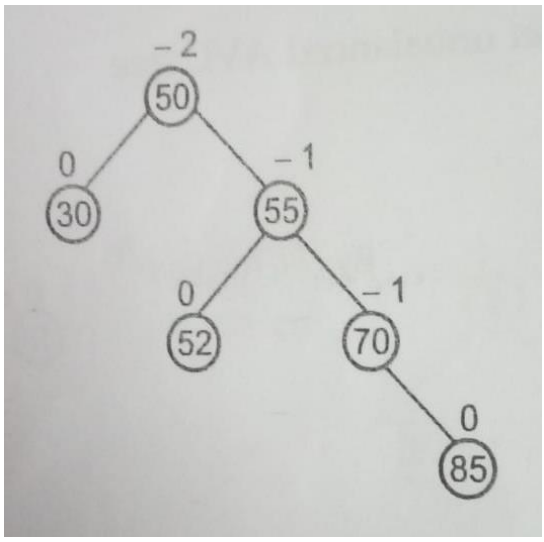


By LL- rotation, we get

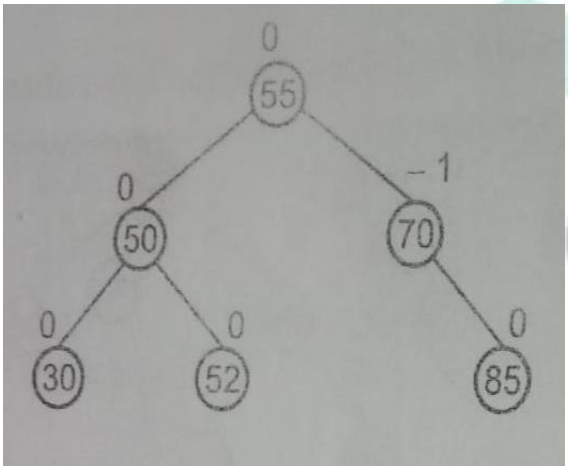


Right- right rotation

In this rotation new node is inserted at the right sub tree of the right sub tree of root.
For example, insert node '85'



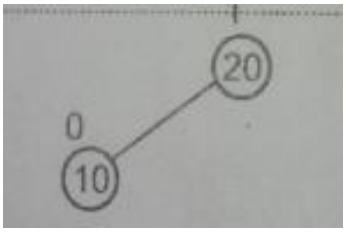
By RR rotation, we get



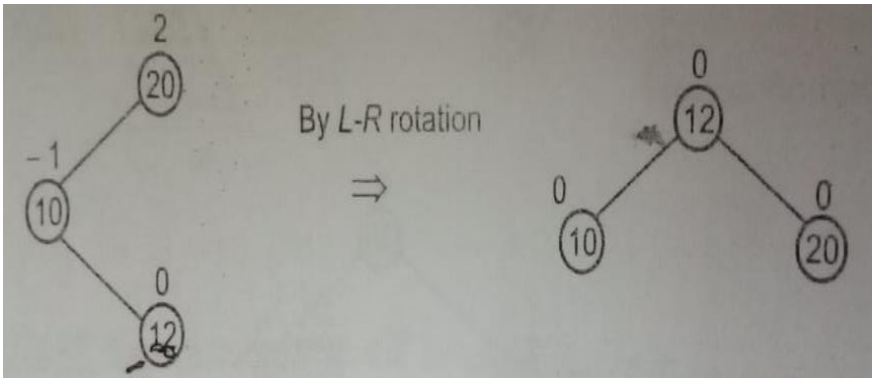
Left-right rotation

In this rotation new node is inserted at the right sub tree of the left sub tree of root.

For example

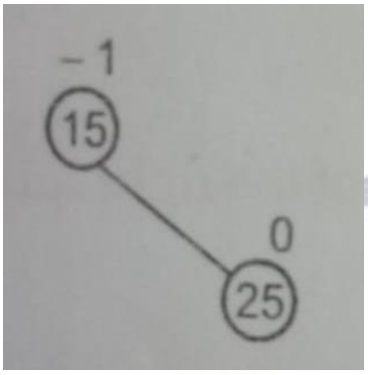


Inserting node with value '12' we get the the unbalanced AVL search tree.

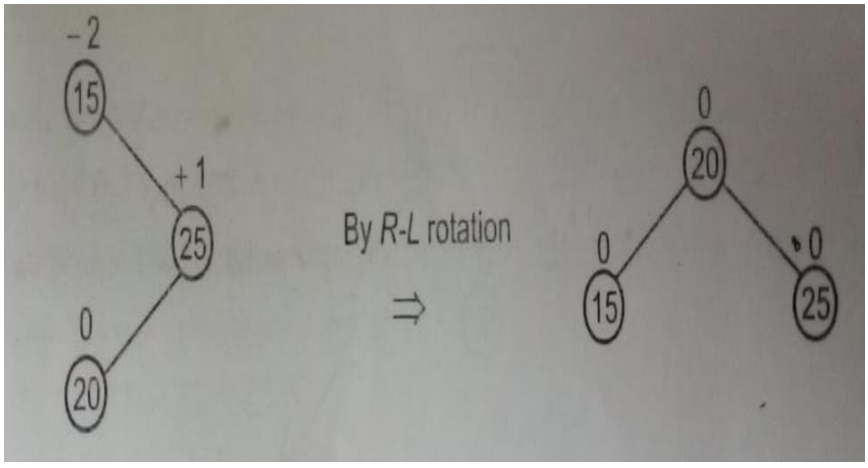


Right-left rotation

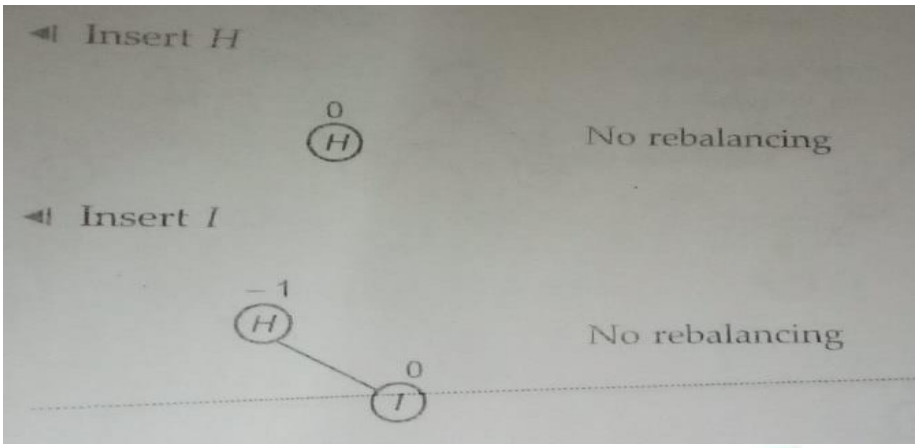
In this rotation new node is inserted at the left sub tree of the right sub tree of root.
For example,

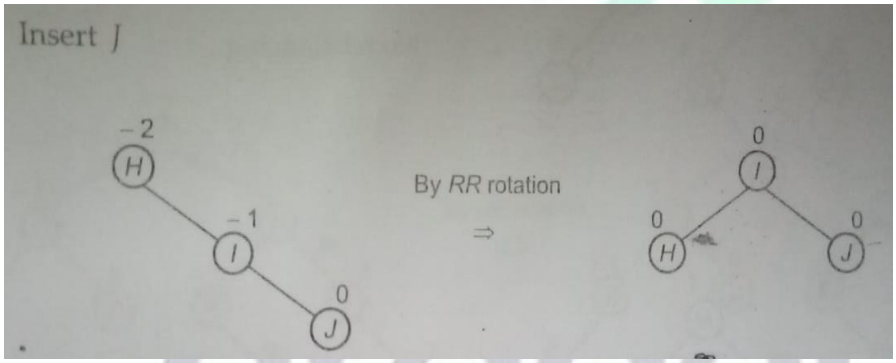


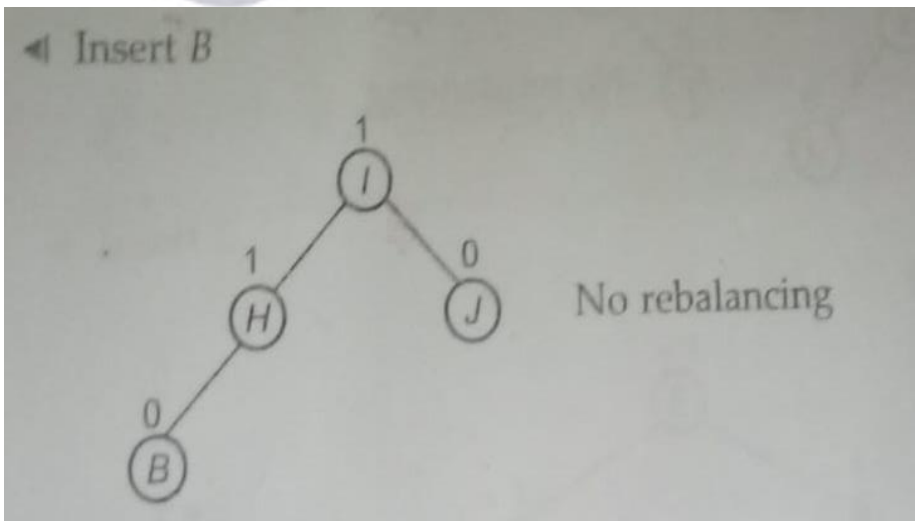
insert the node 20, we get unbalanced AVL tree

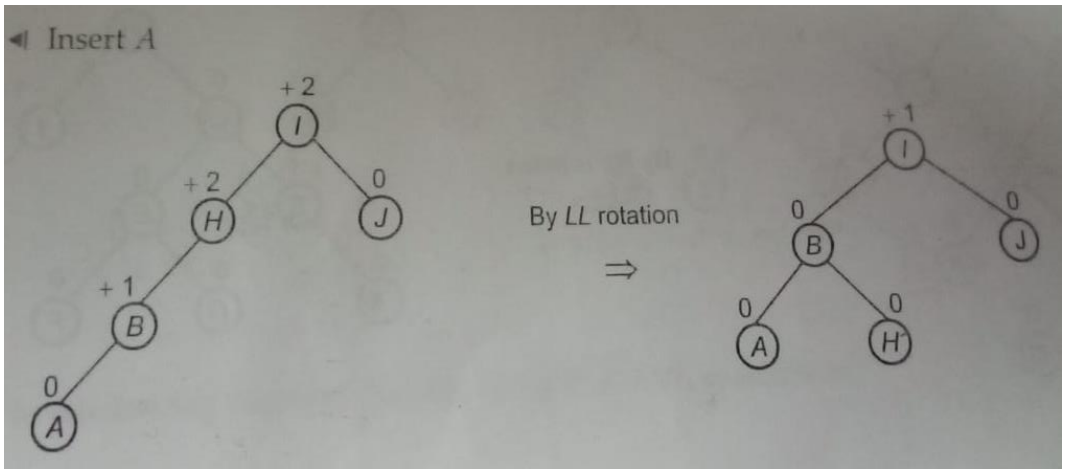


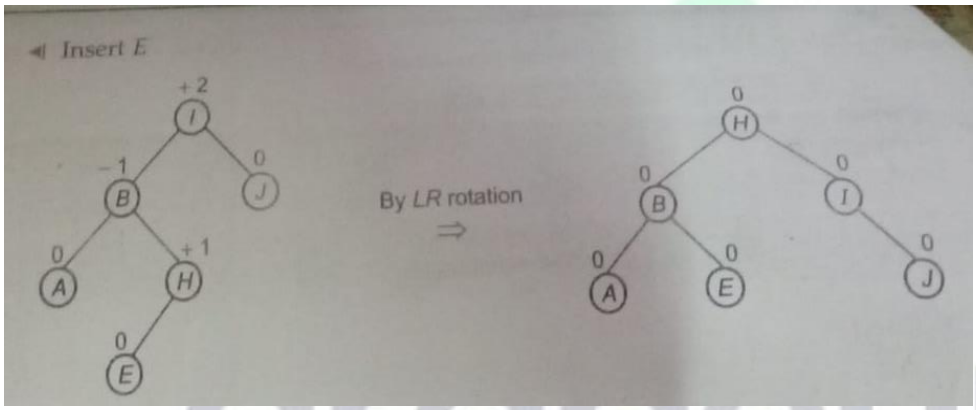
Example: create an AVL search tree from the given set of values:

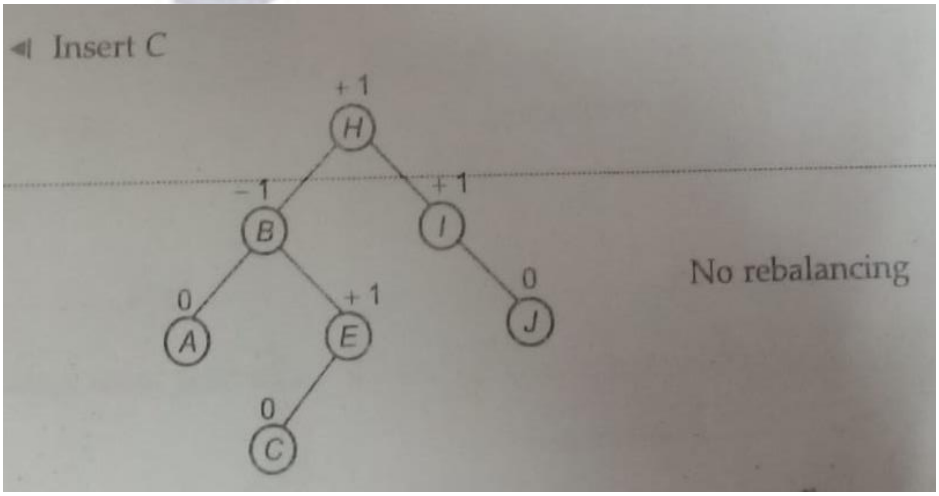


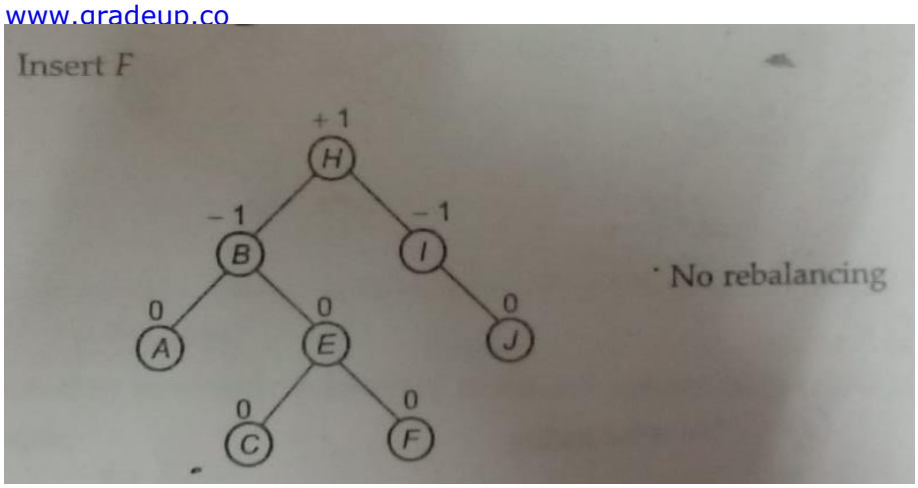


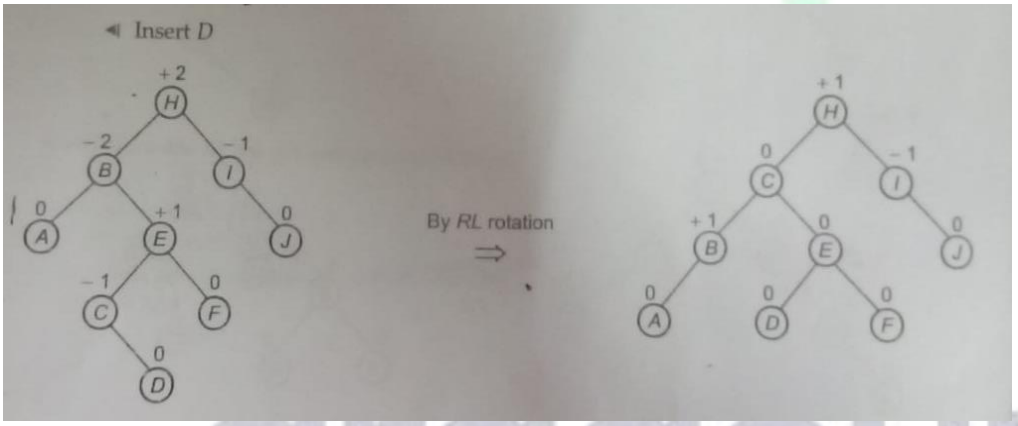


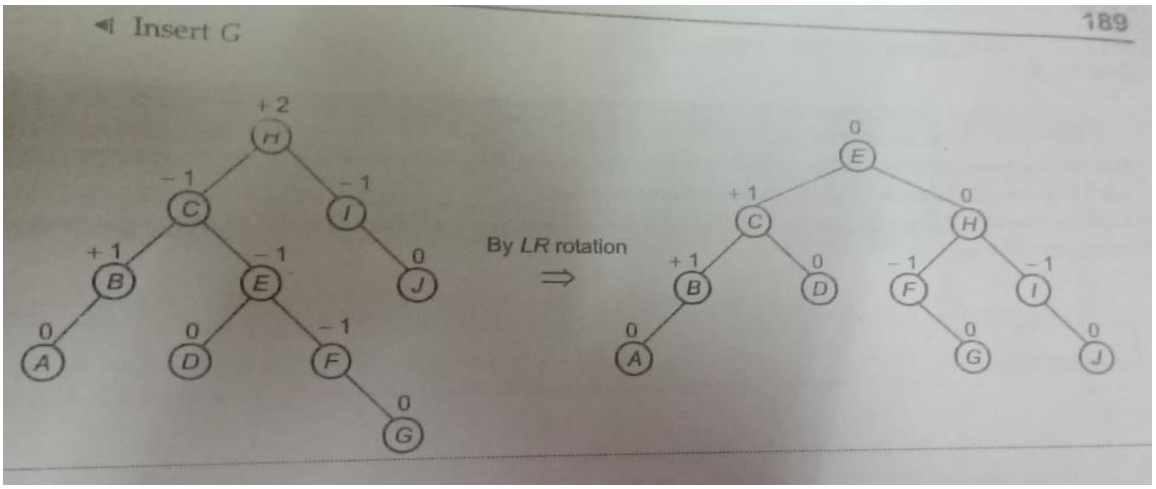

Insert A
+2
I
+2
H
0
J
+1
B
0
A
By LL rotation
⇒
+1
I
0
B
0
J
0
A
0
H

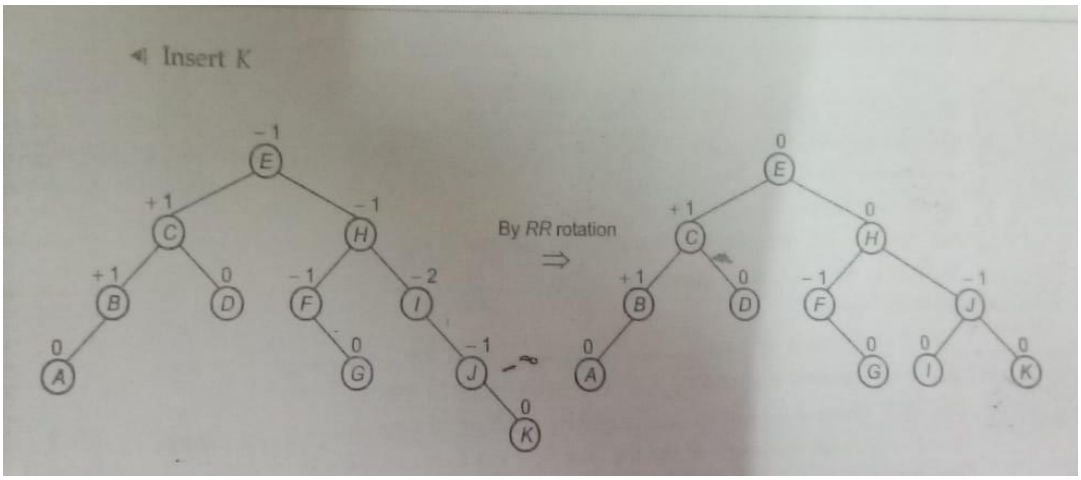

Insert E
+2
I
−1
B
0
J
0
A
+1
H
0
E
By LR rotation
⇒
0
H
0
B
0
I
0
A
0
E
0
J

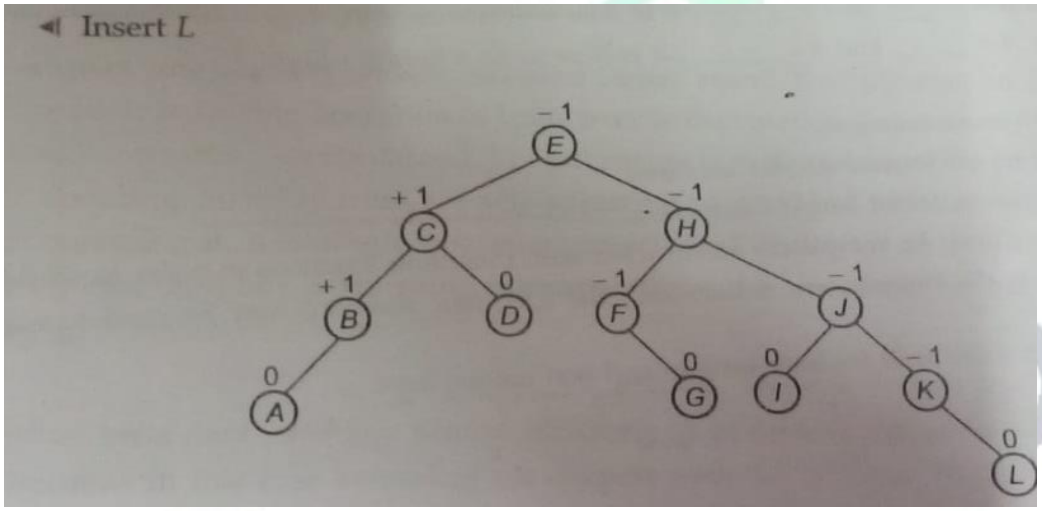

Insert C
+1
H
−1
B
+1
I
0
A
+1
E
0
J
0
C
No rebalancing











No rebalancing required. So , this is the final AVL search tree.

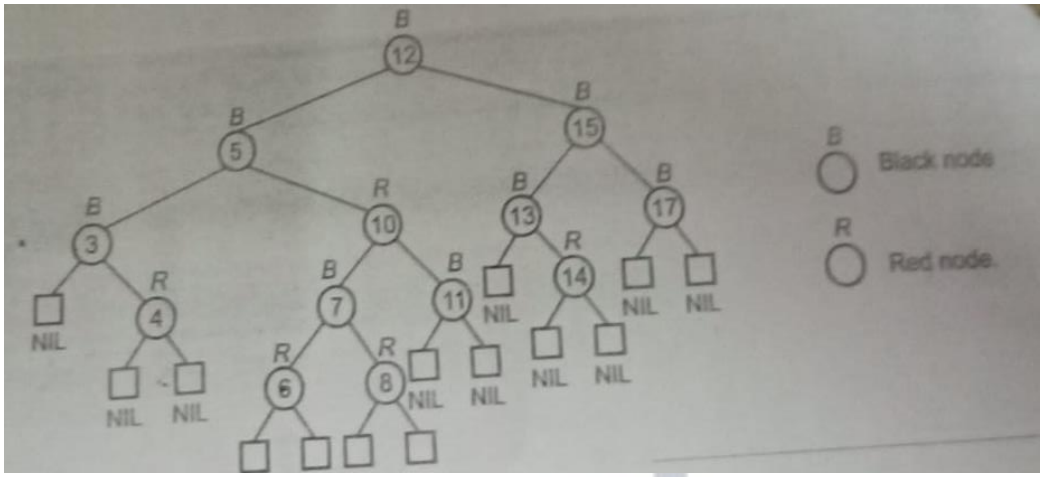
Efficiency of AVL tree: the overhead of calculating balance factors, determining imbalance and correcting it may be considered even though these algorithms are still 0(lg n). to do all of the rebalancing when one node goes out of balance since all changes are restricted to the search path of the node being inserted. Height balancing for a single node is still 0(1). However, it would be expected that on the average , only about half the insertions result in an unbalanced tree.

Red-Black trees: It is a type of self balancing binary search tree. It was invented in 1972 by Rudolf Bayer who called them “symmetric binary B-trees”.

Properties of red-black trees are:

1. The root is always black.
2. A nil is considered to be black. This means that every non- NIL node has two vhildren.
3. Black children rule; the children of each red node are black.
4. Black height rule: For each node v, there exists an integer bh (v) such that each downward path from v to a nil has exactly bh(v) black real nodes. Call this quantity the black height of v. We define the black height of an RB tree to be the black height of its root.

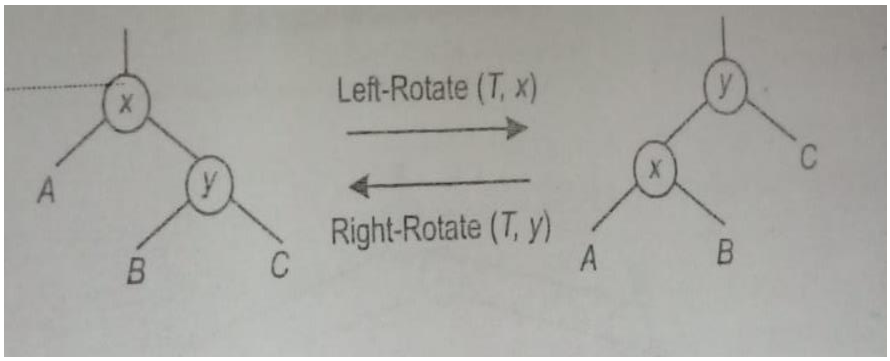
   A tree T is an almost red-black tree if the root is red, but other conditions above hold.



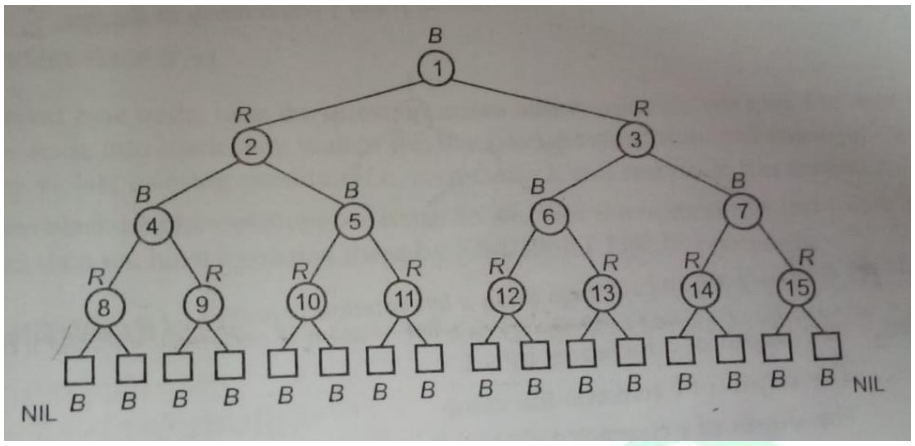
Lemma: Let be an RB tree having some n internal nodes. Then the height of T is at most 2 lg (n+1).

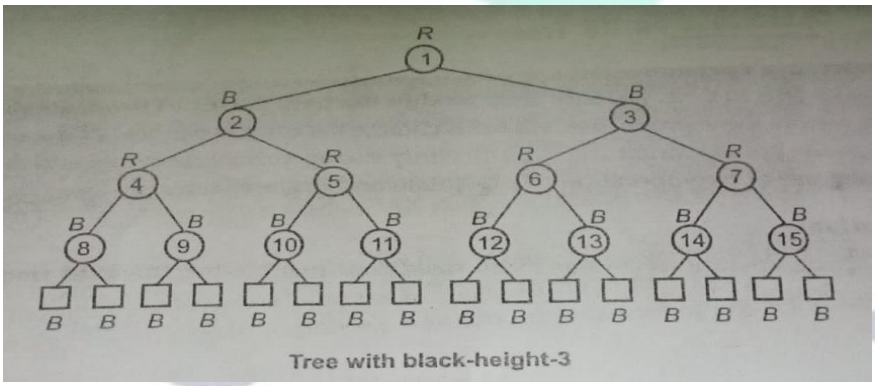
Operations on RB trees: The search tree operations TREE-INSERT and TREE-DELETE, when run on a red-black tree with n keys, take 0(lg n) time. Because they modify the tree, the result may violate the red-black properties. To restore these properties, we must change the colors of some of the nodes in the tree and also change the pointer structure.

Rotations: restructuring operations on red-black trees can often be expressed more clearly in terms of the rotation operation.

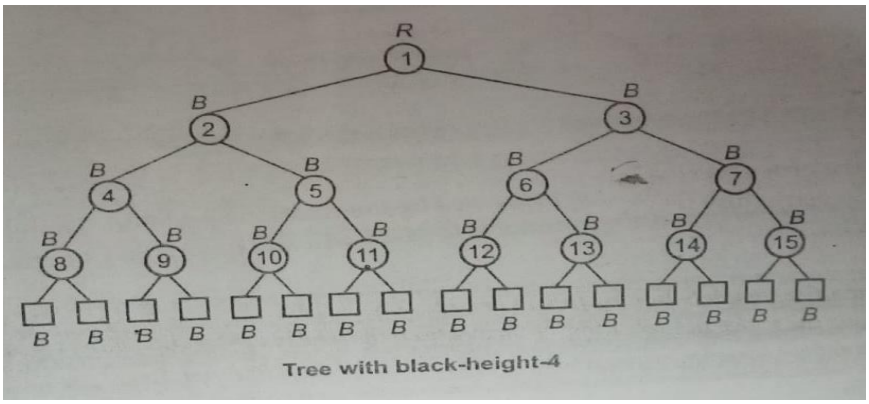


Example: Draw the complete binary tree of height 3 on the keys {1,2,3,…,15}. Add the NIL leaves and colour the nodes in three different ways such that the black- heights of the resulting trees are: 2,3 and 4





Tree with black-height-3



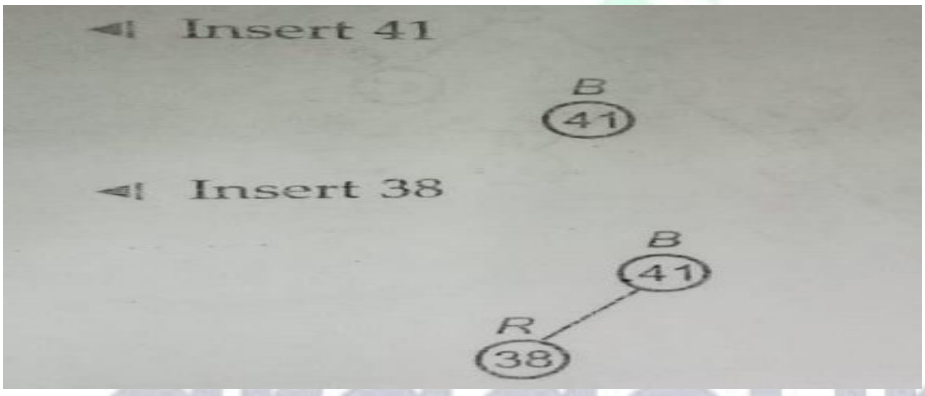
Tree with black-height-4

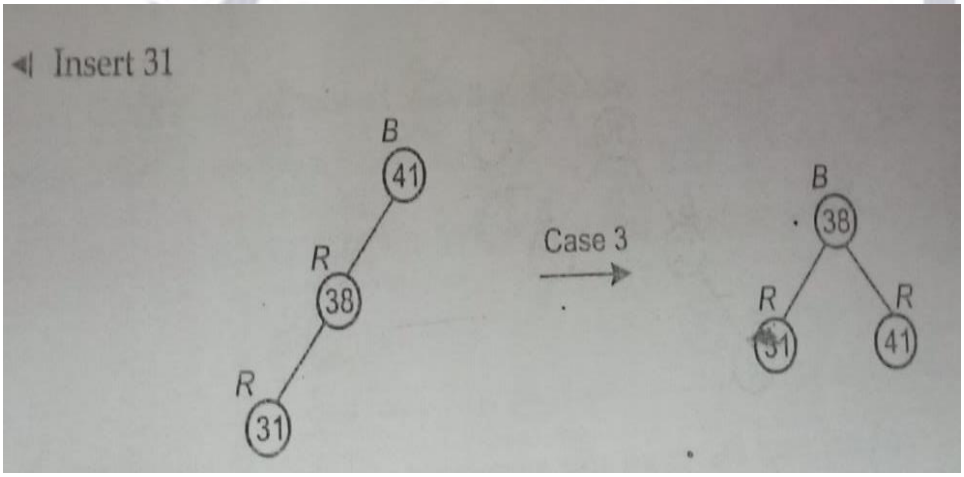
Insertion:

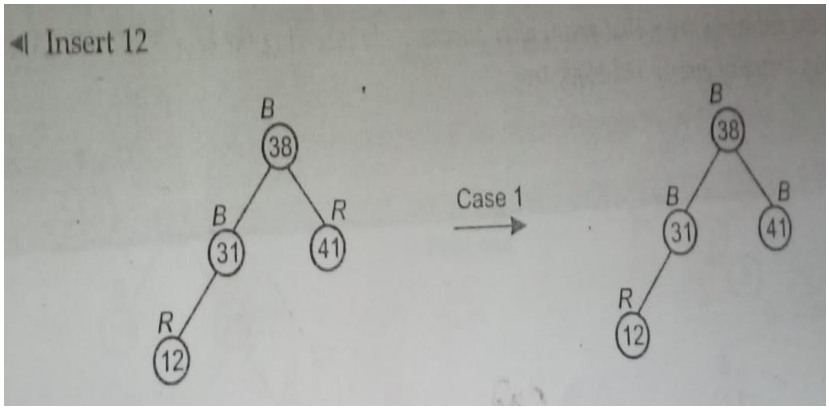
- Insert the new code the way it is done in binary search trees
- Color the node red
- If an inconsistency arises for the red-black tree, fix the tree according to the type of discrepancy.

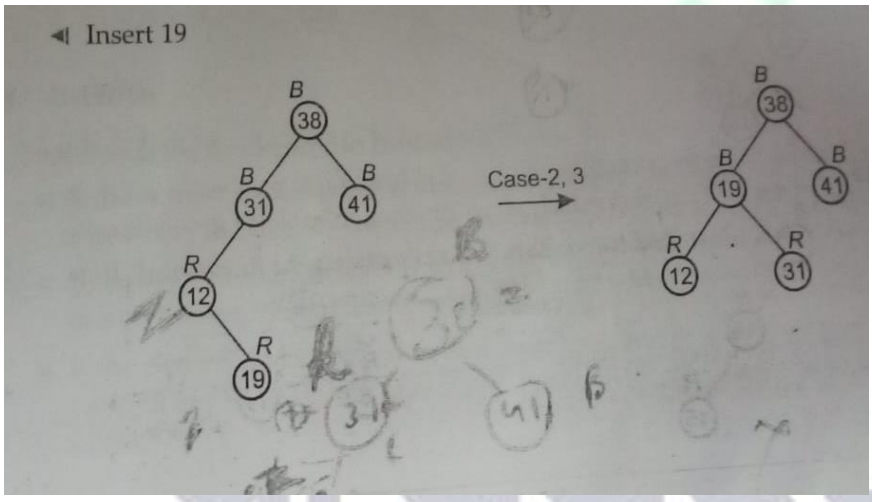
A discrepancy can result from a parent and a child both having a red color. This type is determined by the location of the node with respect to its grand parent, and the color of the sibling of the parent.

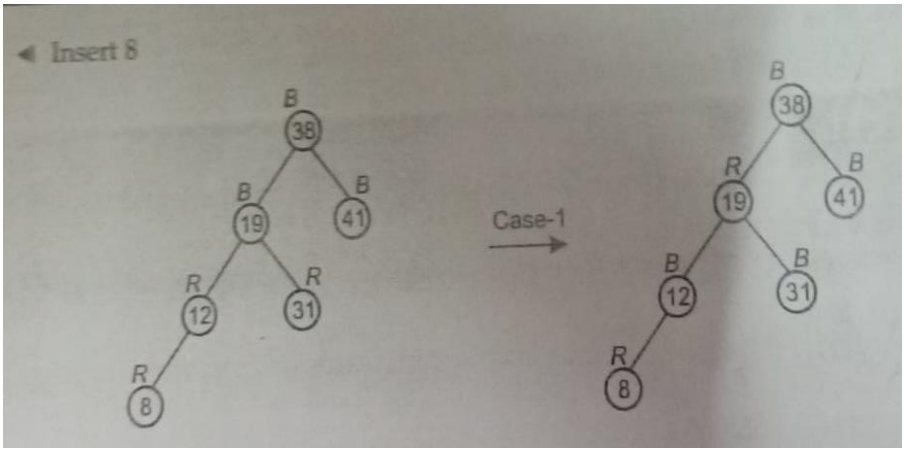
Example: show the red-black trees that result after successively inserting the keys 41, 38, 31, 12, 19, 8 into an initially empty red-black tree.



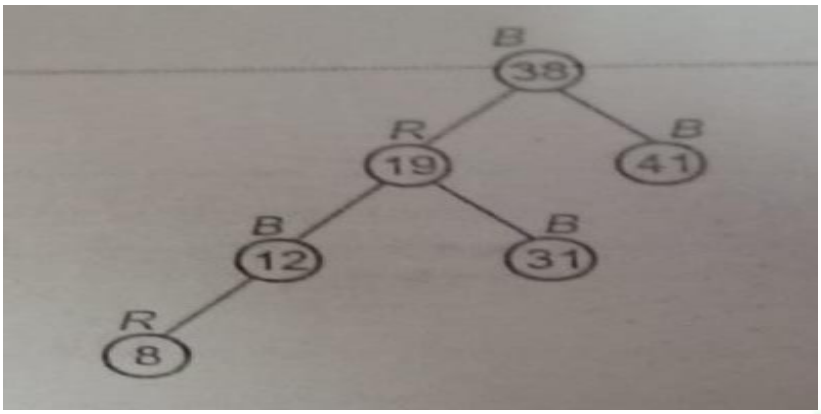



Insert 12
B
38
B
31
R
41
R
12
Case 1
B
38
B
31
B
41
R
12

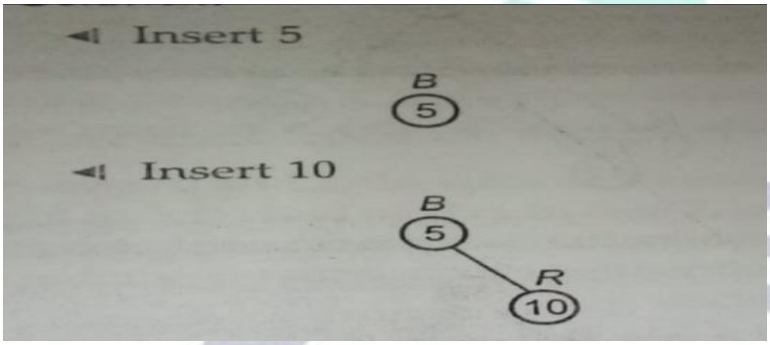

Insert 19
B
38
B
31
B
41
R
12
R
19
Case-2, 3
B
38
B
19
B
41
R
12
R
31

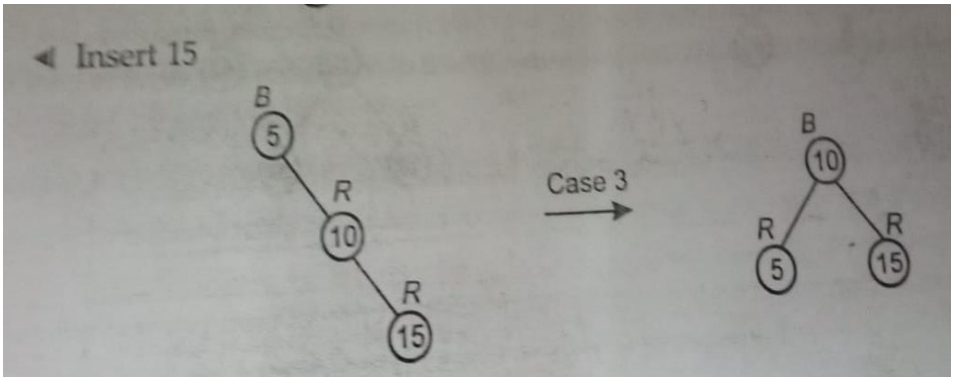

Insert 8
B
38
B
19
B
41
R
12
R
31
R
8
Case-1
B
38
R
19
B
41
B
12
B
31
R
8

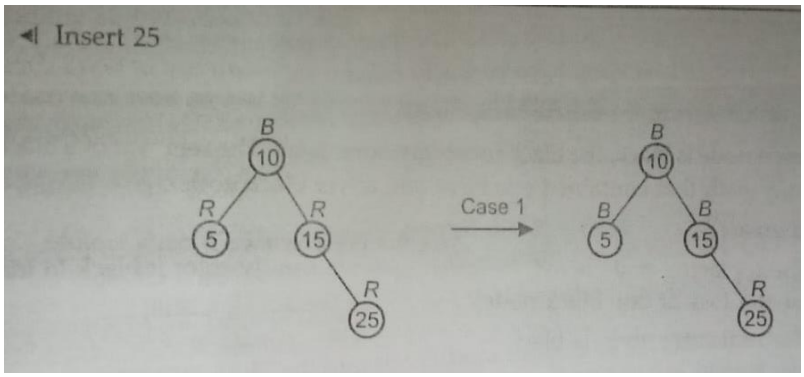
Thus final tree is

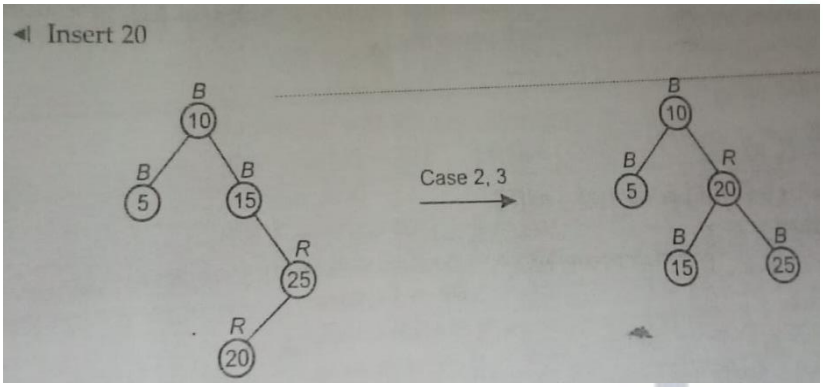


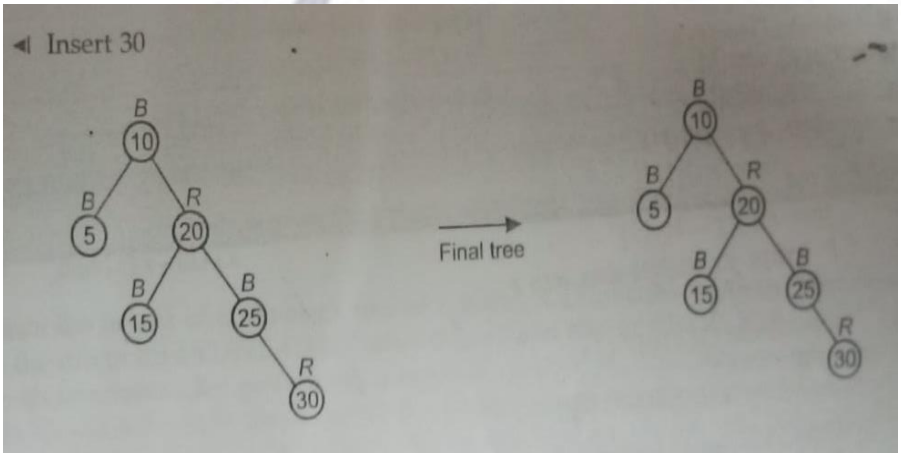
Example: show the red-black trees that result after successively inserting the keys 5, 10, 15, 25, 20 and 30 into an initially empty red-black tree.






Insert 25
B
10
R
5
R
15
R
25
Case 1
B
10
B
5
B
15
R
25


Insert 20
B
10
B
5
B
15
R
25
R
20
Case 2, 3
B
10
B
5
R
20
B
15
B
25


Insert 30
B
10
B
5
R
20
B
15
B
25
R
30
Final tree
B
10
B
5
R
20
B
15
B
25
R
30

Deletion:

- If the element to be deleted is in a node with only left child, swap this node with the one containing the largest element in the left sub tree.
- If the element to be deleted is in a node with only right child, swap this node with the one containing the smallest element in the right sub tree.
- If the element to be deleted is in a node with both right child and a left child, then swap in any of the above two ways. While swapping, swap only the keys but not the colors.
- The item to be deleted is now in a node having only a left child or only a right child . Replace this node with its sole child. This may violate red constraint or black constraint violation of red constraint can be easily fixed.
- If the deleted node is black, the black constraint is violated. The removal of a black node y causes any path that contained y to have one fewer black node.

Two cases arise:

1. The replacing node is red, in which case we merely color it black to make up for the loss of one black node.
2. The replacing node is black.

The procedure RB-DELETE is a minor modification of the TREE-DELETE procedure. After splicing out a node, it calls an auxiliary procedure RB-DELETE-FIXUP that changes colors and performs rotations to restore the red-black properties.

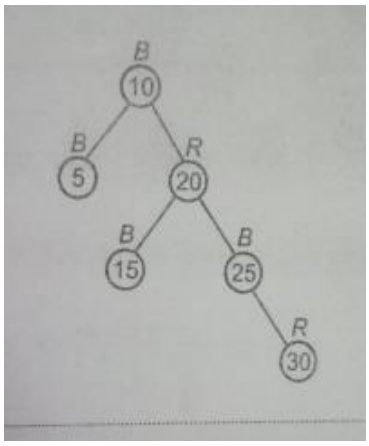
Elementary properties of Red-Black Tree.

1. Black-height of an red-black tree is the black height of its root. This equals to the number of black edges to reach a leaf.
2. If red-black tree has black-height:

a) Then it has atleast 2^{bh} -1 internal black nodes.

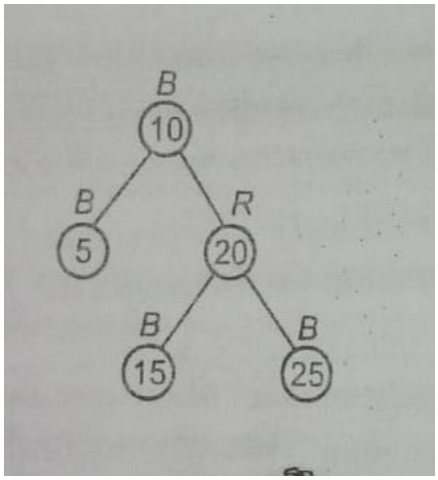
b) It has atleast 4^{bh} -1 internal nodes.

Example: show the red-black tree that results from successively deleting the keys 30, 25, 20, 15, 10 and 5 from the final tree.

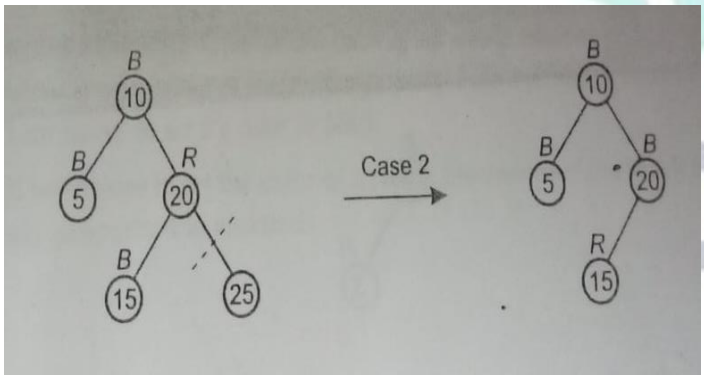


Solution

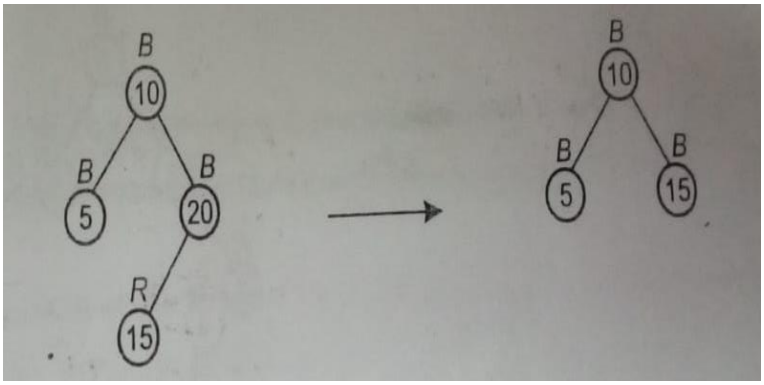
Delete 30



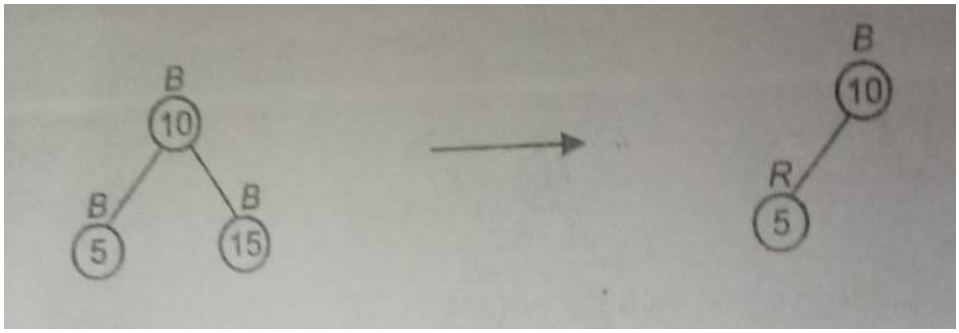
delete 25



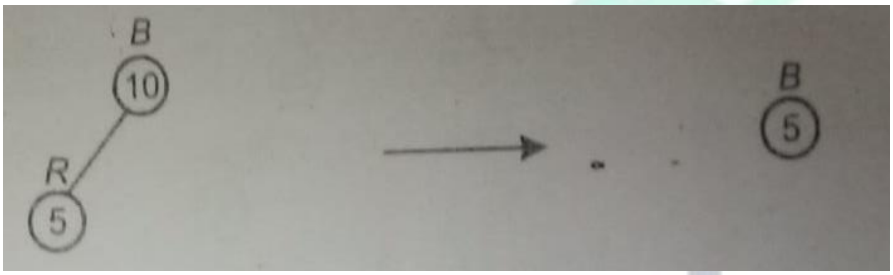
delete 20



delete 15



delete 10



delete 5. No tree.

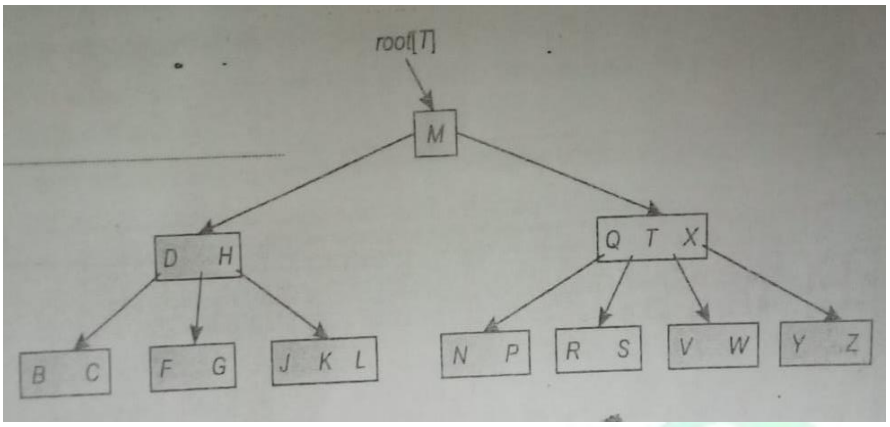
B- Trees: It is a tree data structure that keeps data sorted and allows insertions and deletions in logarithmic amortized time. They are balanced search trees designed to work well on magnetic disks or other direct-access secondary storage devices. It is most commonly used in databases and file systems. Its creators, Rudolf Bayer and Ed McCreight, have not explained what, if anything, the B stands for. The most common belief is that B stands for balanced, as all the leaf nodes are at the same level in the tree.

It is kept balanced by requiring that all leaf nodes are at the same depth. This depth will increase slowly as elements are added to the tree, but an increase in the overall depth is infrequent, and results in all leaf nodes being one more hop further removed from the root.

Difference from Red Black tree

B-trees differ from red-black trees in that B-trees nodes may have many children, from a handful to thousands. That is, “branching factor” of a B-tree can be quite large, although it is usually determined by characteristics of the disk unit used. B-trees are similar to red- black trees in that every n-node B-tree

has height 0(lg n), although the height of a B-tree can be considered less than that of a red-black tree because its branching factor can be much larger. B-trees are similar to red-black trees but they are better at minimizing disk I/O operations.



The number of children a B-tree can have limited by the size of a disk page.

Key points for B-tree are as follows:

1. They are balanced search designed to work will on magnetic disks or other direct access secondary storage devices.
2. Many data base systems use B-trees or variants of B-trees such as B^+ or B^* to store information.
3. They are similar to RB trees but they are better at minimizing disk I/O operations.

A B-tree is rooted tree having the following properties:

1. All leaves have the same depth, which is the tree's height h.
2. There are lower and upper bounds on the number of keys a node can contain. These bounds are expressed in terms of a fixed interger t>2 called the minimum degree of the B-tree:

a) Every node other than the root must have at least t-1 keys. Every internal node other the root has at least t children. If the tree is non empty, the root must have at least one key.

b) Every node can contain at most 2t-1 keys. Therefore, an internal node can have at most 2 t children. We can say that a node is full if it contains exactly 2 t -1 keys.

Theorem

If n>1, then for any n-keys B-tree T of height h and minimum degree t>2, then

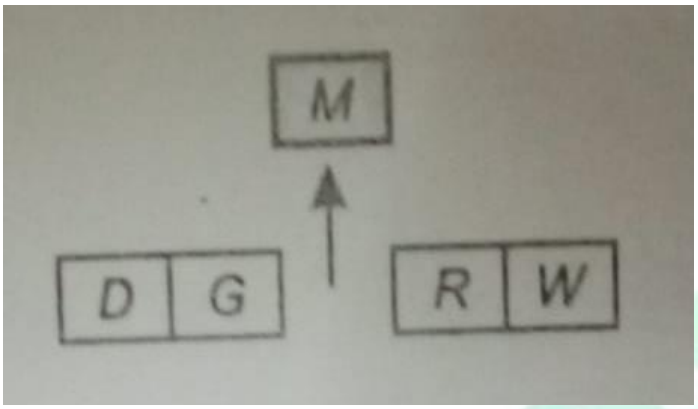
$h<\log_t , n+1/2$

In a B-tree-search procedure, the nodes encountered during the recursion from a path down-ward from the root of the tree. The number of disk pages accessed by B-TREE-SEARCH is therefore $0(\log_t n)$ where h is the height of the tree and n is the number of keys in the tree.

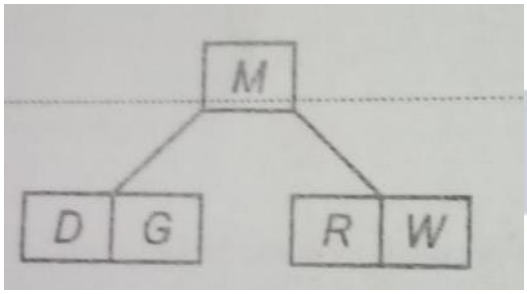
Example: After inserting D, G, M, R and W into a B-tree with minimum degree 3, 2, to 5 values per node:

D	G	M	R	W

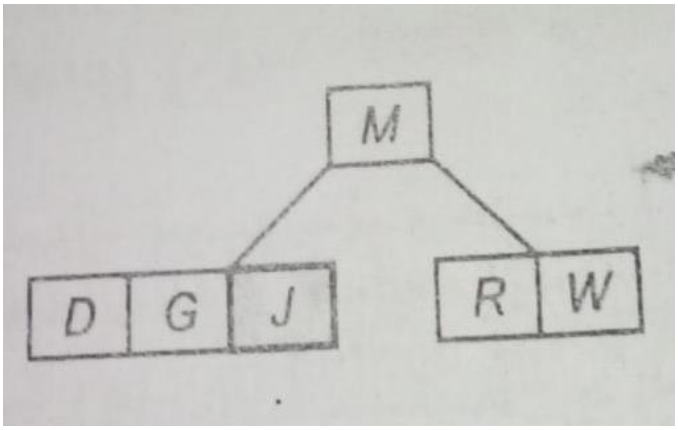
To insert 'J' : node is full thus before insert node j.



The root node must be split. Move the middle value up and create two children

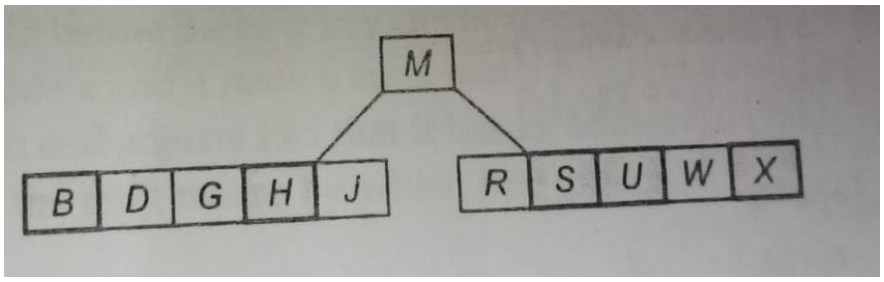


Set the child pointers

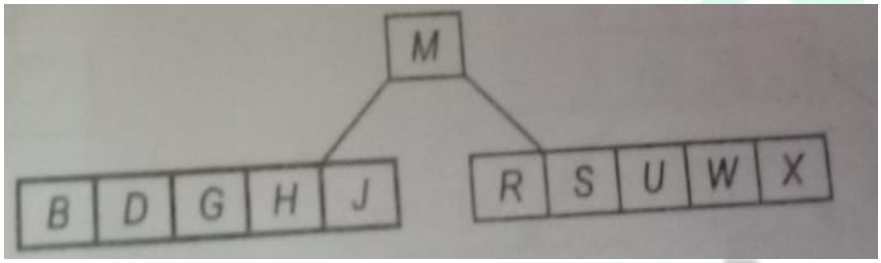


Place j in the appropriate node.

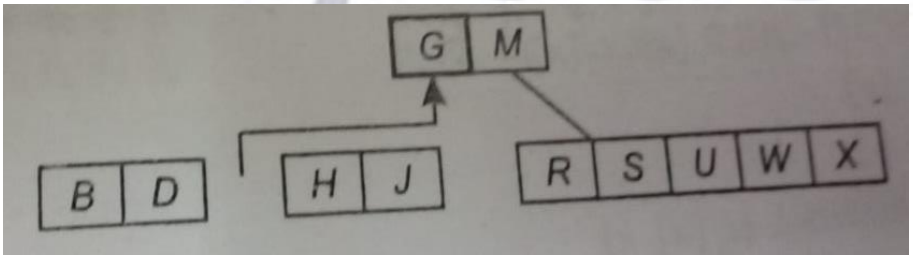
After inserting B, H, S, U, and X:



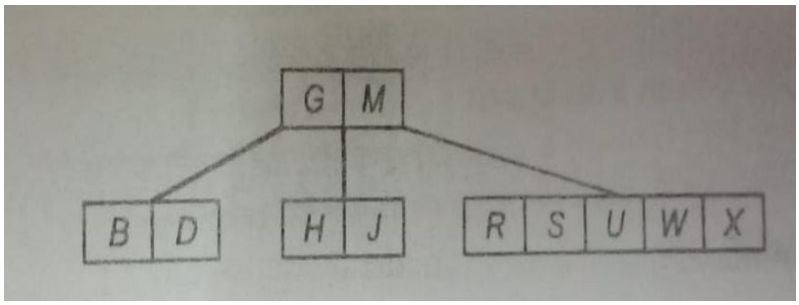
To insert A



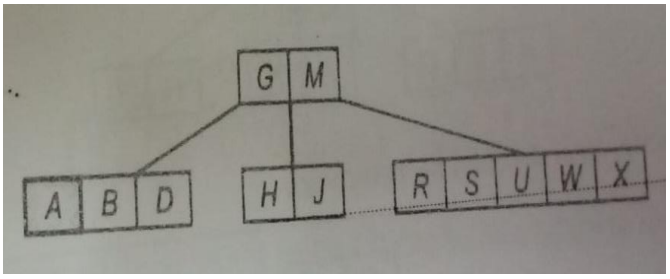
The node where A must go, is full, so it must be split before inserting node A.



Move the middle value G, up ibto its parent and create 2 children

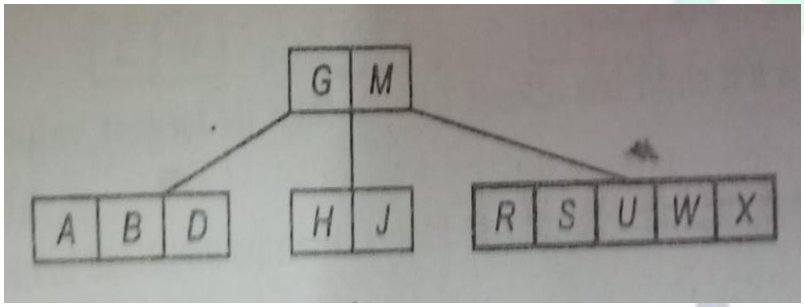


See the children pointers.

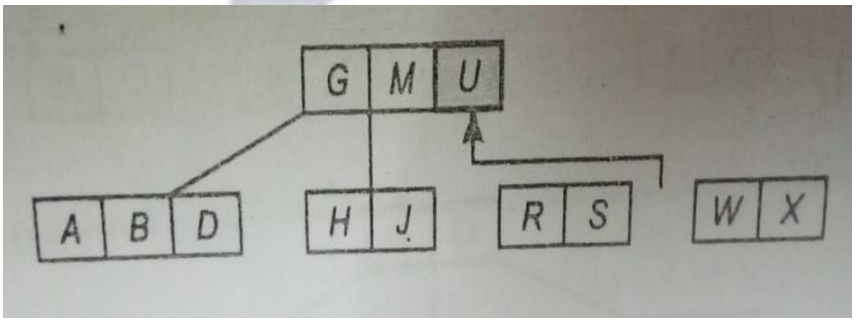


Place A in the appropriate node.

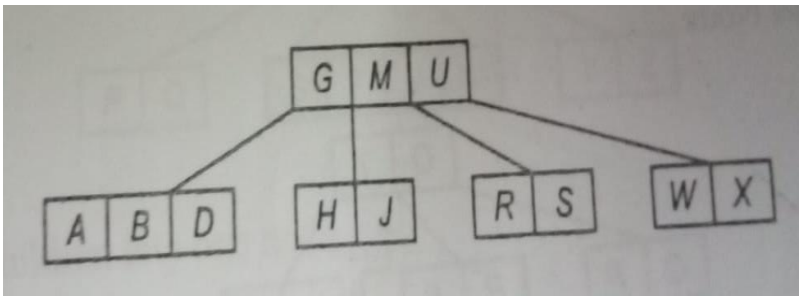
To insert T



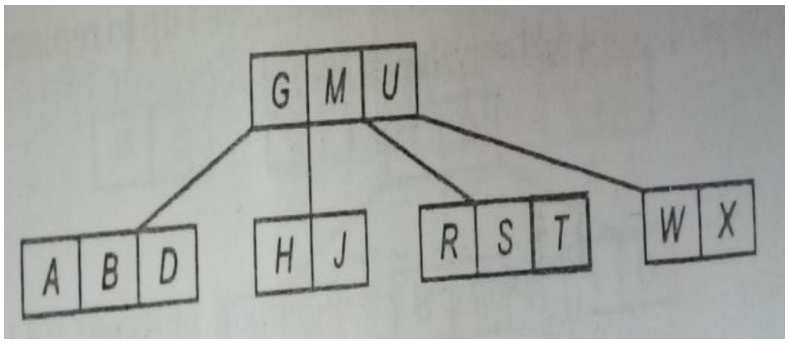
The node where T goes is full, so it must be split.



Move the middle value U up to its parent and create two children



See the child pointers.

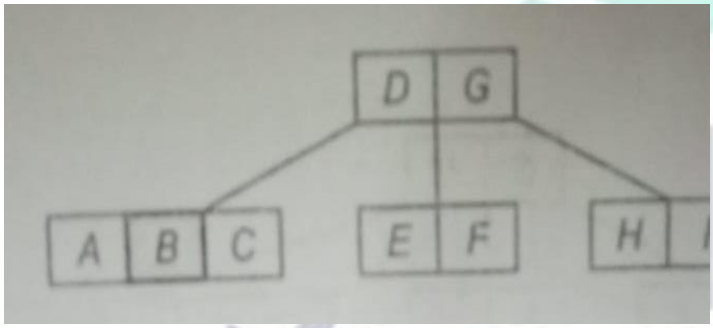


Delete a key from a B-tree

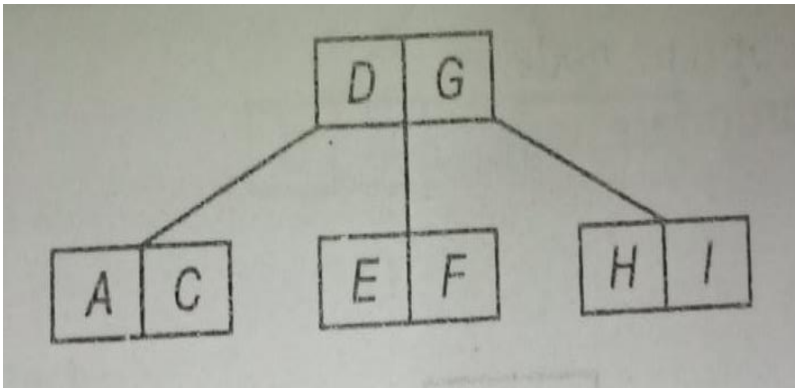
It has several different cases:

Case1: if x is a leaf node, then the key can just be removed.

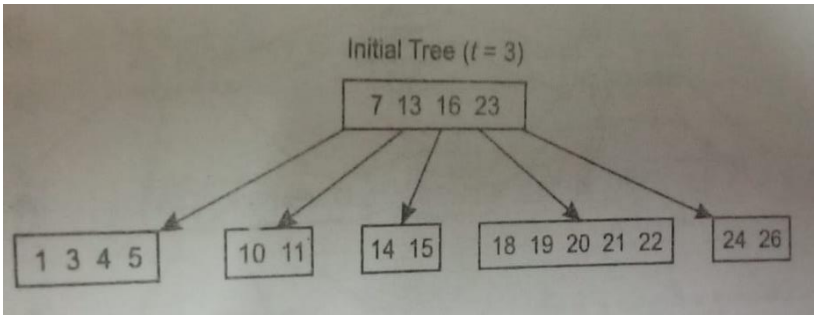
Delete B



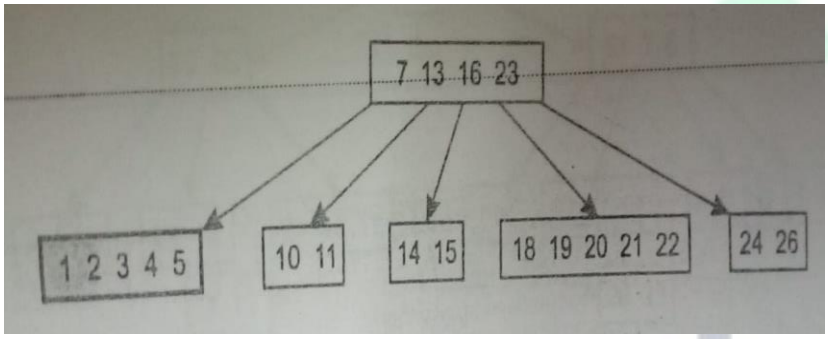
B is in a leaf node, so it can just be removed.



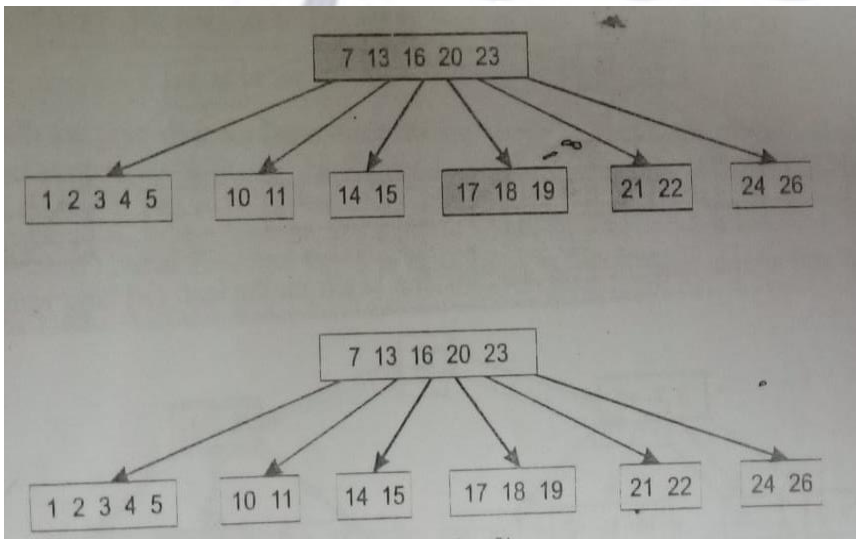
Inserting a key in a B-tree



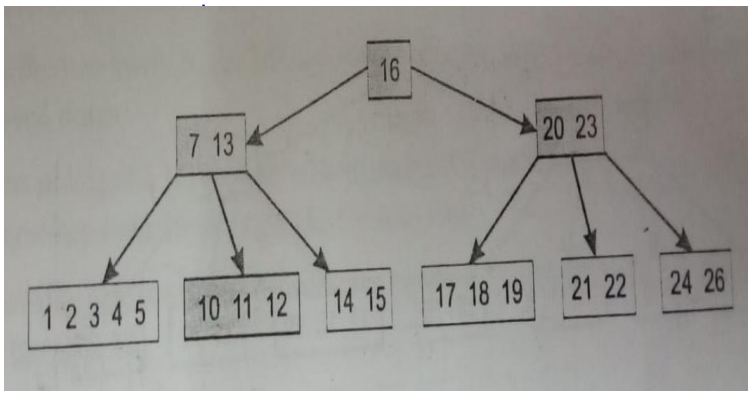
Insert 2



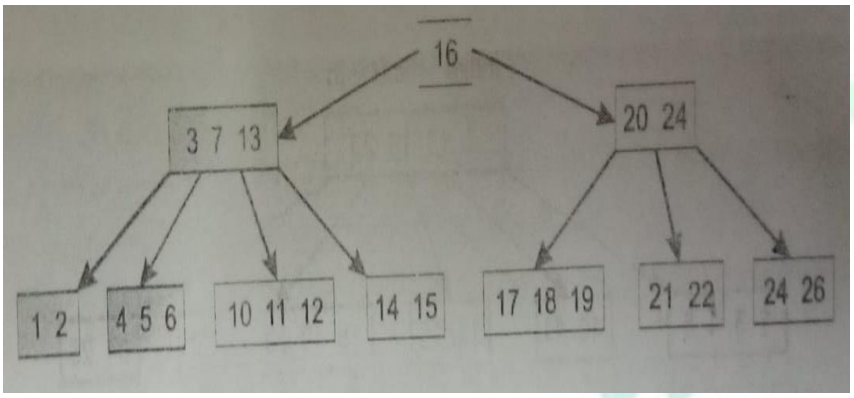
Insert 17



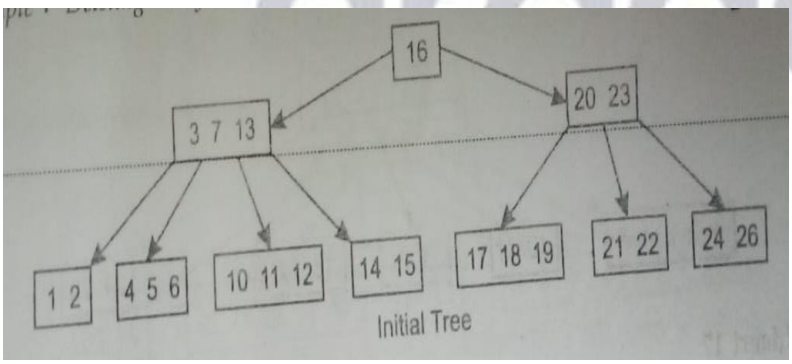
Insert 12



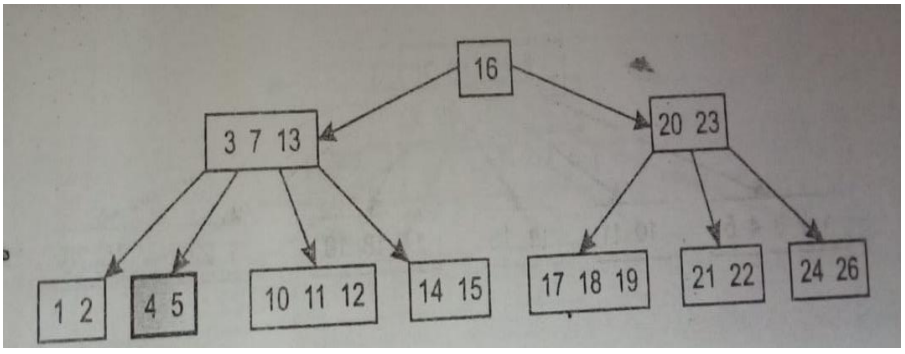
Insert 6



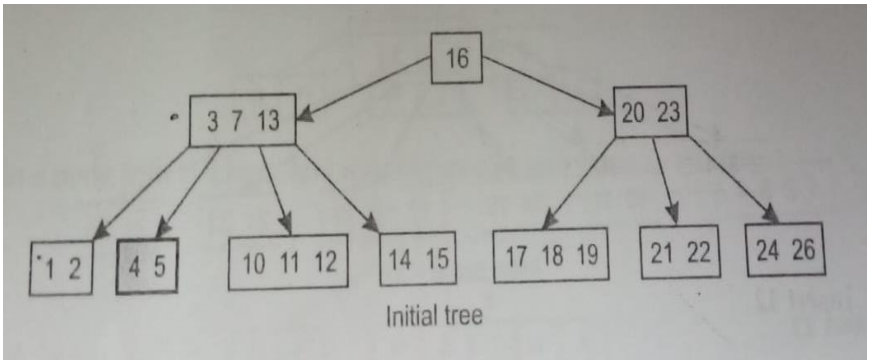
Deleting a key in a B-tree



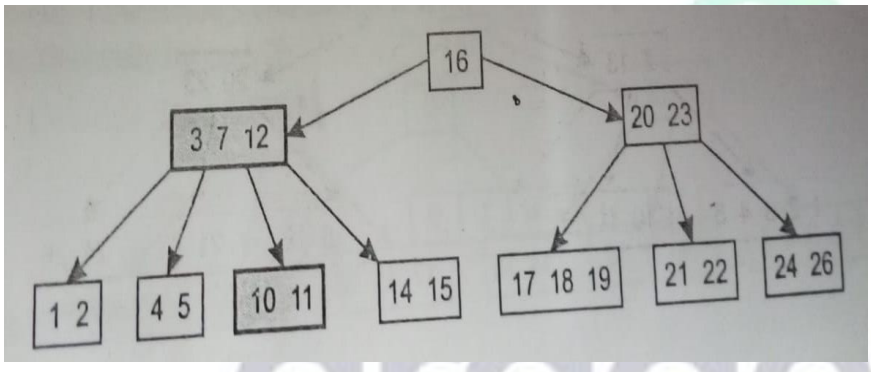
6 deleted



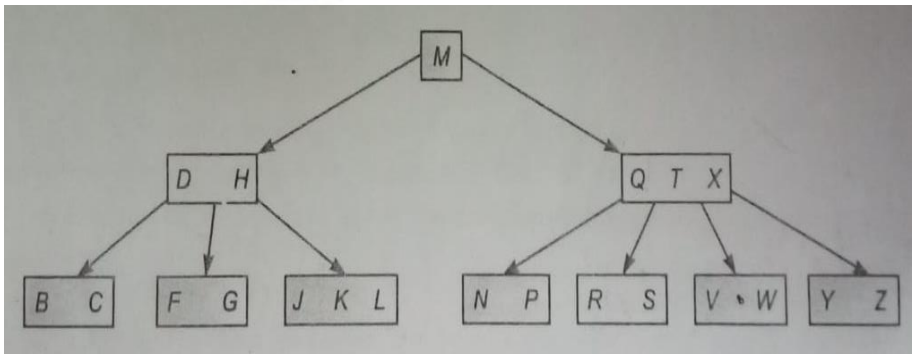
The first and simple case involves deleting the key from the leaf. T-1 keys remain.



13 deleted



Example: for what values of t is the tree given below is a legal B-tree



For a legal B-tree root node contains at least one key and all other nodes contain t-1 keys.

Since in the given B-tree node 2 contains 2 keys D,H i.e.,

(t-1)keys= D, H=2 keys

This is possible only when t=3.

B*-Trees

B*-tree of order m is a search tree that is either empty or that satisfies three properties:

The root node has minimum two and maximum 2 floor ((2m-2)/3) +1 children

Other internal nodes have the minimum floor ((2m-1)/3) and maximum m children

All external nodes are on the same level.

The advantage of using B* trees over B-trees is a unique feature called the 'two-to-three' split. By this, the minimum number of keys in each node is not half the maximum number, but two-thirds of it, making data far more compact. However, the disadvantage of this is a complex deletion operation.

. If inserting into a full leaf node (which is not the root) and which has a full right sibling (and whose parent has at least one free key):

Take an array consisting of 'm-1' keys of the full leaf-node, the parent key of this node, the new key to be inserted, and the 'm-1' keys of its right sibling (Totally m-1 + 1 + 1 + m-1 = 2m keys)

Sort these keys

Create three new nodes:

p – whose keys are the first (2m – 2)/3 elements of 'marray'

The element at index (2m – 2)/3 is stored as 'parent1'

q – whose keys are the next (2m – 1)/3 elements of 'marray' after parent1

The element at index (4m)/3 is stored as 'parent2'

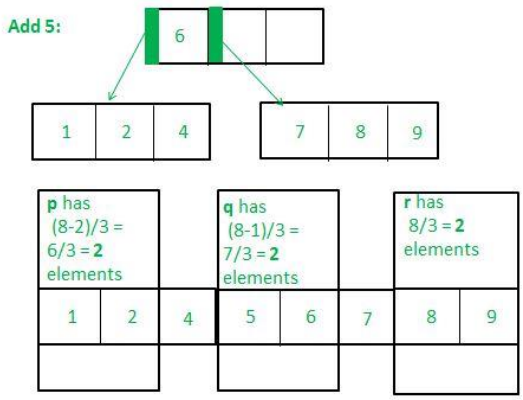
r – whose keys are the last (2m)/3 elements of 'marray' after parent2

The key in the leaf's parent which points to this leaf should have its value replaced as 'parent1'

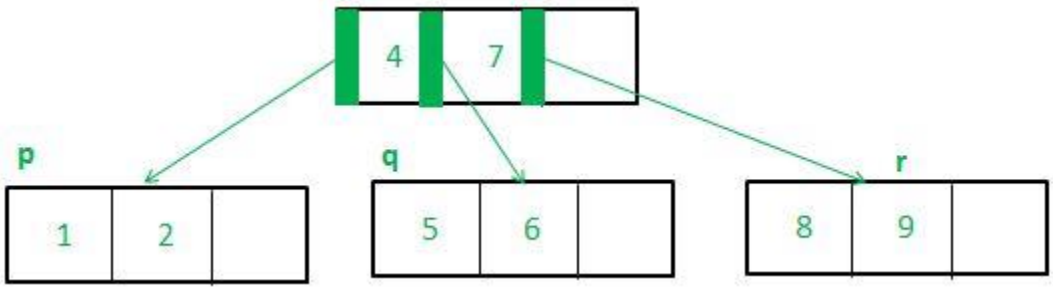
If the parent key in iv) has any adjacent keys, they should be shifted to the right. In the space that remains, place 'parent2'.

p, q and r must be made child keys of parent1 and parent2 (if 'parent1' and 'parent2' are the first two keys in the parent node), else p, q, r must be made the child keys of the key before parent 1, parent 1, and parent 2 respectively.

Before Insertion :

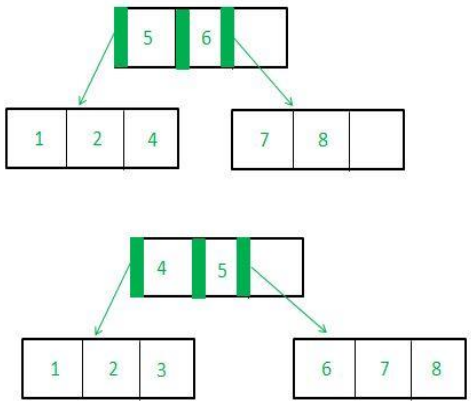


After insertion:



2. If inserting into a full leaf node (which is not the root) with empty/non-full right sibling.

Simply shift the last element of the current node to the position of the parent, shift all the keys in the right sibling to the right, and insert the previous parent. Now, use the gap in your own node to rearrange and fit in the new key.



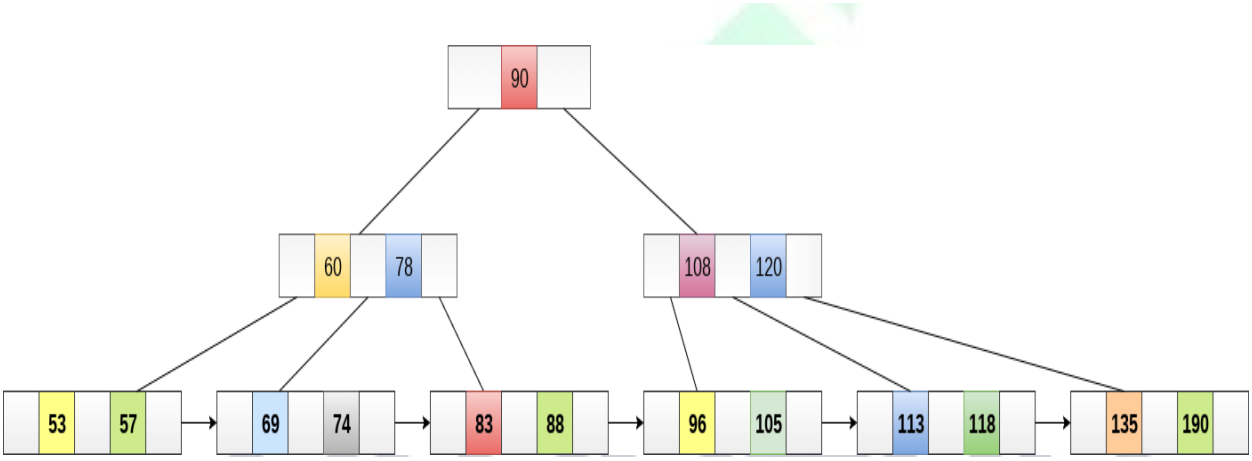
B+ tree

It is an extension of B Tree which allows efficient insertion, deletion and search operations.

In this, Keys and records both can be stored in the internal as well as leaf nodes. Whereas, in B+ tree, records (data) can only be stored on the leaf nodes while internal nodes can only store the key values.

The leaf nodes of these are linked together in the form of a singly linked lists to make the search queries more efficient.

They are used to store the large amount of data which can not be stored in the main memory. Due to the fact that, size of main memory is always limited, the internal nodes (keys to access records) of the B+ tree are stored in the main memory whereas, leaf nodes are stored in the secondary memory.



Advantages of B+ Tree

- Records can be fetched in equal number of disk accesses.
- Height of the tree remains balanced and less as compare to B tree.
- We can access the data stored in a B+ tree sequentially as well as directly.
- Keys are used for indexing.
- Faster search queries as the data is stored only on the leaf nodes.

insertion in B+ Tree

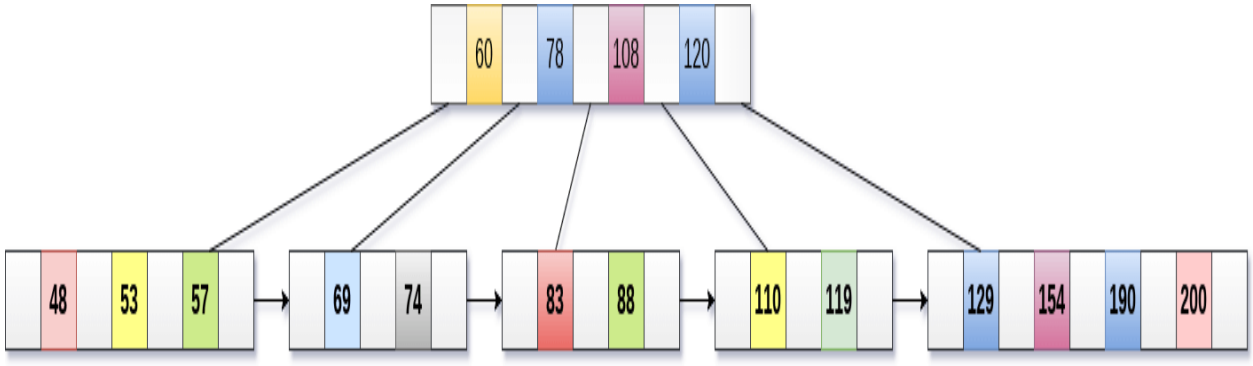
Step 1: Insert the new node as a leaf node

Step 2: If the leaf doesn't have required space, split the node and copy the middle node to the next index node.

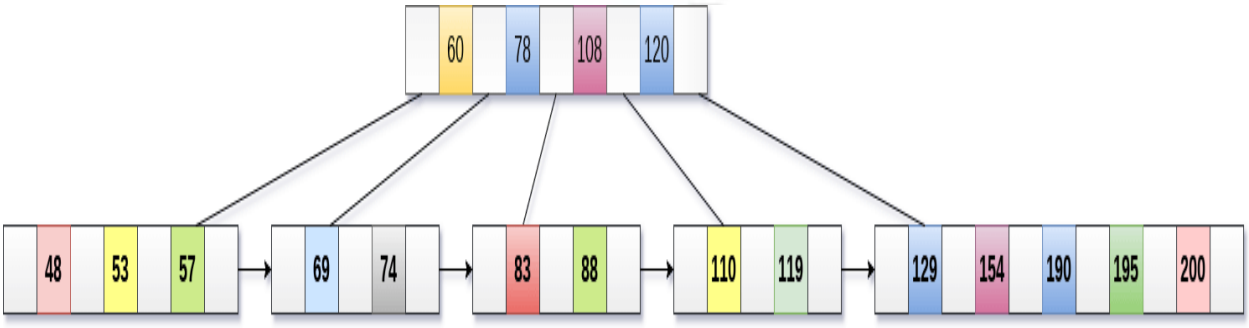
Step 3: If the index node doesn't have required space, split the node and copy the middle element to the next index page.

Example :

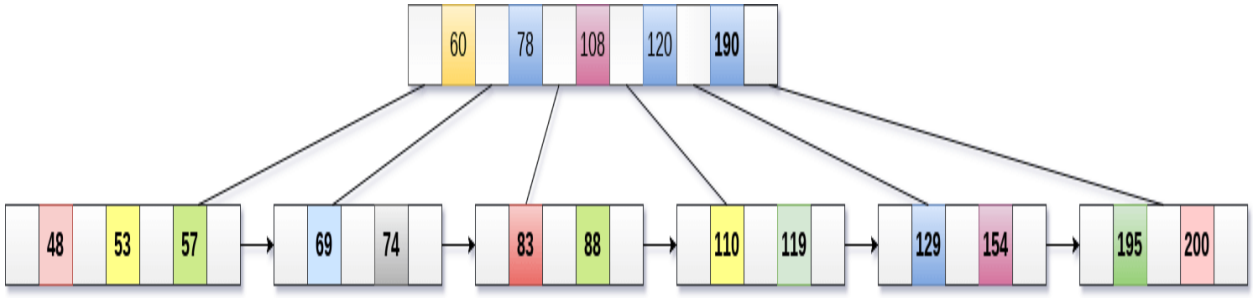
Insert the value 195 into the B+ tree of order 5



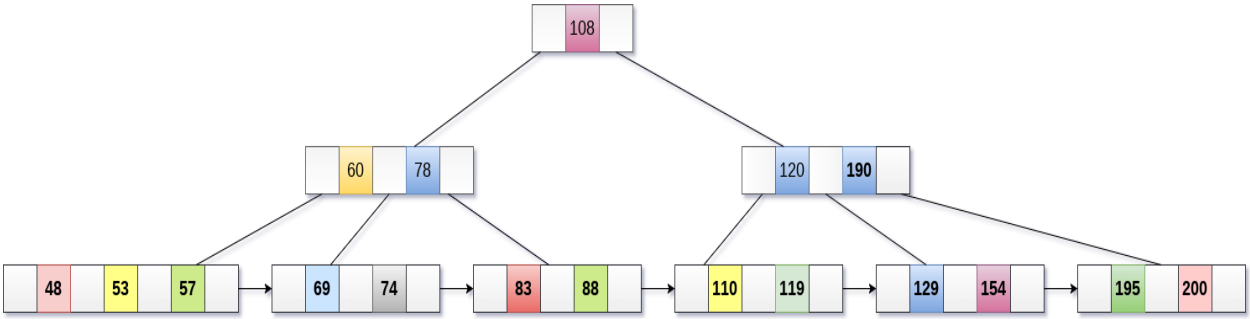
195 will be inserted in the right sub-tree of 120 after 190. Insert it at the desired position.



The node contains greater than the maximum number of elements i.e. 4, therefore split it and place the median node up to the parent.



Now, the index node contains 6 children and 5 keys which violates the B+ tree properties, therefore we need to split it, shown as follows.



Deletion in B+ Tree

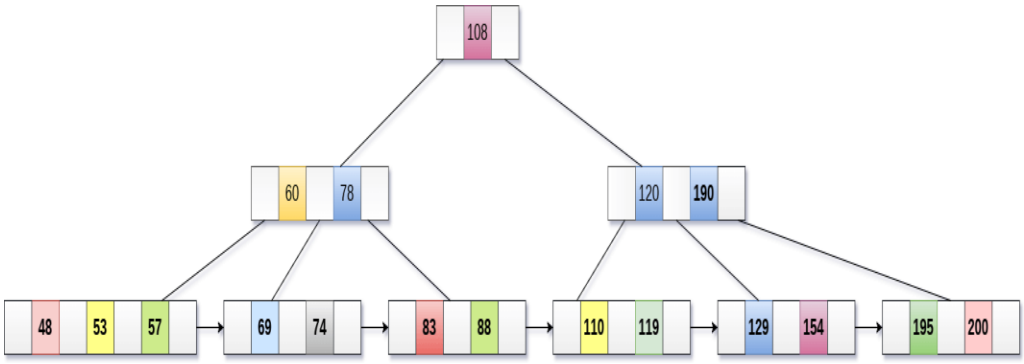
Step 1: Delete the key and data from the leaves.

Step 2: if the leaf node contains less than minimum number of elements, merge down the node with its sibling and delete the key in between them.

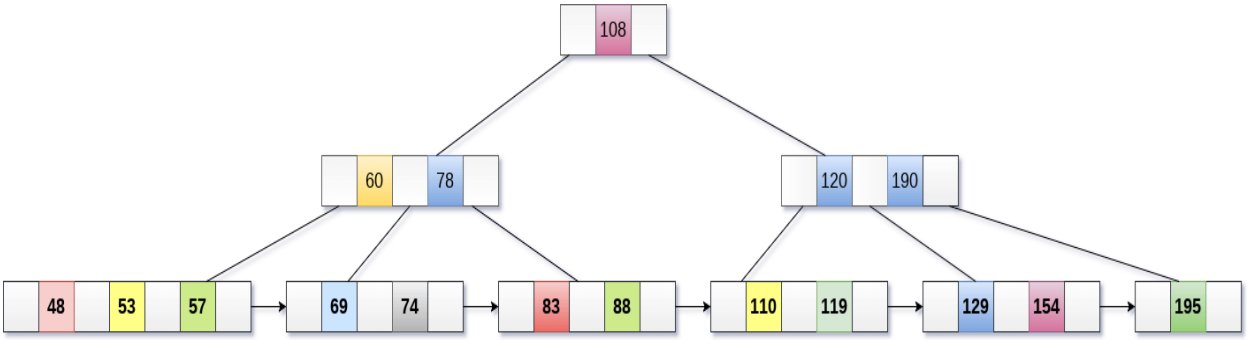
Step 3: if the index node contains less than minimum number of elements, merge the node with the sibling and move down the key in between them.

Example :

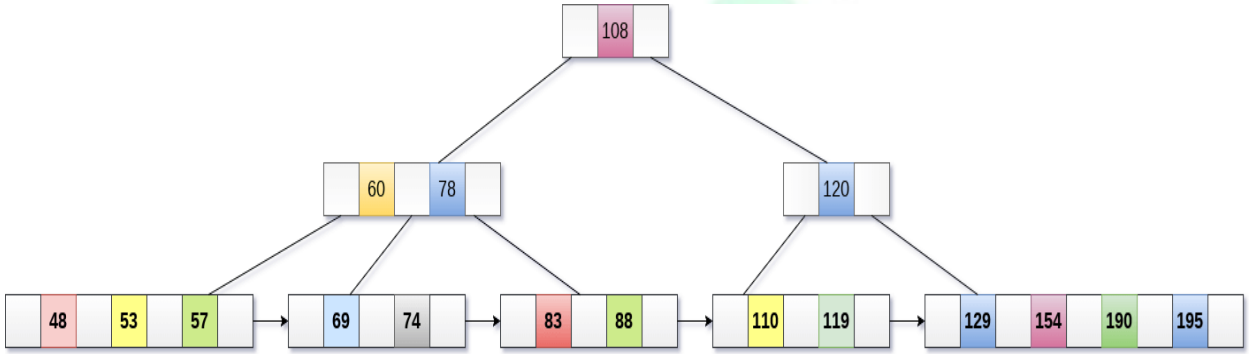
Delete the key 200 from the B+ Tree



200 is present in the right sub-tree of 190, after 195. delete it.

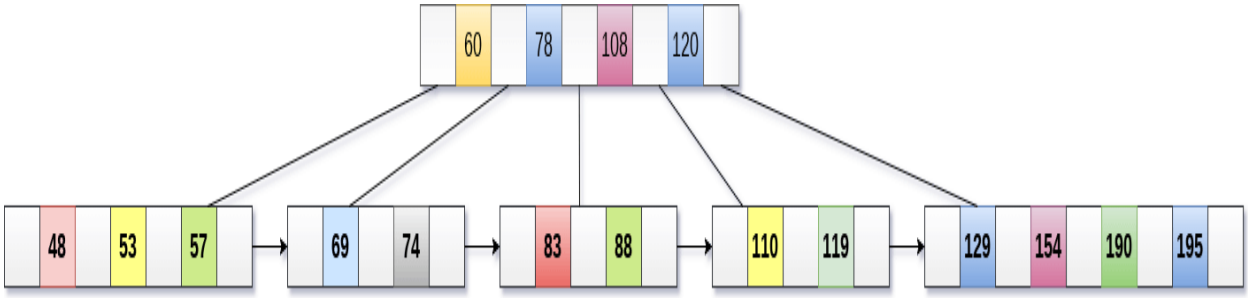


Merge the two nodes by using 195, 190, 154 and 129.



Now, element 120 is the single element present in the node which is violating the B+ Tree properties. Therefore, we need to merge it by using 60, 78, 108 and 120.

Now, the height of B+ tree will be decreased by 1.



gradeup SUPER

Gradeup UGC NET
Super Superscription

Features:

1. 7+ Structured Courses for UGC NET Exam
2. 200+ Mock Tests for UGC NET & MHSET Exams
3. Separate Batches in Hindi & English
4. Mock Tests are available in Hindi & English
5. Available on Mobile & Desktop

Gradeup Super Subscription, Enroll Now