UNIT III

ADVANCED DATA STRUCTURES

AVL TREE:

- AVL tree is a height-balanced binary search tree.
- That means, an AVL tree is also a binary search tree but it is a balanced tree.
- A binary tree is said to be balanced if, the difference between the heights of left and right subtrees of every node in the tree is either -1, 0 or +1.
- In other words, a binary tree is said to be balanced if the height of left and right children of every node differ by either -1, 0 or +1.
- In an AVL tree, every node maintains extra information known as **balance factor.**
- AVL Tree is invented by GM Adelson - Velsky and EM Landis in 1962. The tree is named AVL in honour of its inventors.

An AVL tree is defined as follows...

An AVL tree is a balanced binary search tree. In an AVL tree, balance factor of every node is either -1, 0 or +1.

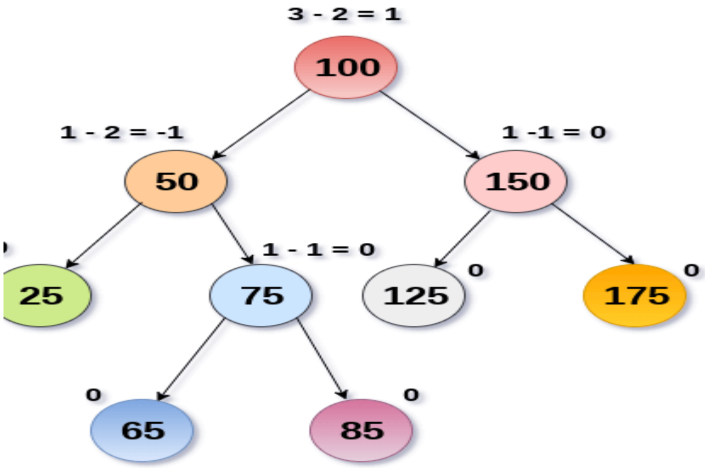
Balance Factor:

- Balance factor of a node is the difference between the heights of the left and right subtrees of that node

Balance factor = height Of Left Subtree – height Of Right Subtree

- **Height Of Left Subtree – Height Of Right Subtree = {-1,0,1}**
- **|BF|=|HLS-HRS|<=1**

Example of AVL Tree:



- The above tree is a binary search tree and every node is satisfying balance factor condition. So this tree is said to be an AVL tree.

NOTE:

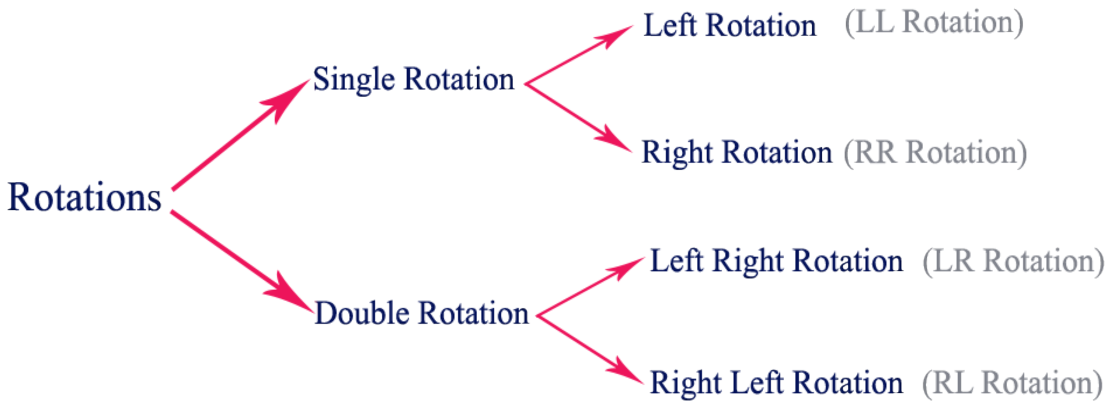
- Every AVL Tree is a binary search tree but every Binary Search Tree need not be AVL tree.

AVL Tree Rotations

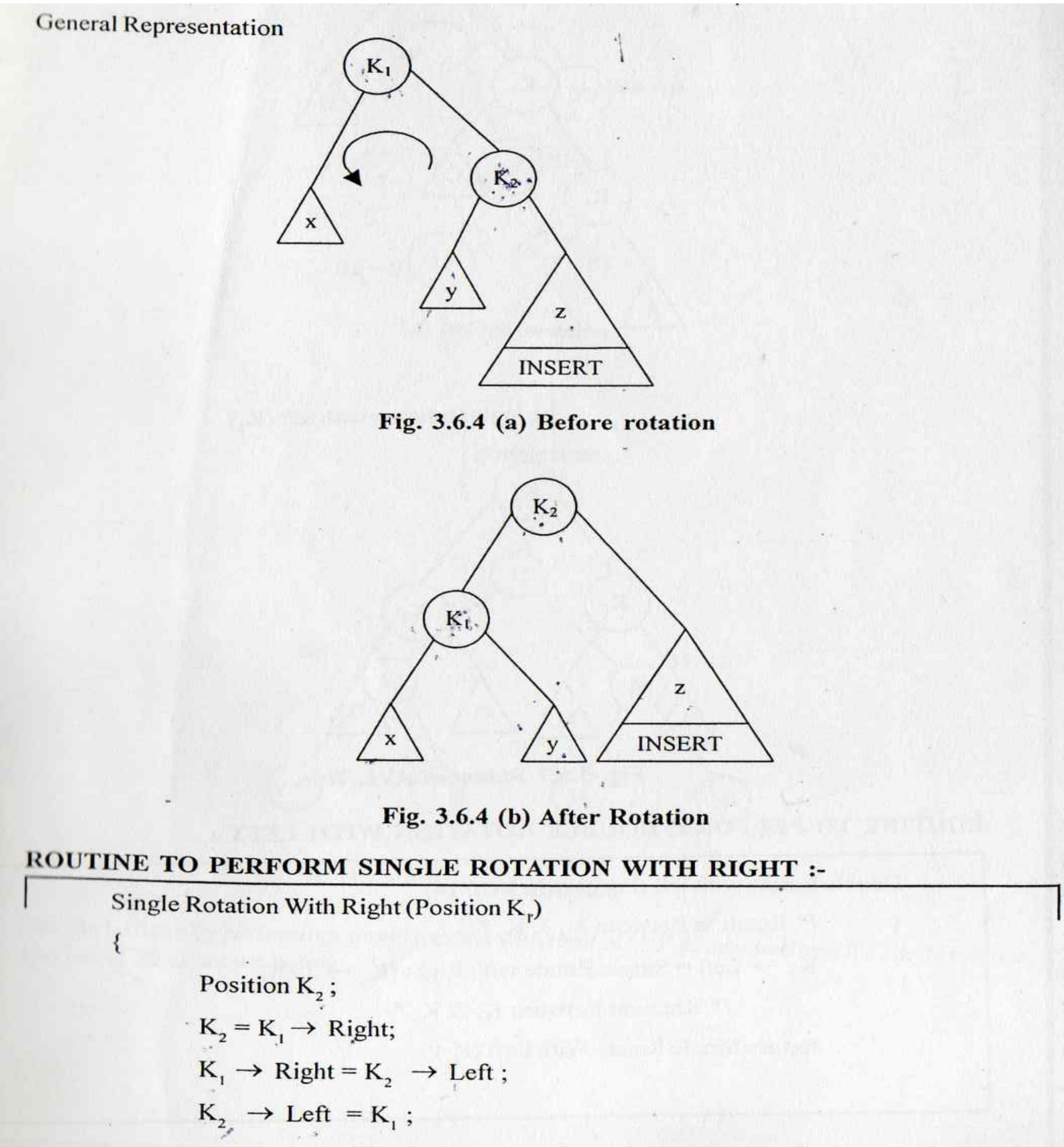
- In AVL tree, after performing operations like insertion and deletion we need to check the **balance factor** of every node in the tree.
- If every node satisfies the balance factor condition then we conclude the operation otherwise we must make it balanced.
- Whenever the tree becomes imbalanced due to any operation we use **rotation** operations to make the tree balanced.
- Rotation operations are used to make the tree balanced.

Rotation is the process of moving nodes either to left or to right to make the tree balanced.

- **There are four rotations and they are classified into two types**.


Rotations
Single Rotation
Left Rotation (LL Rotation)
Right Rotation (RR Rotation)
Double Rotation
Left Right Rotation (LR Rotation)
Right Left Rotation (RL Rotation)

1. Single Left Rotation (LL Rotation)



General Representation

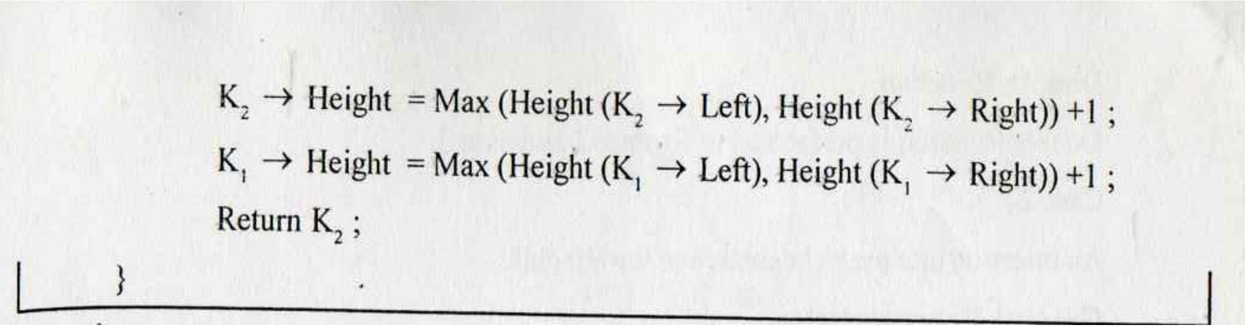
K₁
K₂
x
y
z
INSERT

Fig. 3.6.4 (a) Before rotation

K₂
K₁
x
y
z
INSERT

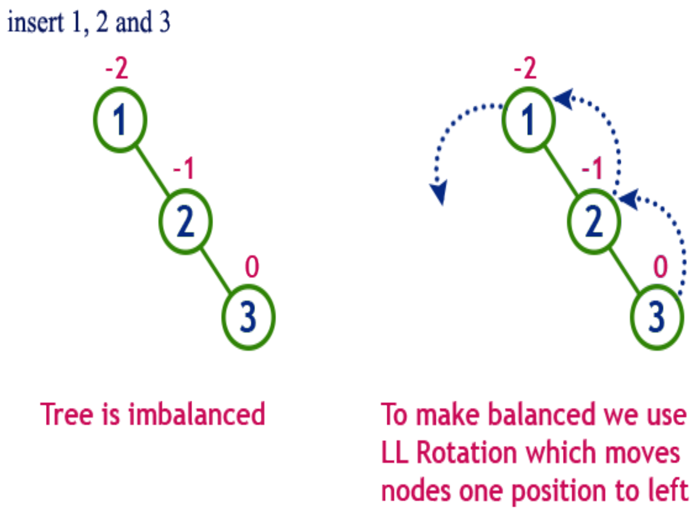
Fig. 3.6.4 (b) After Rotation

ROUTINE TO PERFORM SINGLE ROTATION WITH RIGHT :-



```
Single Rotation With Right (Position K_r)
{
        Position K_2 ;
        K_2 = K_1 → Right;
        K_1 → Right = K_2  → Left ;
        K_2  → Left  = K_1 ;
        K_2 → Height  = Max (Height (K_2 → Left), Height (K_2 → Right)) +1 ;
        K_1 → Height  = Max (Height (K_1 → Left), Height (K_1 → Right)) +1 ;
        Return K_2 ;
}
```

- In **LL Rotation**, every node moves one position to left from the current position.
- To understand LL Rotation, let us consider the following insertion operation in AVL Tree...



2. Single Right Rotation (RR Rotation)

General Representation

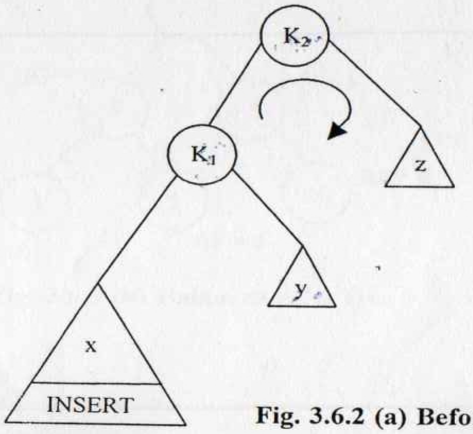


Fig. 3.6.2 (a) Before rotation

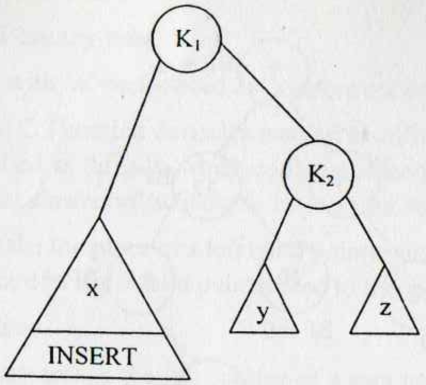


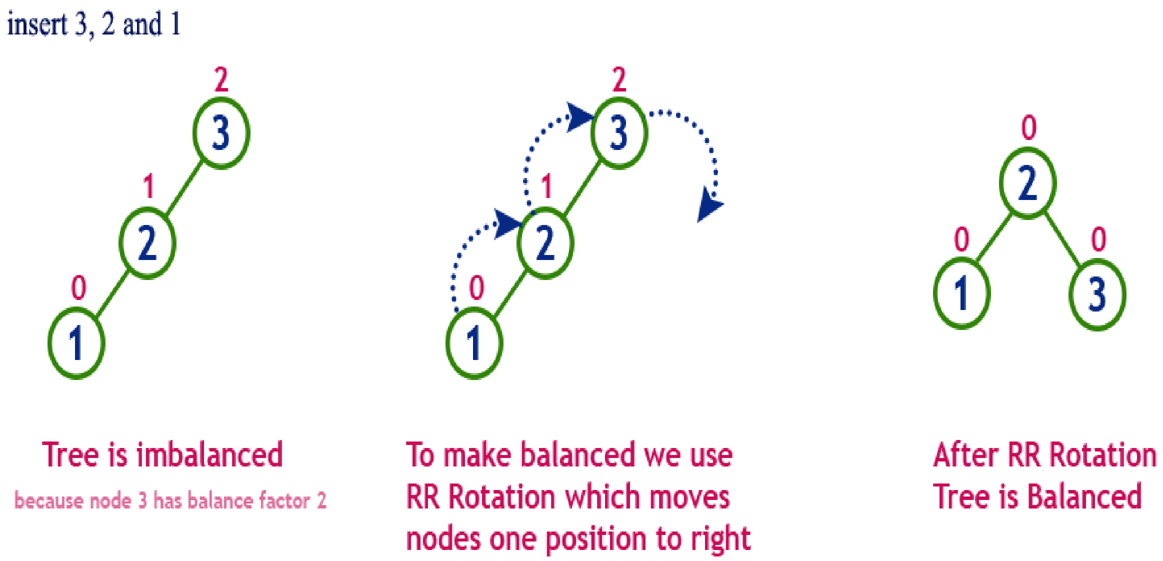
Fig. 3.6.2 (b) After rotation

ROUTINE TO PERFORM SINGLE ROTATION WITH LEFT

```
SingleRotatewithLeft (Position K2)
{
        Position K1;
        K1 = K2 → Left ;
        K2 → left = K1 → Right ;
        K1 → Right = K2 ;
        K2 → Height = Max (Height (K2 → Left), Height (K2 → Right)) + 1 ;
        K1 → Height = Max (Height (K1 → left), Height (K1 → Right)) + 1;
        return K1 ;
}
```

- In **RR Rotation**, every node moves one position to right from the current position.

- To understand RR Rotation, let us consider the following insertion operation in AVL Tree...



3.Left Right Rotation (LR Rotation)

- The LR Rotation is a sequence of single left rotation followed by a single right rotation.
- In LR Rotation, at first, every node moves one position to the left and one position to right from the current position.
- To understand LR Rotation, let us consider the following insertion operation in AVL Tree...

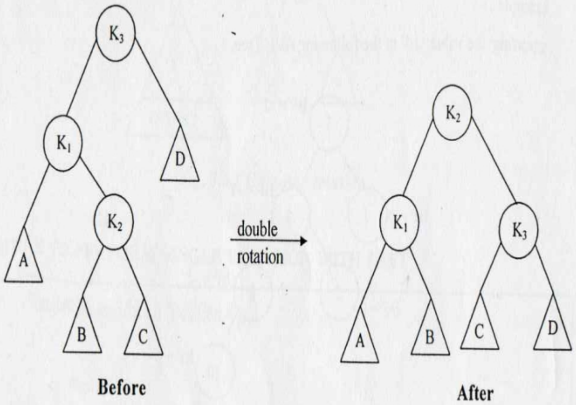
Double Rotation

Double Rotation is performed to fix case 2 and case 3.

Case 2 :

An insertion into the right subtree of the left child.

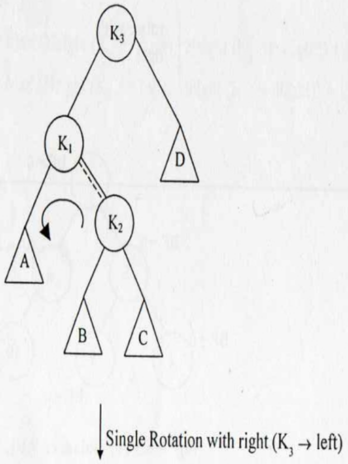
General Representation



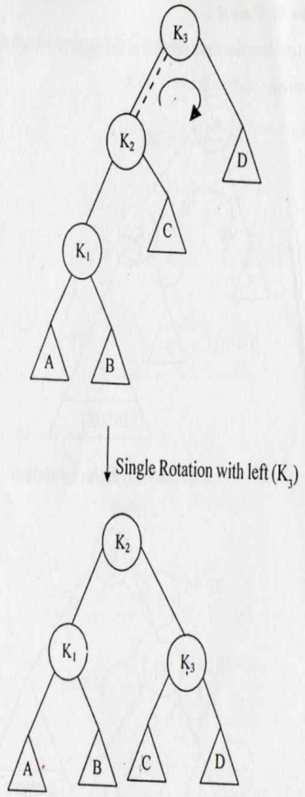
Before **After**

Fig. 3.6.6

This can be performed by 2 single rotations.



Single Rotation with right ($K_3 \rightarrow$ left)

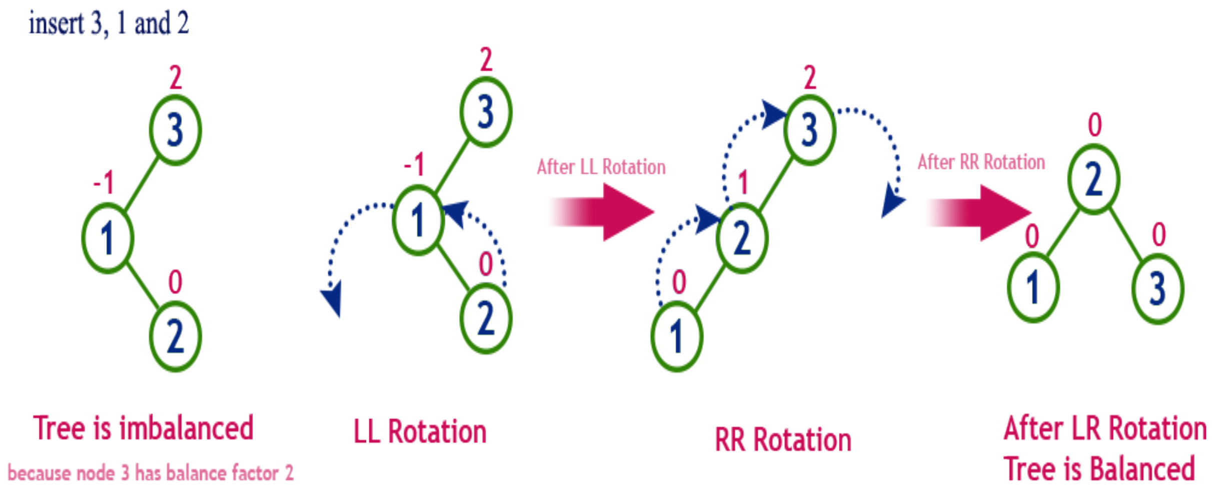


Single Rotation with left (K_3)

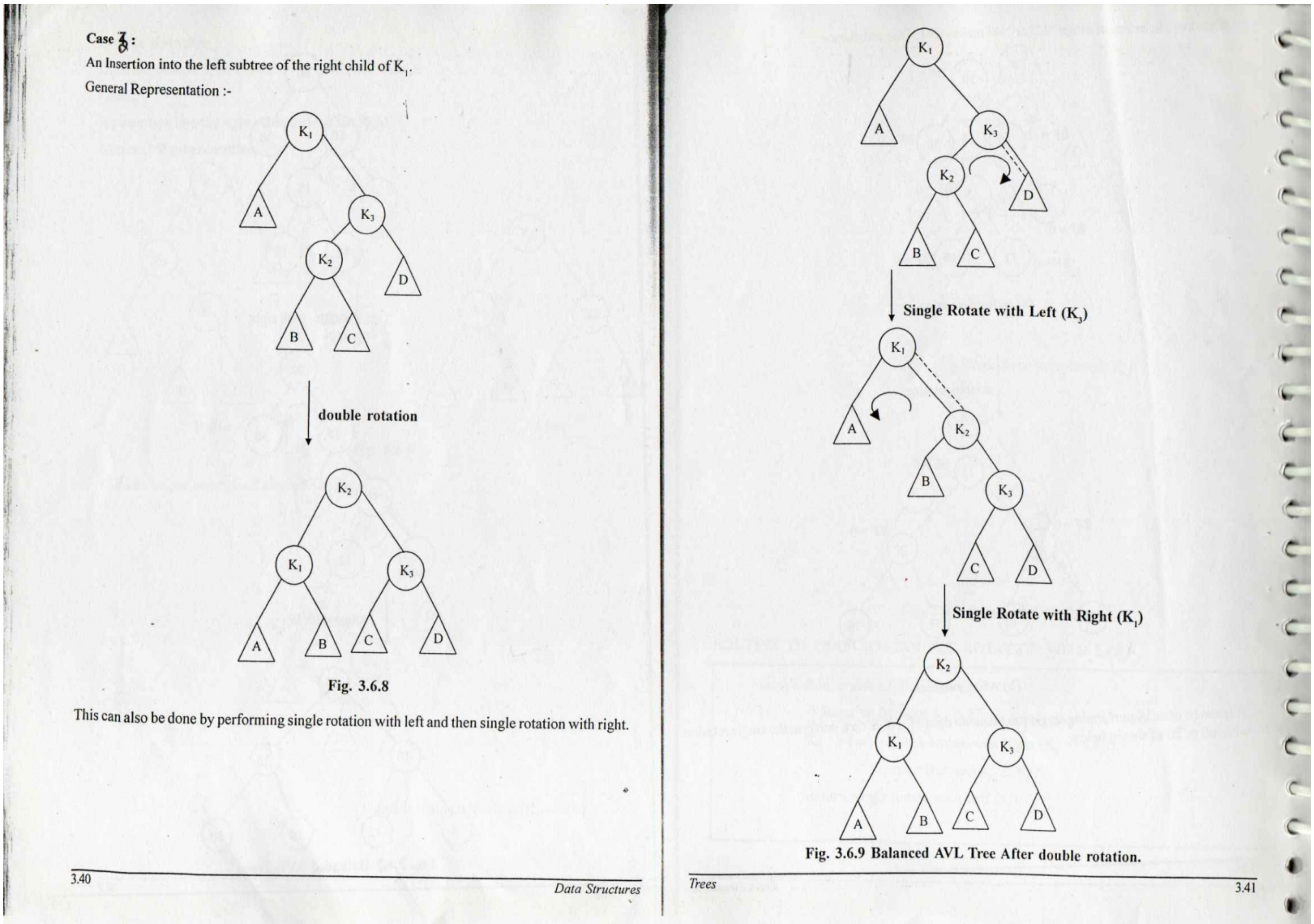
Fig. 3.6.7 Balanced AVL Tree

ROUTINE TO PERFORM DOUBLE ROTATION WITH LEFT :

```
Double Rotate with left (Position K3)
{       /* Rotation Between K1 & K2 */
        K3 → Left = Single Rotate with Right (K3 → Left);
                /* Rotation Between K3 & K2 */
        Return Single Rotate With Left (K3);
}
```



4. Right Left Rotation (RL Rotation)



Case 7:

An Insertion into the left subtree of the right child of K_1.

General Representation :-

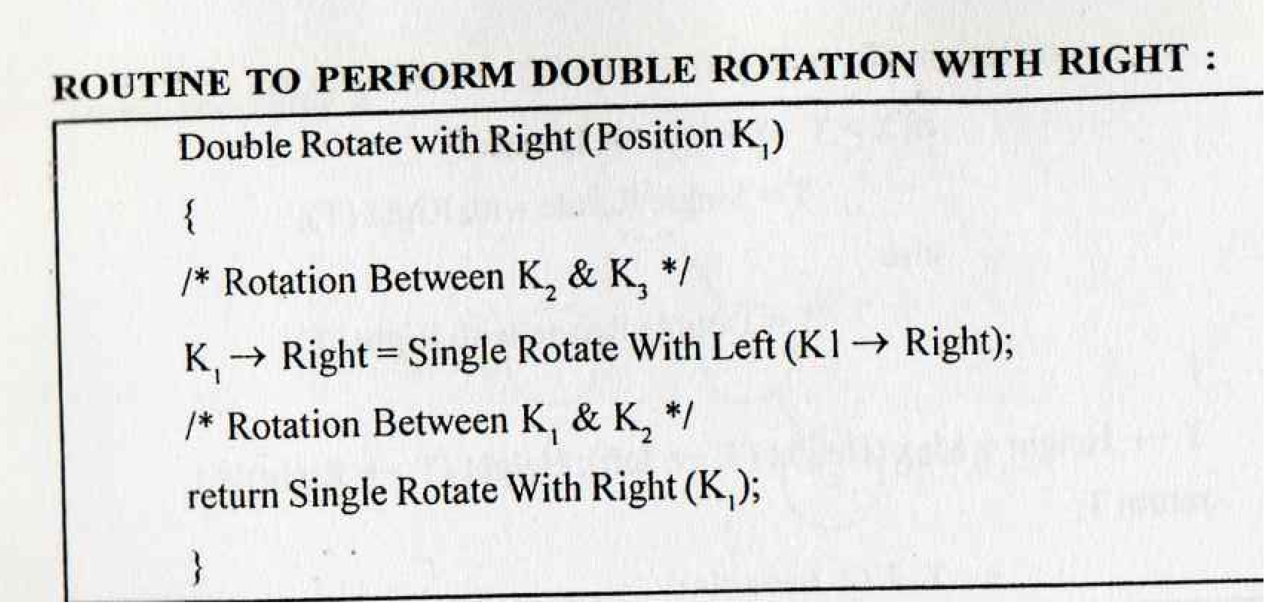
K1
K3
K2
A
B
C
D
double rotation

Fig. 3.6.8

This can also be done by performing single rotation with left and then single rotation with right.

K1
K3
K2
A
B
C
D
Single Rotate with Left (K_3)
Single Rotate with Right (K_1)

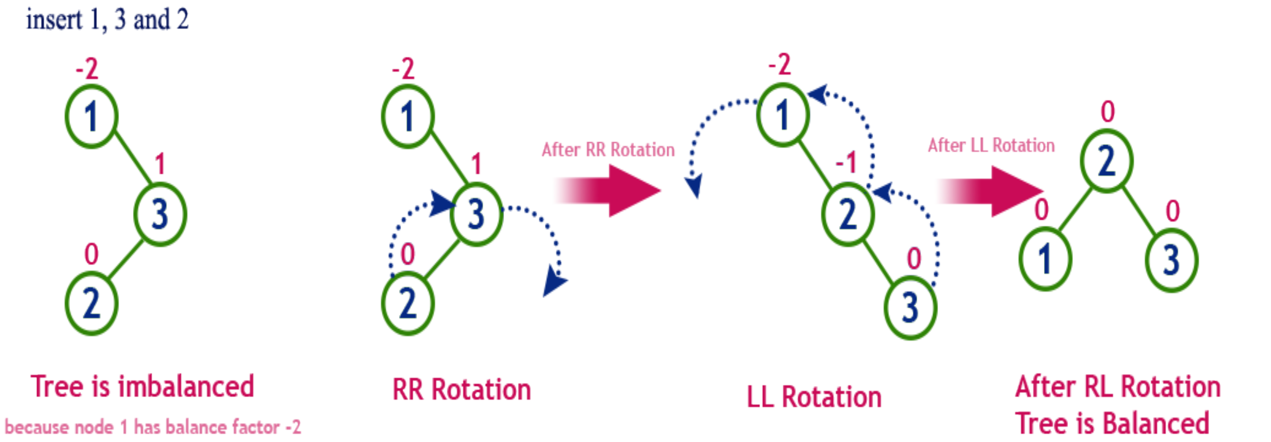
Fig. 3.6.9 Balanced AVL Tree After double rotation.



ROUTINE TO PERFORM DOUBLE ROTATION WITH RIGHT :

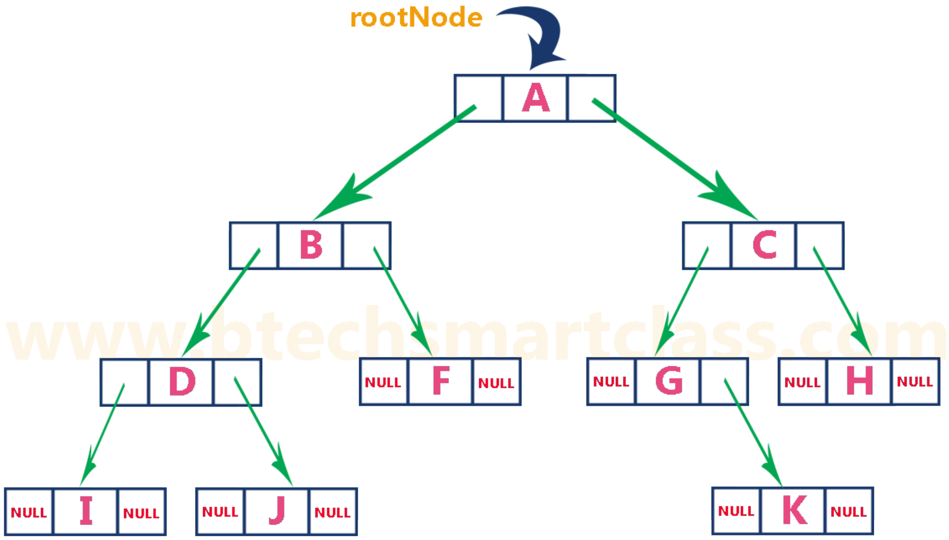
```
Double Rotate with Right (Position K1)
{
/* Rotation Between K2 & K3 */
K1 → Right = Single Rotate With Left (K1 → Right);
/* Rotation Between K1 & K2 */
return Single Rotate With Right (K1);
}
```

- The RL Rotation is sequence of single right rotation followed by single left rotation.
- In RL Rotation, at first every node moves one position to right and one position to left from the current position.
- To understand RL Rotation, let us consider the following insertion operation in AVL Tree...



Representation of AVL Tree

```
Struct AVLNode

{

   int data;

   struct AVLNode *left,

    struct AVLNode *right;

   int balfactor;

};
```



Operations on an AVL Tree

The following operations are performed on AVL tree...

1. Search: The search operation in the AVL tree is similar to the search operation in a Binary search tree.

Step 1 - Read the search element from the user.

Step 2 - Compare the search element with the value of root node in the tree.

Step 3 - If both are matched, then display "Given node is found!!!" and terminate the function

Step 4 - If both are not matched, then check whether search element is smaller or larger than that node value.

Step 5 - If search element is smaller, then continue the search process in left subtree.

Step 6 - If search element is larger, then continue the search process in right subtree.

Step 7 - Repeat the same until we find the exact element or until the search element is compared with the leaf node.

Step 8 - If we reach to the node having the value equal to the search value, then display "Element is found" and terminate the function.

Step 9 - If we reach to the leaf node and if it is also not matched with the search element, then display "Element is not found" and terminate the function.

2. Insertion:

In AVL Tree, a new node is always inserted as a leaf node. The insertion operation is performed as follows...

Step 1 - Insert the new element into the tree using Binary Search Tree insertion logic.

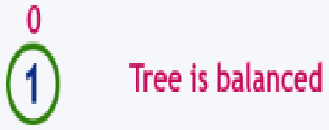
Step 2 - After insertion, check the Balance Factor of every node.

Step 3 - If the Balance Factor of every node is 0 or 1 or -1 then go for next operation.

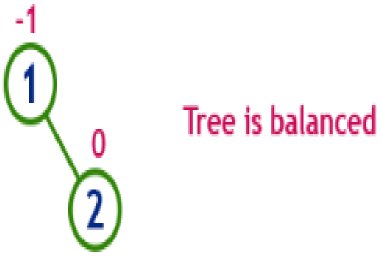
Step 4 - If the Balance Factor of any node is other than 0 or 1 or -1 then that tree is said to be imbalanced. In this case, perform suitable Rotation to make it balanced and go for next operation.

Example: Construct an AVL Tree by inserting numbers from 1 to 8.

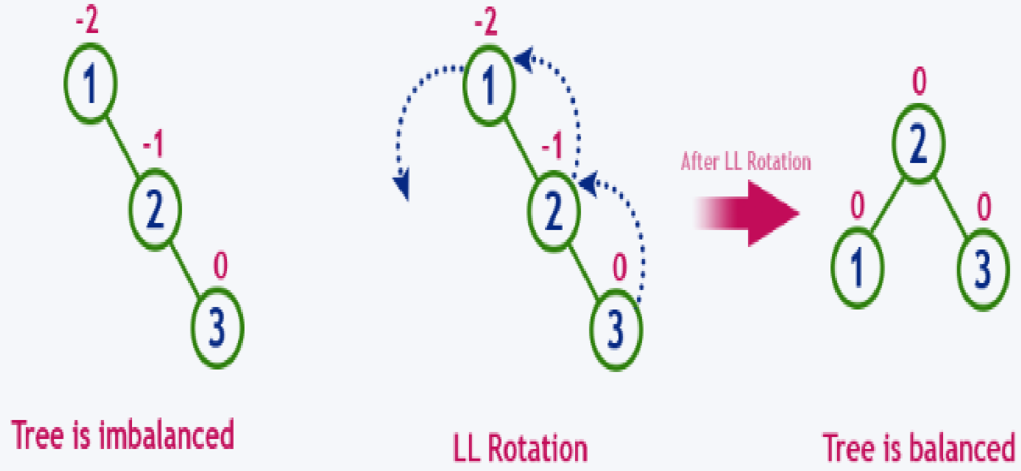
insert 1



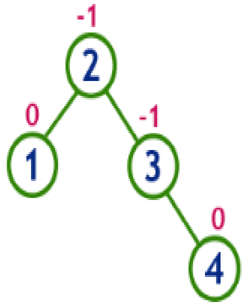
insert 2



insert 3

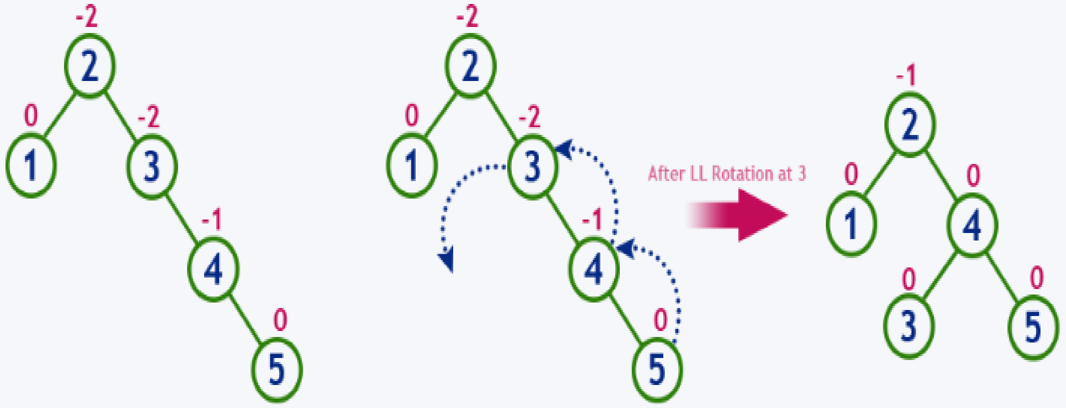


insert 4



Tree is balanced

insert 5

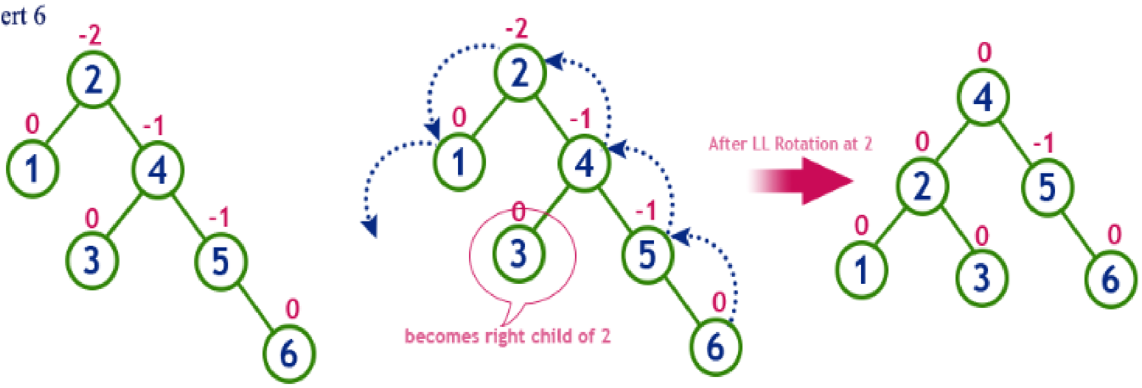


Tree is imbalanced

LL Rotation at 3

Tree is balanced

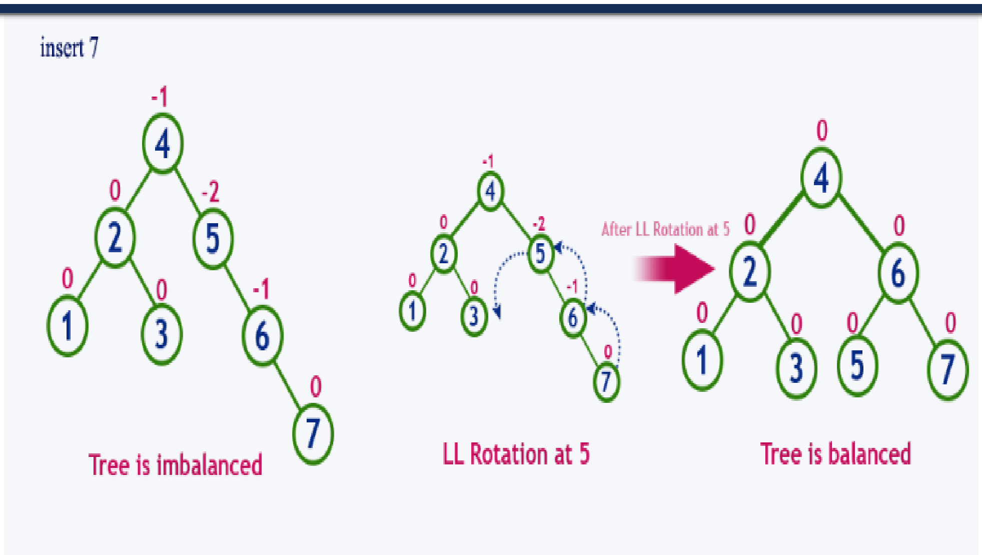
insert 6

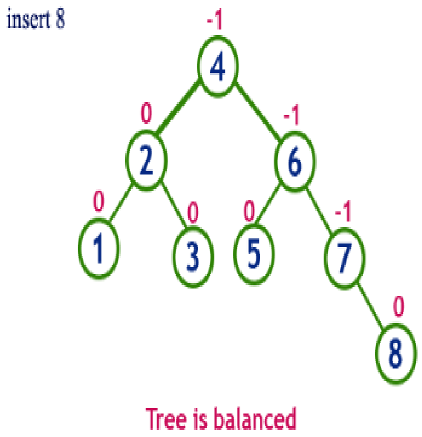


Tree is imbalanced

LL Rotation at 2

Tree is balanced



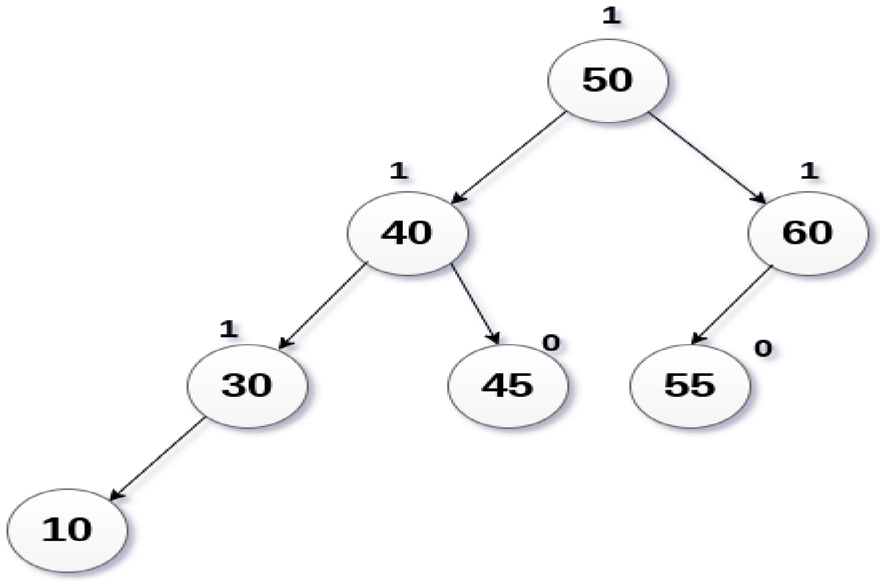


3. Deletion:

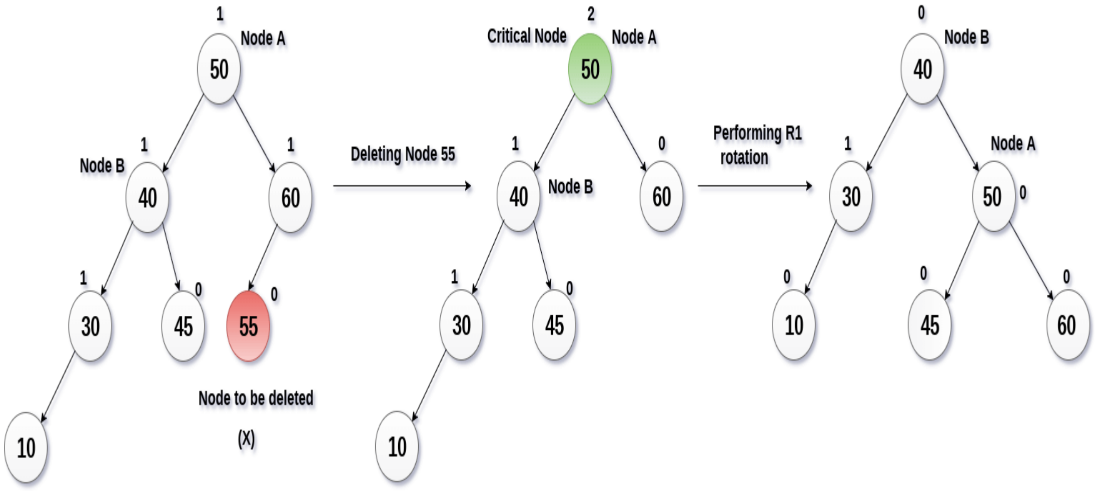
- The deletion operation in AVL Tree is similar to deletion operation in BST.
- But after every deletion operation, we need to check with the Balance Factor condition.
- If the tree is balanced after deletion go for next operation otherwise perform suitable rotation to make the tree Balanced.

Example

Delete Node 55 from the AVL tree shown in the following image.



AVL Tree



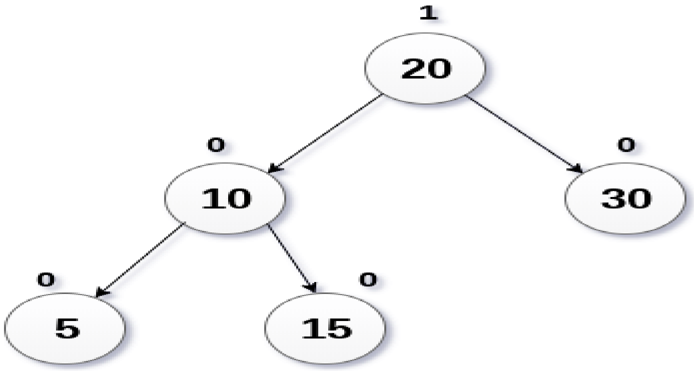
AVL Tree

Non AVL Tree

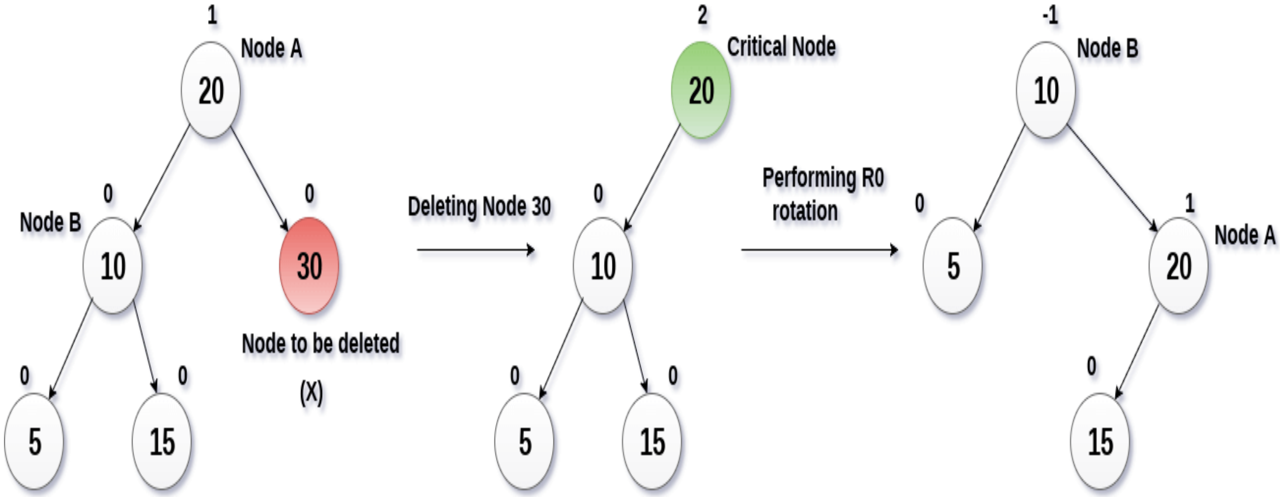
R1 Rotated Tree

Example 2:

Delete the node 30 from the AVL tree shown in the following image.



SOLUTION:



AVL Tree   Non AVL Tree   R0 Rotated Tree

BALANCE FACTOR:

- **Height Of Left Subtree – Height Of Right Subtree = {-1,0,1}**

```
int BF(node *T)
{
        int lh,rh;
        if(T==NULL)
                return(0);

        if(T->left==NULL)
                lh=0;
        else
                lh=1+T->left->ht;

        if(T->right==NULL)
                rh=0;
        else
                rh=1+T->right->ht;

        return(lh-rh);
}
```

<u>HEIGHT</u>

```
int height(node *T)
{
        int lh,rh;
        if(T==NULL)
                return(0);

        if(T->left==NULL)
                lh=0;
        else
                lh=1+T->left->ht;

        if(T->right==NULL)
                rh=0;
        else
                rh=1+T->right->ht;

        if(lh>rh)
                return(lh);

        return(rh);
}
```

H.W

2 4 6 8 10 12 16 20 25 34

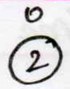
8 5 10 15 20 18 3

AVL Tree

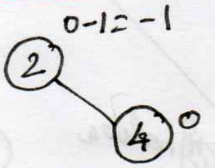
Insertion : 2 4 6 8 10 12 16 20 25 34

$BF = \{HLST - HRST\} = \{-1, 0, +1\}$

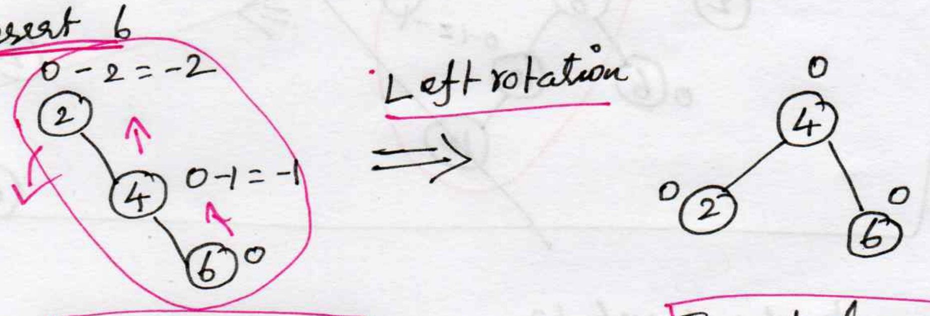
step 1 : insert 2



step 2 : insert 4



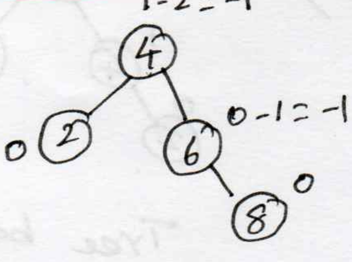
step 3 : insert 6



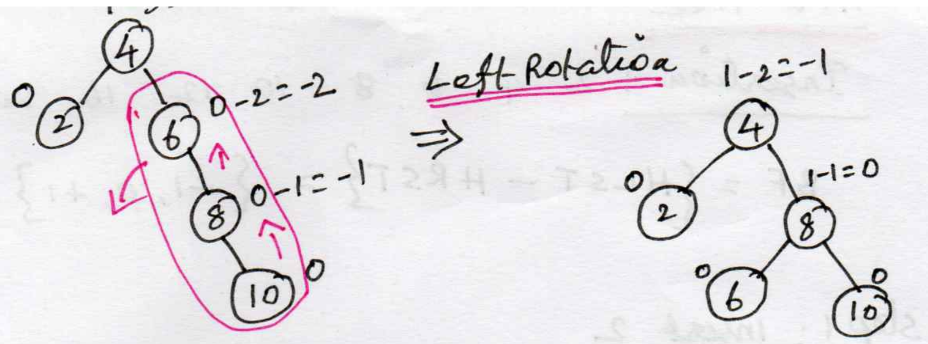
Tree imbalanced.

Tree balanced.

step 4 : Insert 8

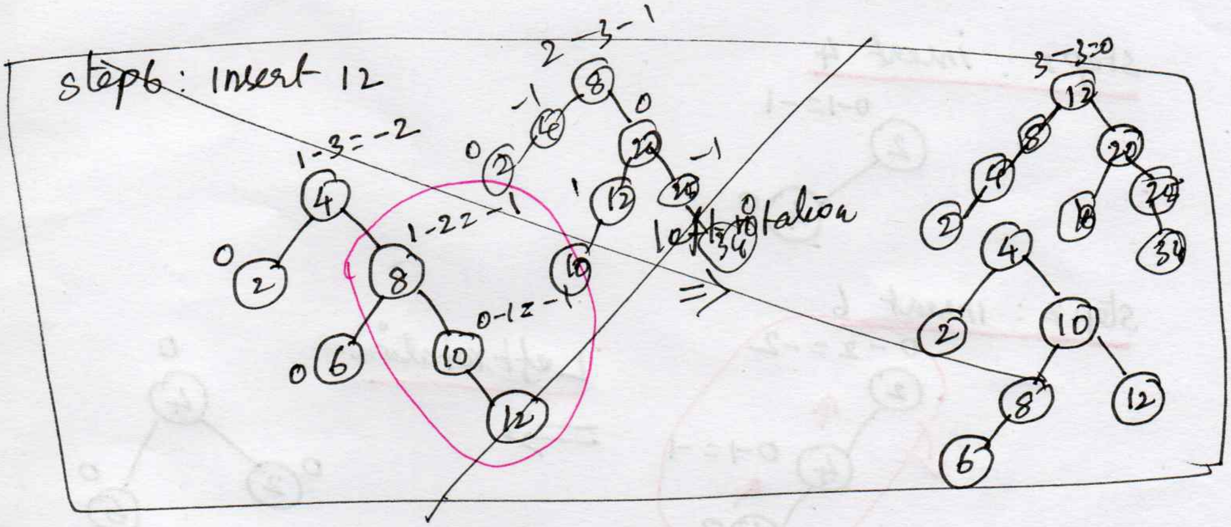


~~step5~~

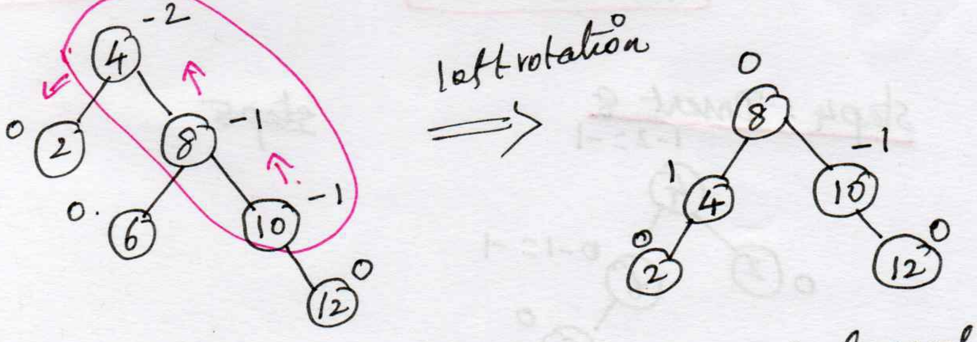


Tree imbalanced

Tree balanced



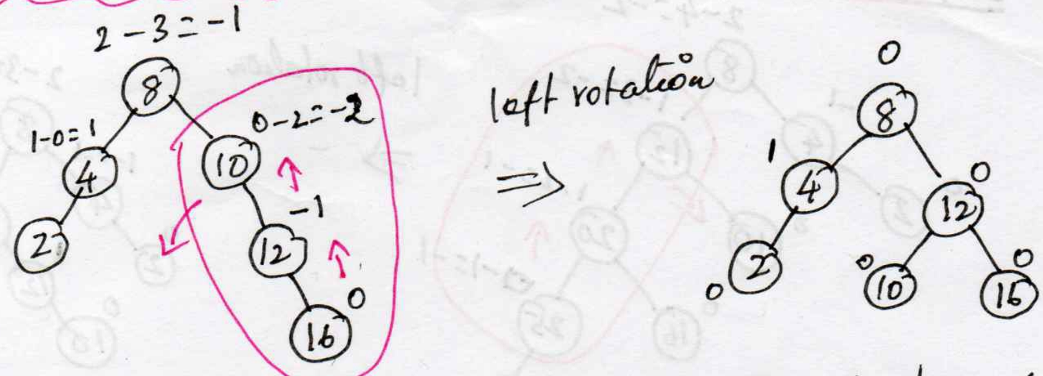
step6: Insert 12



Tree imbalanced.

Tree balanced.

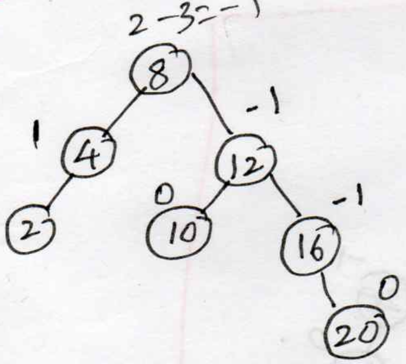
step 7 : insert 16



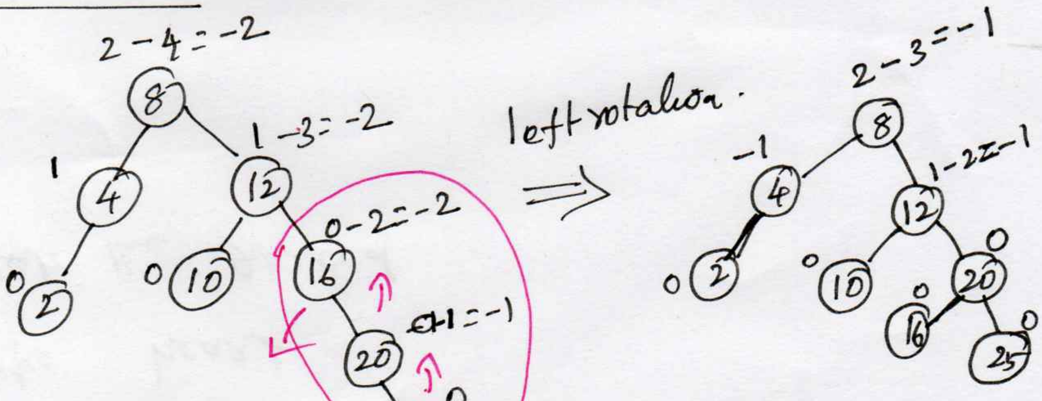
Tree imbalanced

Tree balanced

steps : insert 20.



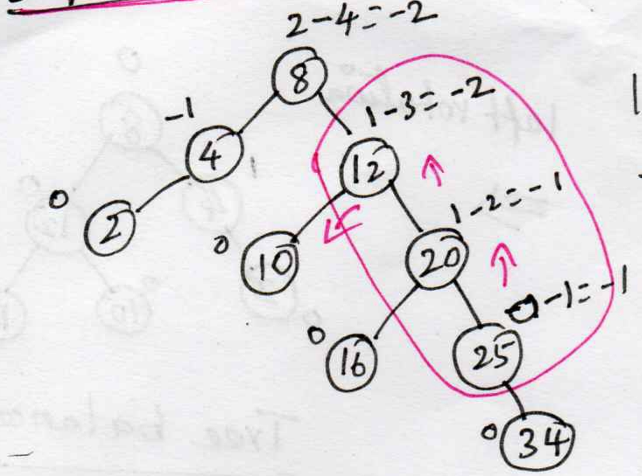
step 9 : insert 25



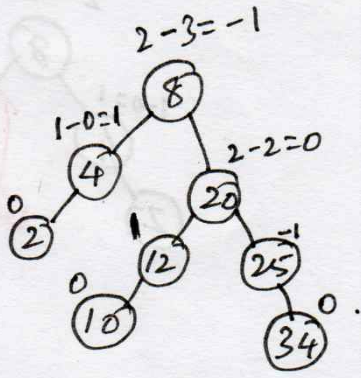
Tree imbalanced

Tree balanced.

Step 10 : insert-34



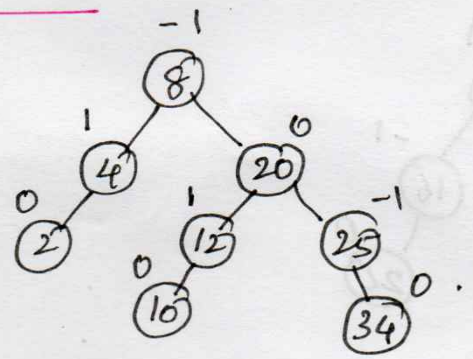
left rotation ⟹



Imbalanced

Balanced

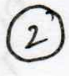
Final o/p



Example 2

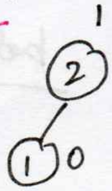
2, 1, 4, 5, 9, 3, 6, 7

step 1 : 2



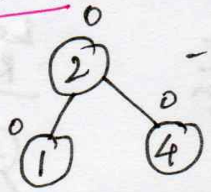
- balanced

step 2 : 1



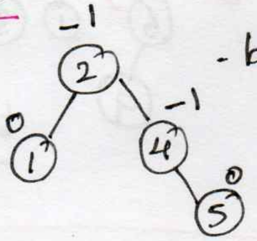
- balanced

step 3 : 4



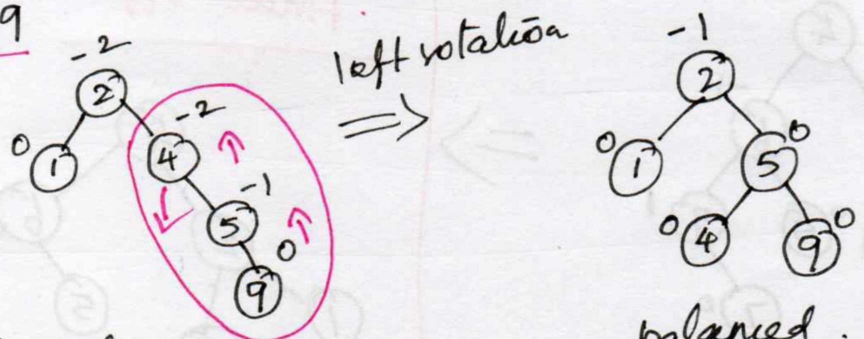
- balanced

step 5 : 5



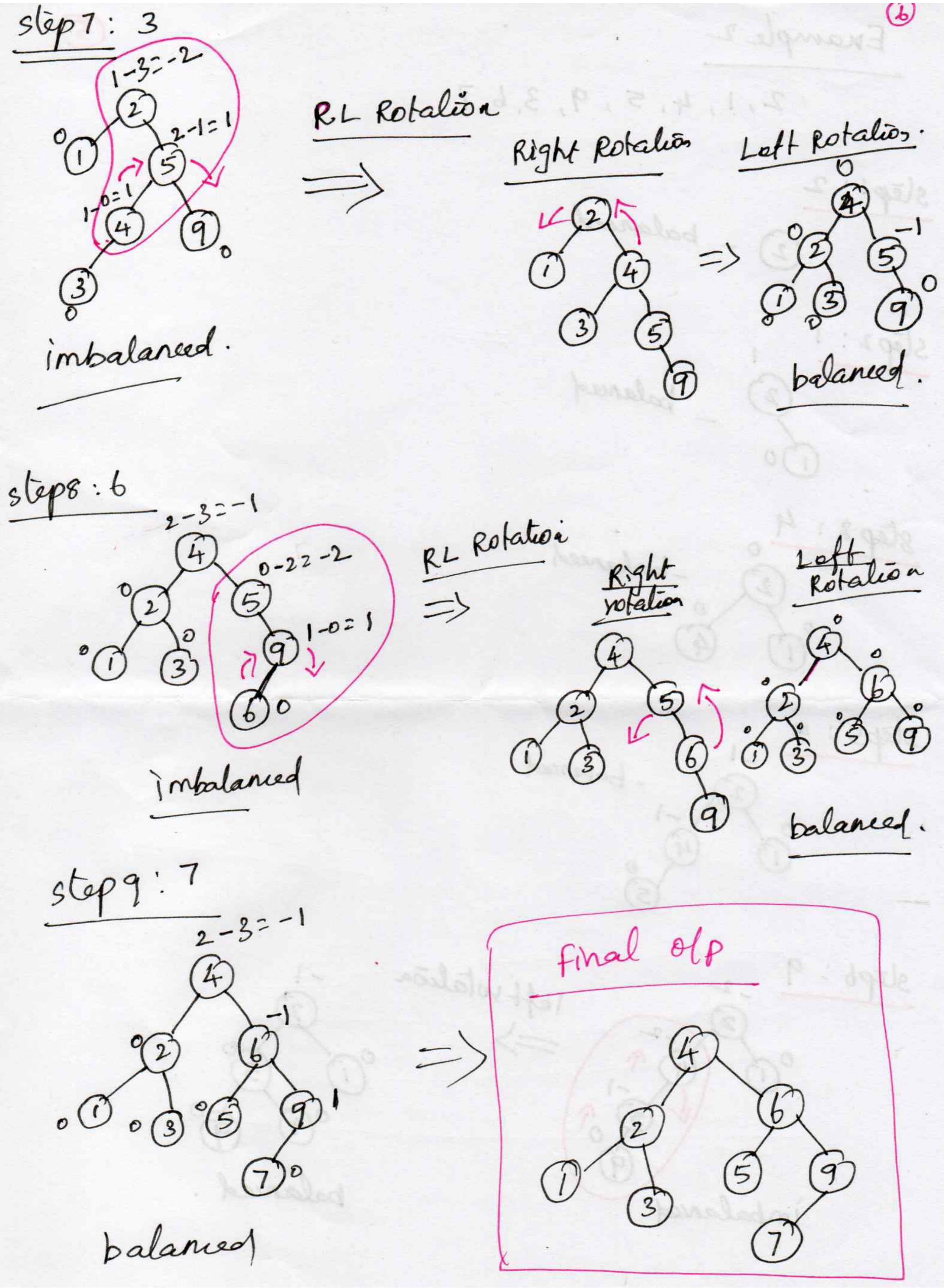
- balanced

step 6 : 9



imbalanced

balanced.


6
step7: 3
1-3=-2
2-1=1
1-0=1
0
imbalanced.
RL Rotation
Right Rotation
Left rotation.
-1
balanced.
steps : 6
2-3=-1
0-2=-2
1-0=1
imbalanced
RL Rotation
Right rotation
Left Rotation
balanced.
step9: 7
2-3=-1
-1
balanced
final o/p

Splay Tree

- Splay tree is another variant of a binary search tree.
- In a splay tree, recently accessed element is placed at the root of the tree. A splay tree is .
    - **Splay Tree is a self - adjusted Binary Search Tree in which every operation on element rearranges the tree so that the element is placed at the root of the tree.**

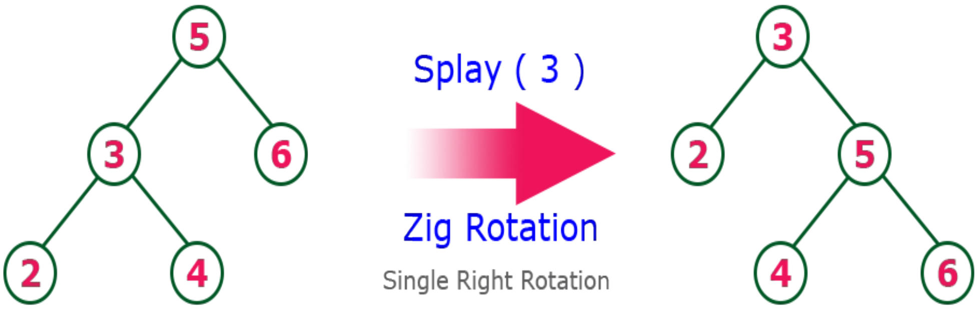
- All the operations in splay tree are involved with a common operation called **"Splaying"**.
- Splaying an element is the process of bringing it to the root position by performing suitable **rotation operations**.
- Every operation on splay tree performs the splaying operation.
- For example, the **insertion operation** first inserts the new element using the binary search tree insertion process, then the newly inserted element is splayed so that it is placed at the root of the tree.
- The **search operation** in a splay tree is nothing but searching the element using binary search process and then splaying that searched element so that it is placed at the root of the tree.
- In splay tree, to splay any element we use the following rotation operations...

<u>Rotations in Splay Tree</u>

1. Zig Rotation(**single right rotation)**
2. Zag Rotation **(single left rotation)**
3. Zig - Zig Rotation (**Double right rotation)**
4. Zag - Zag Rotation (**Double left rotation)**
5. Zig - Zag Rotation (**Double right left rotation)**
6. Zag - Zig Rotation **(Double left right rotation)**

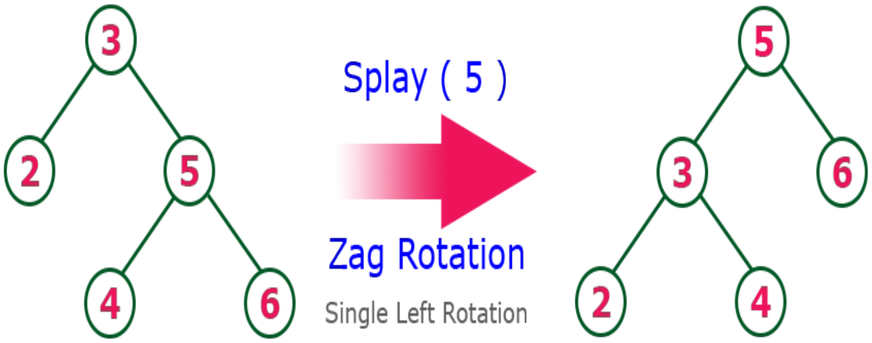
1. Zig Rotation

- The Zig Rotation in splay tree is similar to the **single right rotation** in AVL Tree rotations.
- In zig rotation, every node moves one position to the right from its current position. Consider the following example...



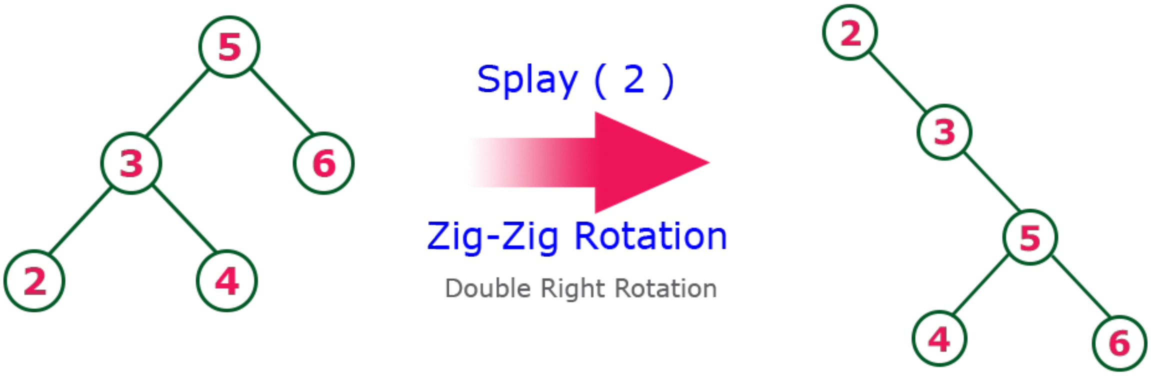
2. Zag Rotation

- The Zag Rotation in splay tree is similar to the single left rotation in AVL Tree rotations.
- In zag rotation, every node moves one position to the left from its current position. Consider the following example...



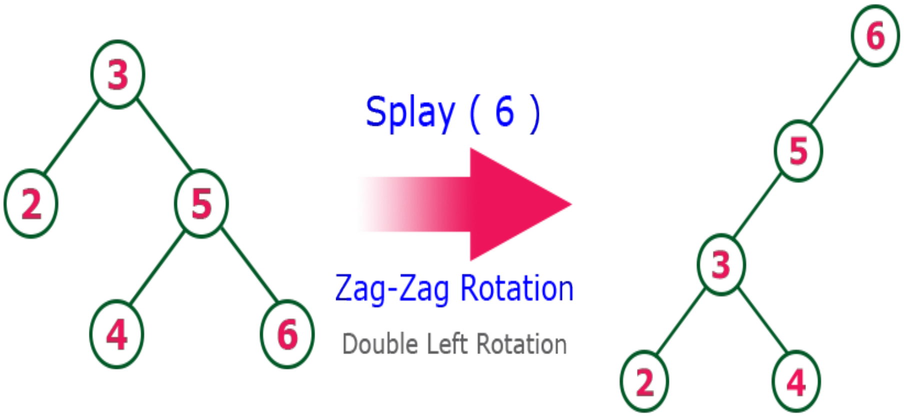
3. Zig-Zig Rotation

- The Zig-Zig Rotation in splay tree is a double zig rotation.
- In zig-zig rotation, every node moves two positions to the right from its current position. Consider the following example...



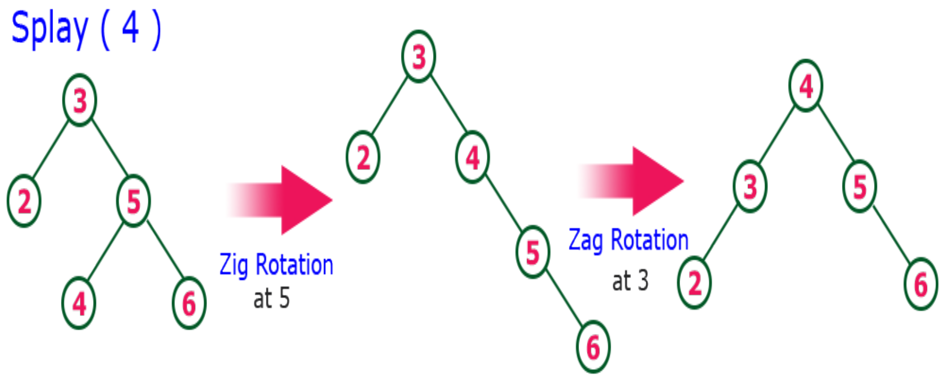
4. Zag-Zag Rotation

- The Zag-Zag Rotation in splay tree is a double zag rotation.
- In zag-zag rotation, every node moves two positions to the left from its current position. Consider the following example...



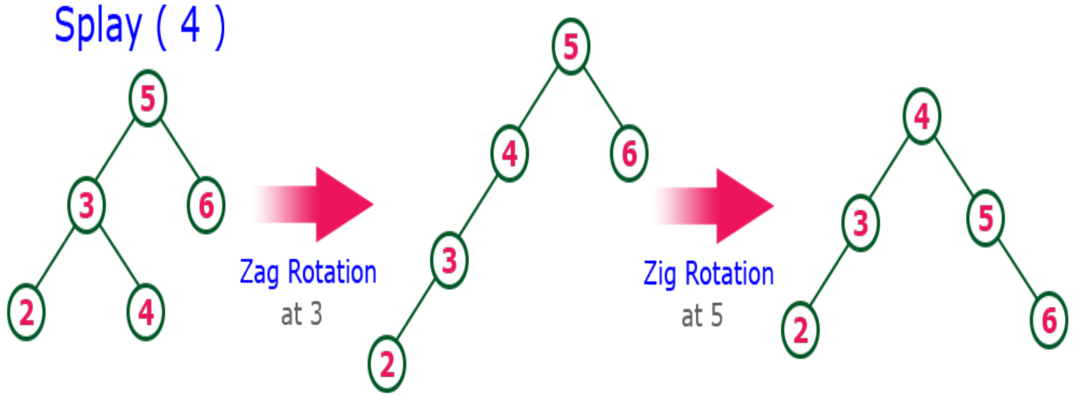
5. Zig-Zag Rotation

- The Zig-Zag Rotation in splay tree is a sequence of zig rotation followed by zag rotation.
- In zig-zag rotation, every node moves one position to the right followed by one position to the left from its current position. Consider the following example...



6. Zag-Zig Rotation

- The Zag-Zig Rotation in splay tree is a sequence of zag rotation followed by zig rotation.
- In zag-zig rotation, every node moves one position to the left followed by one position to the right from its current position. Consider the following example...

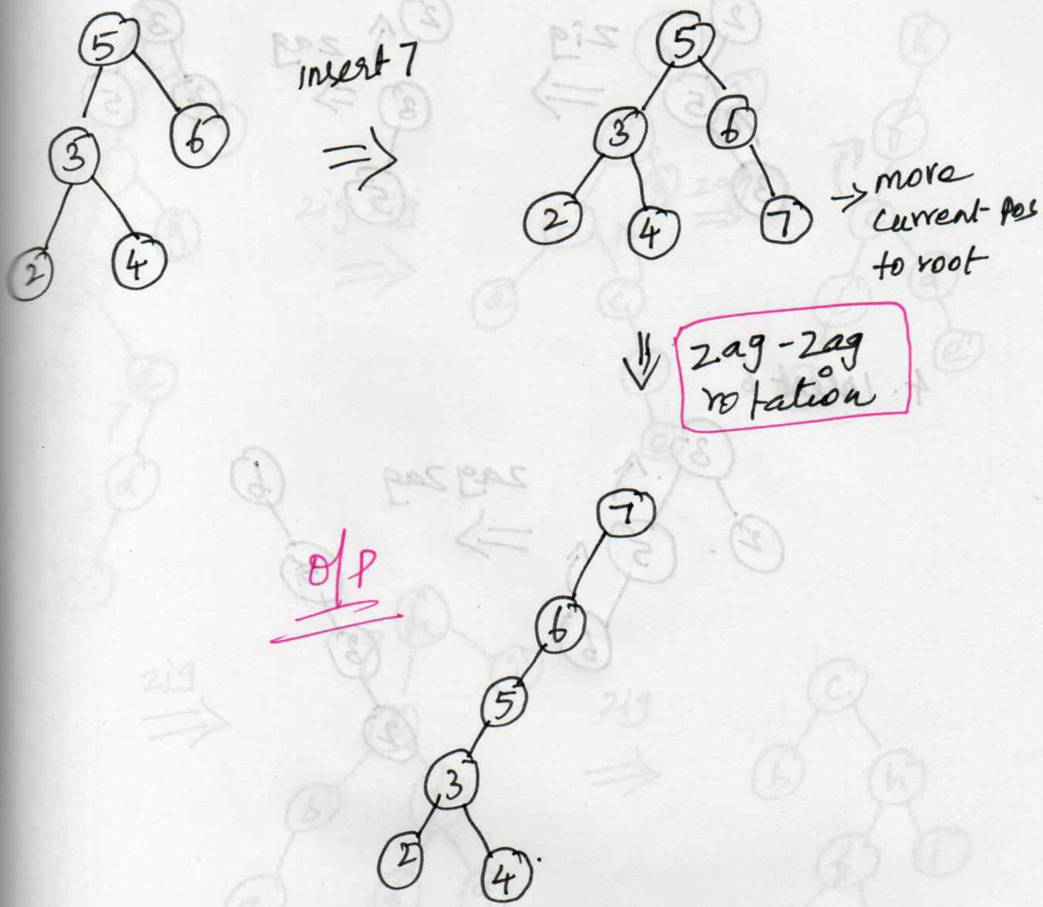


- Every Splay tree must be a binary search tree but it is need not to be balanced tree.

Insertion Operation in Splay Tree

- The insertion operation in Splay tree is performed using following steps...
  Step 1 - Check whether tree is Empty.
  Step 2 - If tree is Empty then insert the newNode as Root node and exit from the operation.
  Step 3 - If tree is not Empty then insert the newNode as leaf node using Binary Search tree insertion logic.
  Step 4 - After insertion, Splay the newNode

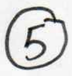
Ex: 1 Insertion



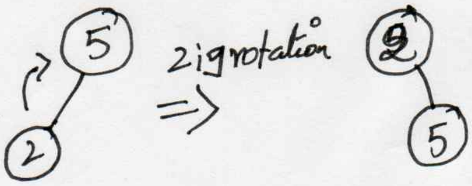
Ex: 2

5 , 2, 3, 6

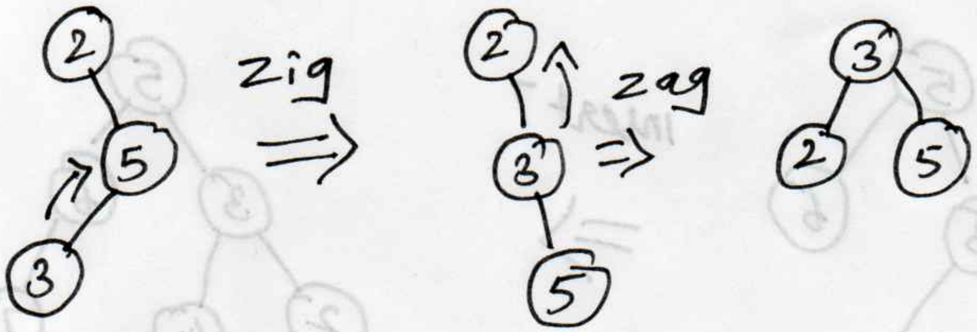
1. insert 5



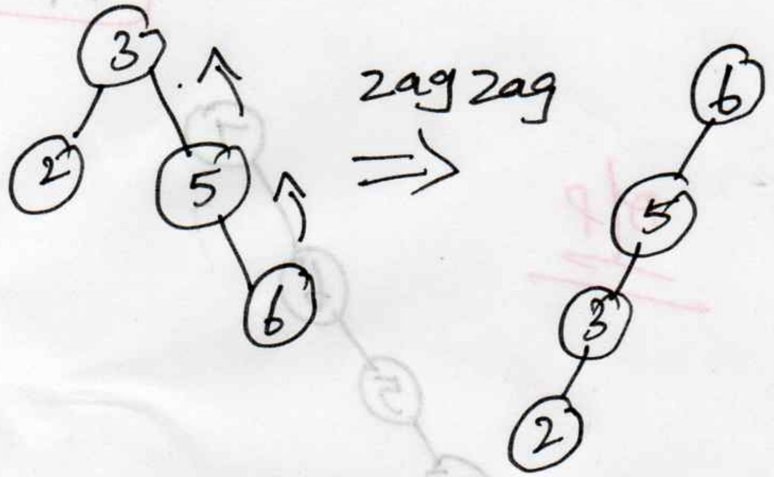
2. insert 2



3. Insert 3



4. Insert 6

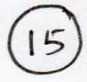


Splay Tree

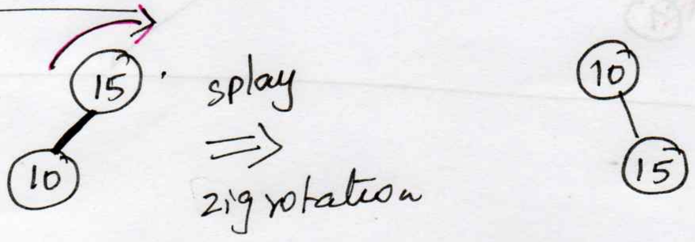
Insertion operation

15, 10, 17, 7, 13, .

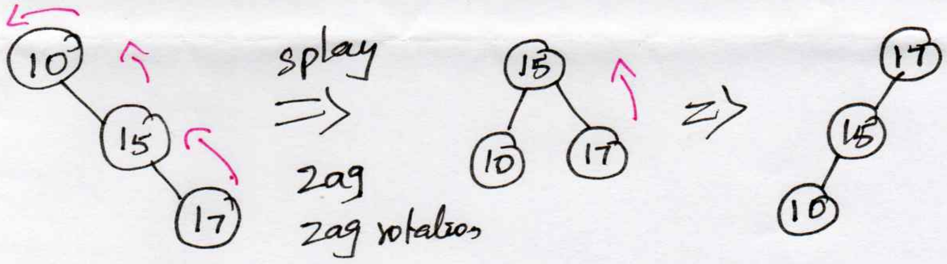
1. insert 15



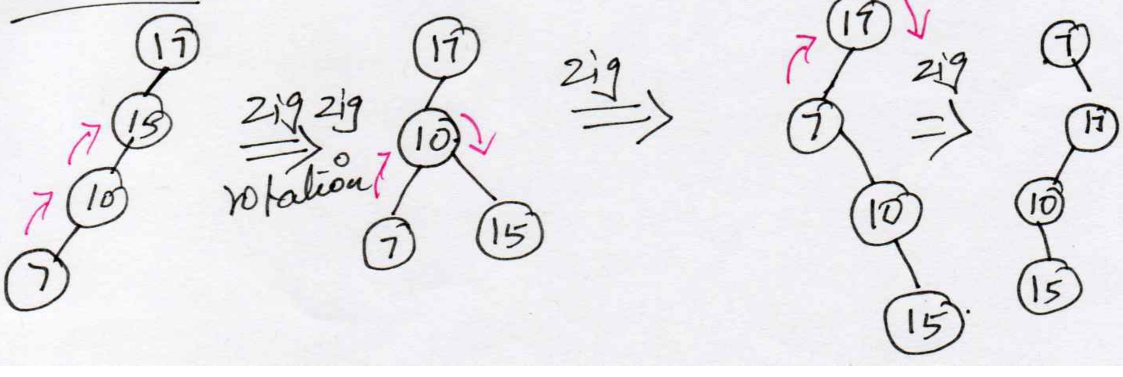
2. Insert 10



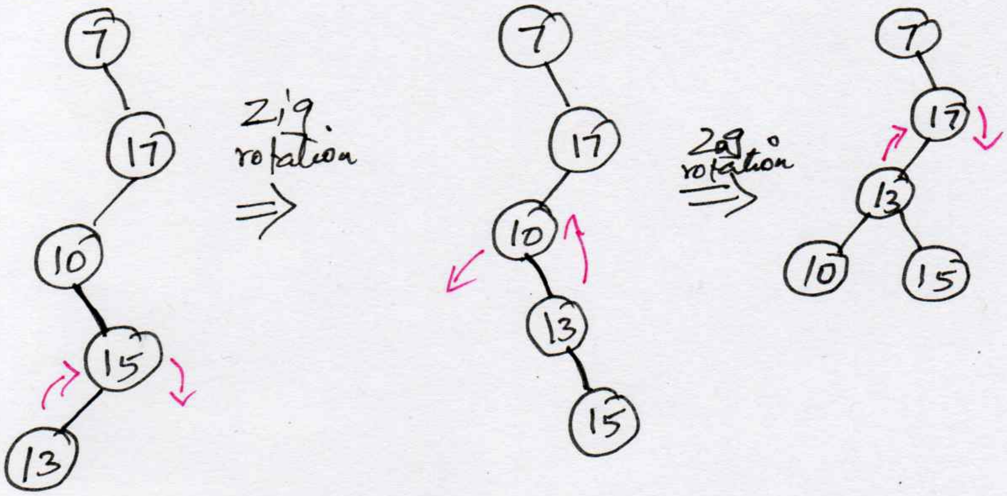
3. Insert 17

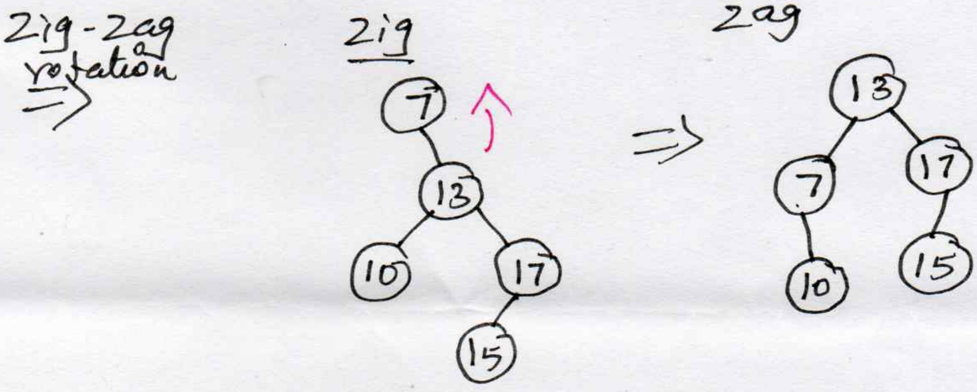


4. Insert 7

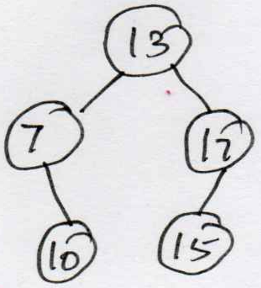


5. insert 13

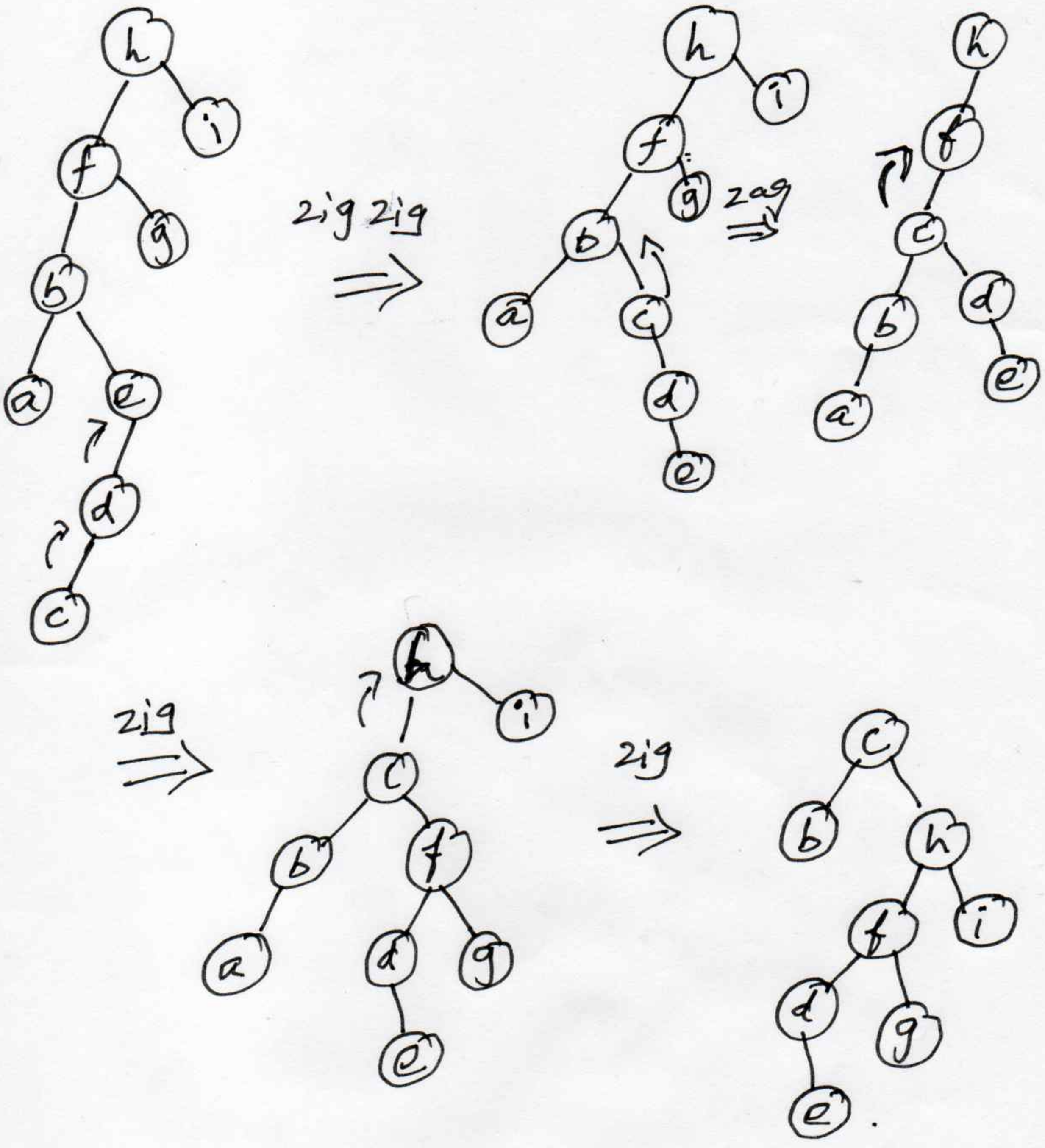




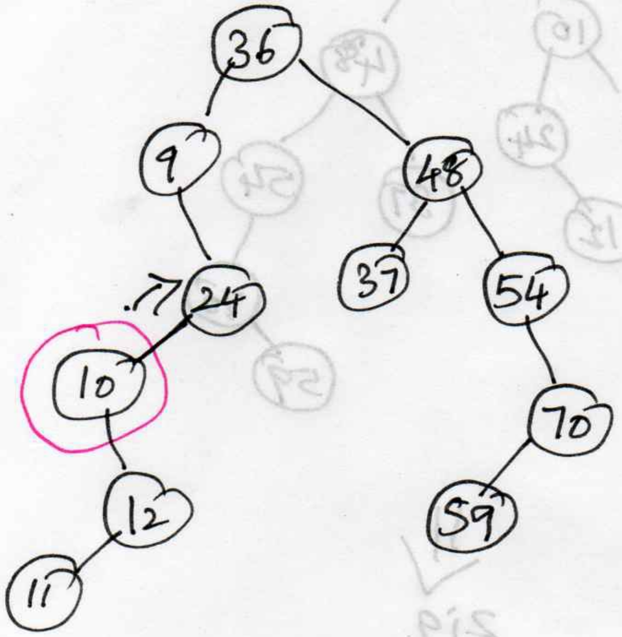
Final Result



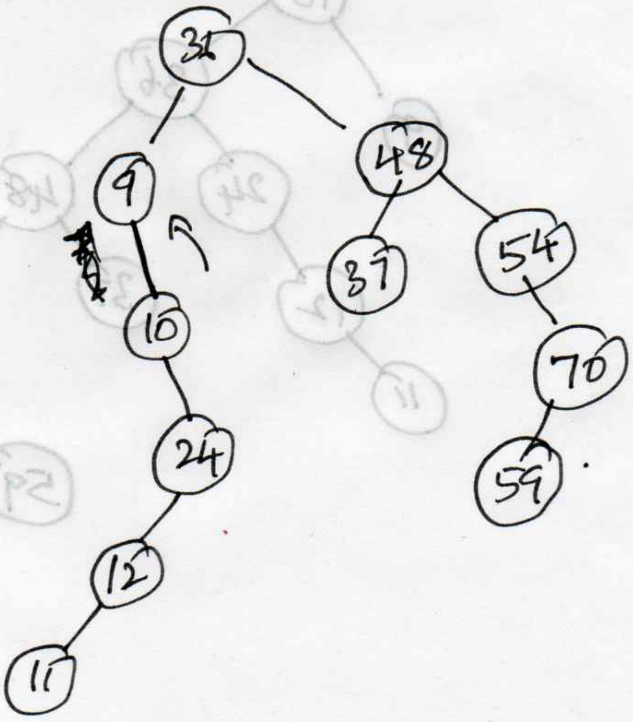
To access c


h
i
f
g
b
a
e
d
c
zig zig
h
i
f
g
b
a
c
d
e
zag
k
f
c
b
d
a
e
zig
h
i
c
b
f
a
d
g
e
zig
c
b
h
f
i
d
g
e

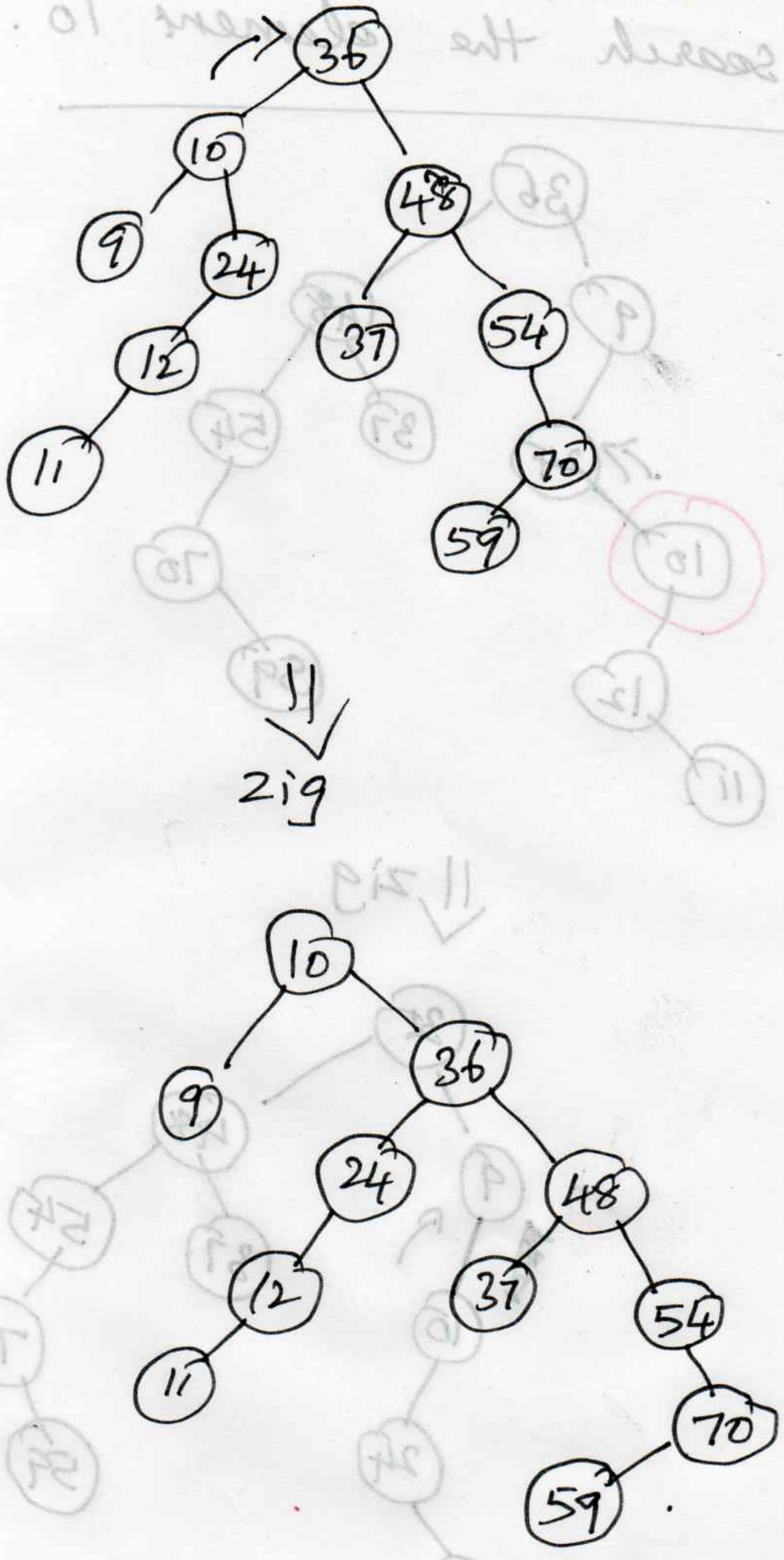
TO search the element 10



⇓ zig

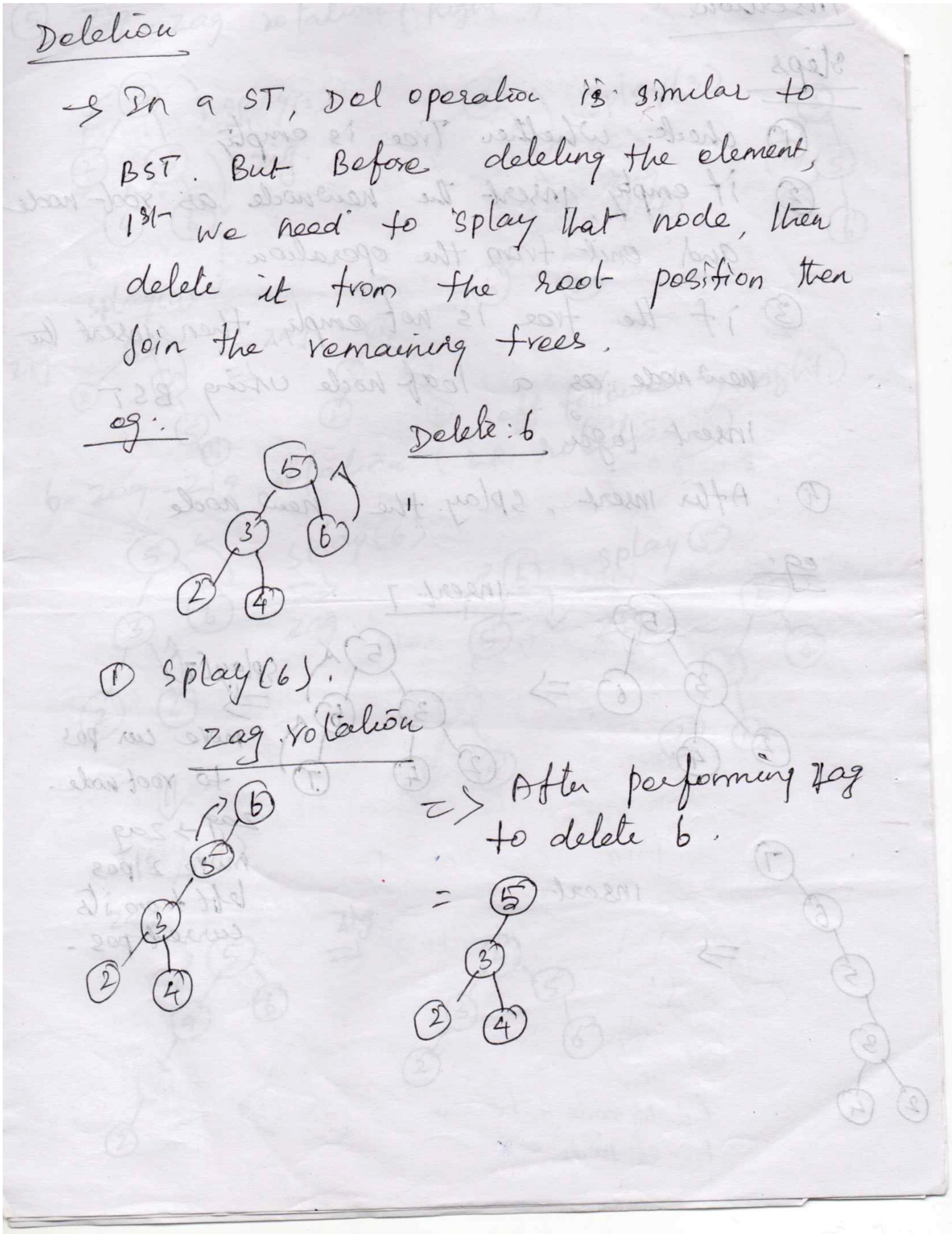


⇓ zag.


zig

<u>Deletion Operation in Splay Tree</u>

- The deletion operation in splay tree is similar to deletion operation in Binary Search Tree.
- But before deleting the element, we first need to splay that element and then delete it from the root position.
- Finally join the remaining tree using binary search tree logic.



Deletion

→ In a ST, Del operation is similar to BST. But Before deleting the element, 1st we need to splay that node, then delete it from the root position then join the remaining trees.

eg:

Delete: 6

5
3
6
2
4

① Splay(6).

Zag rotation

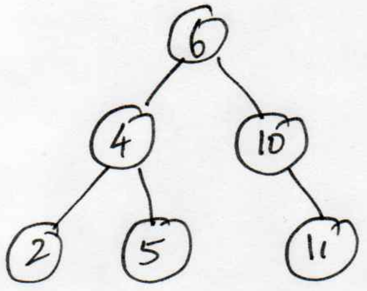
6
5
3
2
4

=> After performing zag to delete 6.

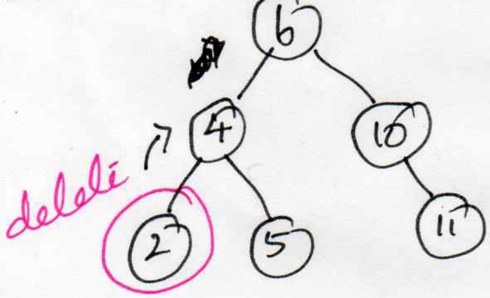
=

5
3
2
4

Deletion

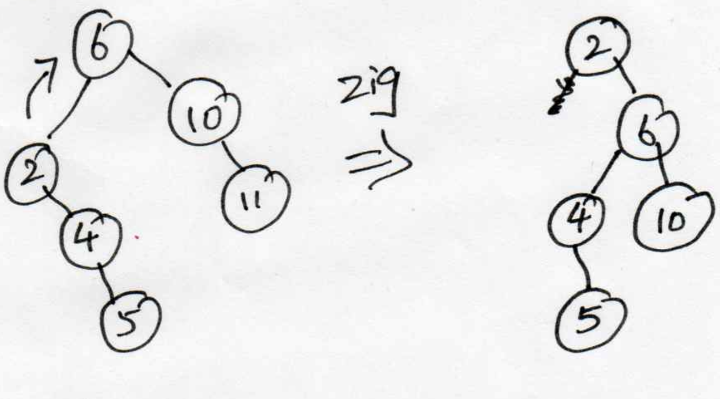


To Delete 2

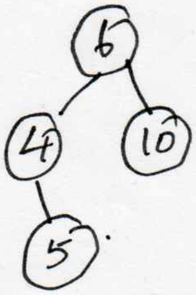


⇓

zig rotation.



Now Delete 6



Final. O/p

H.W

1. 8,17,1,14,16,15 insertion using splay tree.

B - Tree

- In search trees like binary search tree, AVL Tree, etc., every node contains only one value (key) and a maximum of two children.
- But there is a special type of search tree called **B-Tree** in which a node contains more than one value (key) and more than two children.
- B-Tree also called ***Height Balanced m-way Search Tree***. Later it was named as B-Tree. B-Tree can be defined as follows...

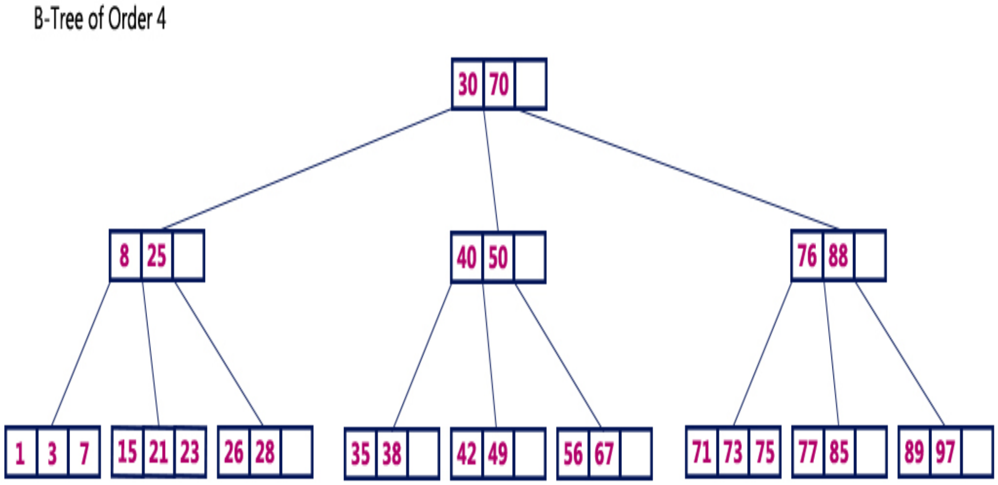
B-Tree is a self-balanced search tree in which every node contains multiple keys and has more than two children.

- The B-Trees are specialized m-way search tree. This can be widely used for disc access.
- A B-tree of order m, can have maximum m-1 keys and m children. This can store large number of elements in a single node. So the height is relatively small. This is one great advantage of B-Trees.

B-Tree of Order m has the following properties...

1. Every node in a B-Tree except root contains at least [m/2]-1 keys and maximum of m-1 keys
2. Every internal node has at least m/2 children.
3. The root node has at least 2 children if it is not leaf.
4. A non leaf node with k children has k-1 keys
5. All leaf nodes must be at the same level.
6. All the key values in a node must be in Ascending Order.

- **For example, B-Tree of Order 4** contains a maximum of 3 key values in a node and maximum of 4 children for a node.



Operations on a B-Tree

The following operations are performed on a B-Tree...

1. **Search**
2. **Insertion**
3. **Deletion**

Search Operation in B-Tree

- The search operation in B-Tree is similar to the search operation in Binary Search Tree.
- In a Binary search tree, the search process starts from the root node and we make a 2-way decision every time (we go to either left subtree or right subtree).
- In B-Tree also search process starts from the root node but here we make an n-way decision every time.
- Where 'n' is the total number of children the node has.

In a B-Tree, search operation is performed as follows...

Step 1 - Read the search element from the user.

Step 2 - Compare the search element with first key value of root node in the tree.

Step 3 - If both are matched, then display "Given node is found!!!" and terminate the function

Step 4 - If both are not matched, then check whether search element is smaller or larger than that key value.

Step 5 - If search element is smaller, then continue the search process in left subtree.

Step 6 - If search element is larger, then compare the search element with next key value in the same node and repeat steps 3, 4, 5 and 6 until we find the exact match or until the search element is compared with last key value in the leaf node.

Step 7 - If the last key value in the leaf node is also not matched then display "Element is not found" and terminate the function.

Insertion Operation in B-Tree

In a B-Tree, a new element must be added only at the leaf node. That means, the new key Value is always attached to the leaf node only. The insertion operation is performed as follows...

Step 1 - Check whether tree is Empty.

Step 2 - If tree is **Empty**, then create a new node with new key value and insert it into the tree as a root node.

Step 3 - If tree is **Not Empty**, then find the suitable leaf node to which the new key value is added using Binary Search Tree logic.

Step 4 - If that leaf node has empty position, add the new key value to that leaf node in ascending order of key value within the node.

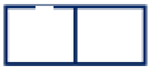
Step 5 - If that leaf node is already full, **split** that leaf node by sending middle value to its parent node. Repeat the same until the sending value is fixed into a node.

Step 6 - If the spilting is performed at root node then the middle value becomes new root node for the tree and the height of the tree is increased by one.

Example:

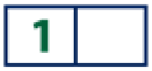
Construct a **B-Tree of Order 3** by inserting numbers from 1 to 10.

- **m=3**
- **max children =m=3,**
- min children: leaf =0,root=2,internal node=m/2=1.5(ceiling) 2
- **max key=m-1=2**

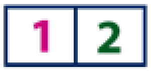


- min key: root=1,other node=[m/2]-1=1.5(ceiling value) 2-1=1

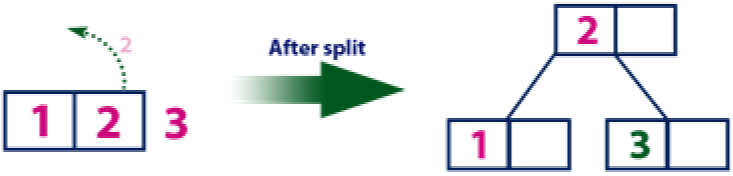
1. **Insert 1**



2. Insert 2

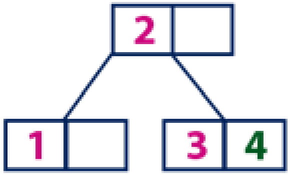


3. Insert 3

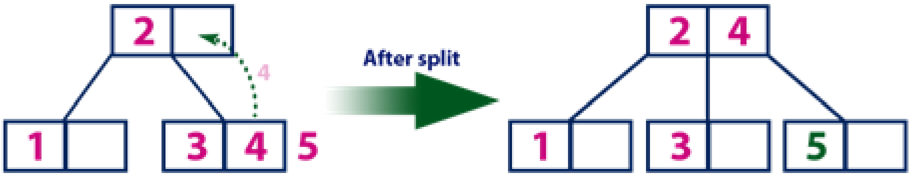


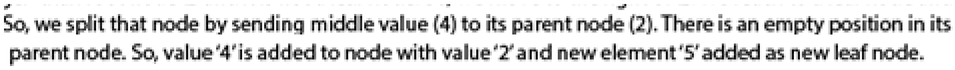
Insert 3, but key value 2, so we split that node by sending middle value 2 to its parent node. But here, this node doesn't has parent. So, this middle value becomes a new root node.

4. Insert 4

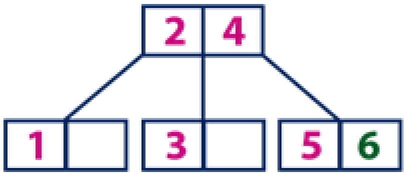


5. Insert 5

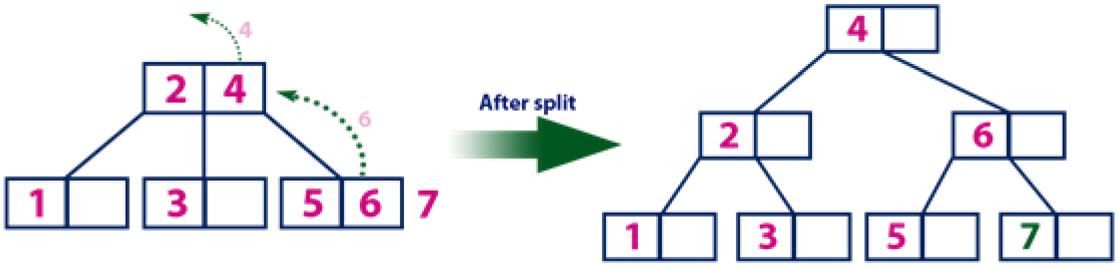


- 
So, we split that node by sending middle value (4) to its parent node (2). There is an empty position in its parent node. So, value '4' is added to node with value '2' and new element '5' added as new leaf node.

6. Insert 6

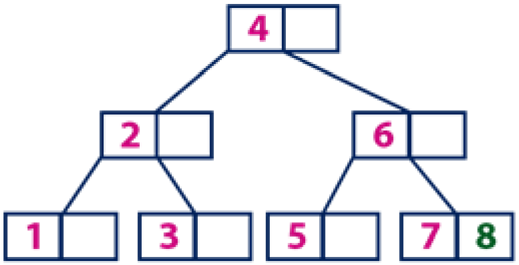


7.Insert 7

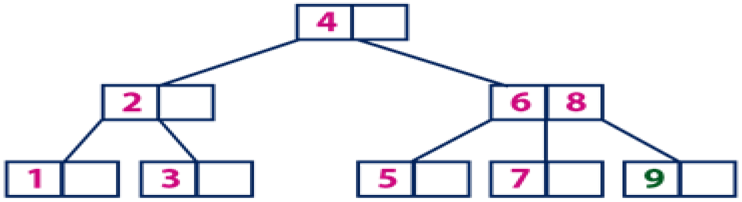


So, we split that node by sending middle value (6) to its parent node (2&4). But the parent (2&4) is also full. So, again we split the node (2&4) by sending middle value '4' to its parent but this node doesn't have parent. So, the element '4' becomes new root node for the tree.

8. Insert 8

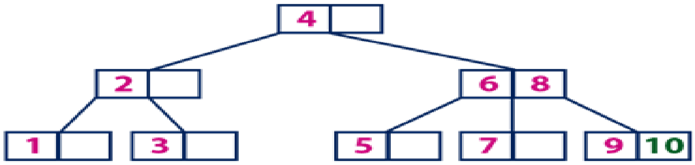


9.Insert 9



So, we split this node by sending middle value (8) to its parent node. The parent node (6) has an empty position. So, '8' is added at that position. And new element is added as a new leaf node.

10. Insert 10



Example 2: construct a B-Tree of order 4 with following set of data

5, 3, 21, 9, 1, 13, 2, 7, 10, 12, 4, 8

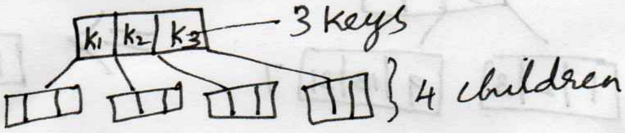
solution

↳ m = 4 (max children)

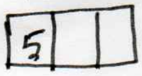
↳ min children : leaf → 0, root → 2, internal node } = m/2 = 4/2 = 2

↳ max key = m - 1 = 3

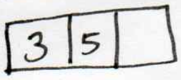
↳ min key : root node → 1, other node m/2 - 1 = 1



1. insert 5

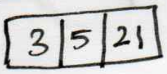


2. Insert 3

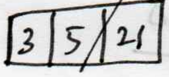


[stored in ascending order]

3. Insert 21

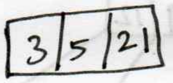


~~4. insert 9~~



~~9 =>~~

4. Insert 9



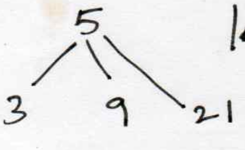
9

→ Find middle value go to 1 level up

→ even middle

3 5 9 21

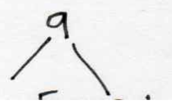
eg : 5 is middle



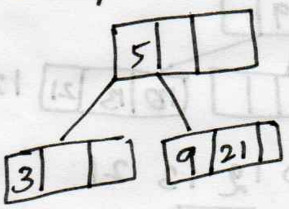
left

both are correct

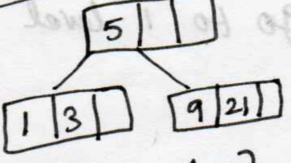
9 is middle



Right.

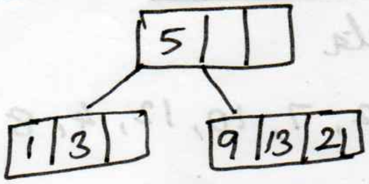


5. insert 1



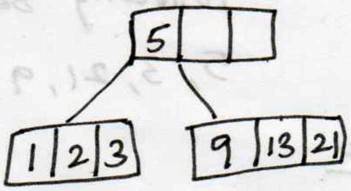
[arrange AO]

6. insert 13



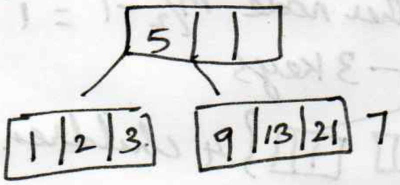
[AO]

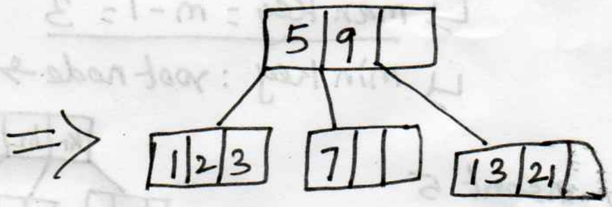
7. insert 2



[Sorted AO]

8. insert 7

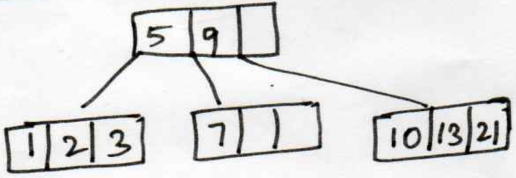




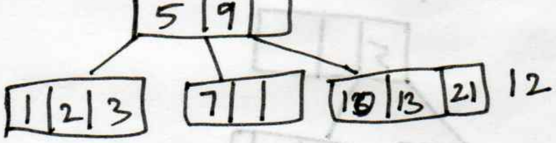
7 9 13 21

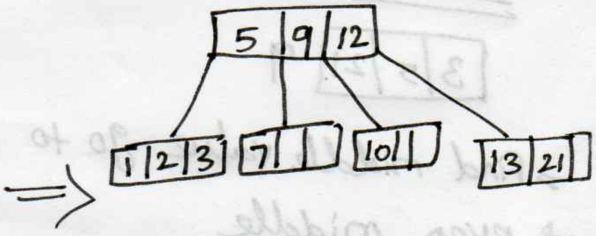
Take middle left 9 → 9 go to 1 level up.

9. insert 10



10. insert 12

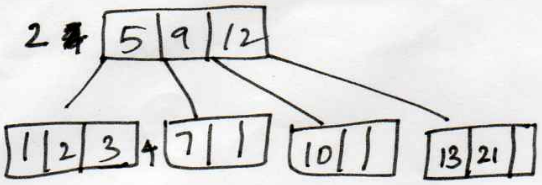




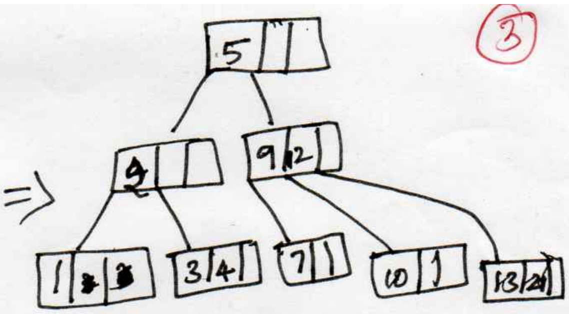
10 12 13 21

Take middle value 12(left) go to 1 level up.

11. Insert 4



⇒



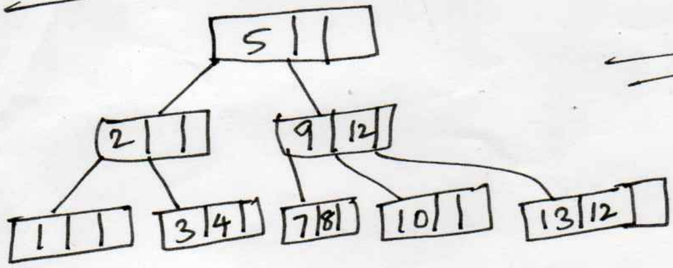
1 2 3 4

2 is middle go to 1 levelup.

2 5 9 12

→ 5 is middle go to 1 levelup.

12. insert 8



⇒ Final o/p

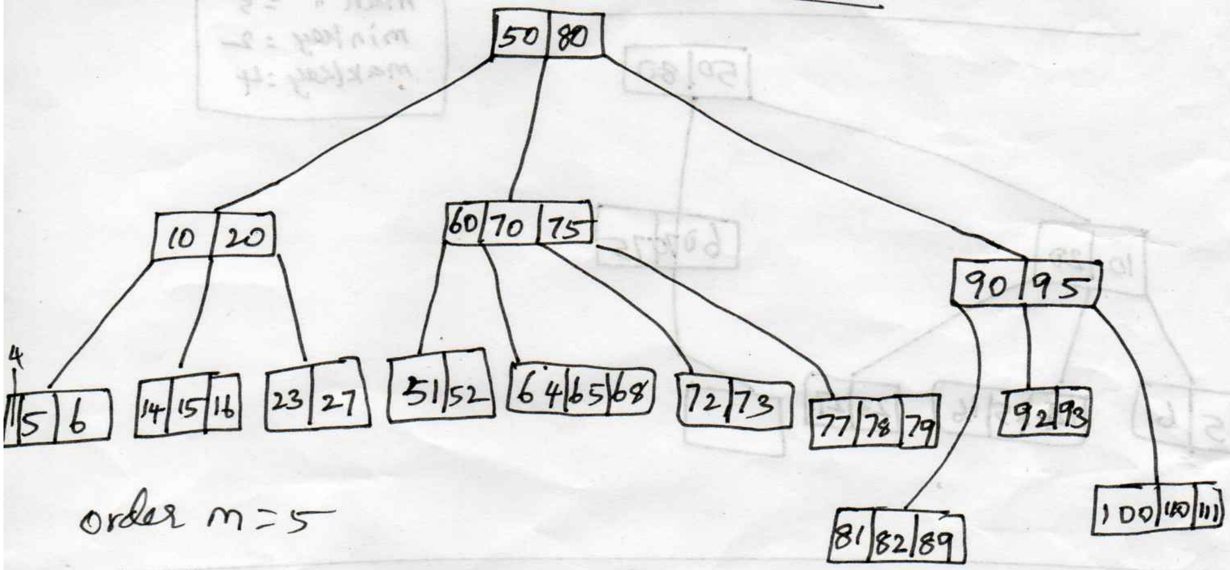
H.W

① To construct a B-Tree of order 5 by inserting values from 1 to 20.

② To construct a B-Tree of order 3 by inserting values 1, 7, 6, 2, 11, 4, 8.

③ To construct a B-Tree of order 4 by inserting values from 60, 70, 75, 51, 52, 65, 68, 77, 78, 79.

Deletion in BTree



order m=5

min ch = 3

max ch = 5

minkey = 2

maxkey = 4

Two possibilites

① if target key is in leaf node

② if target key is in internal nod .

→ leaf node

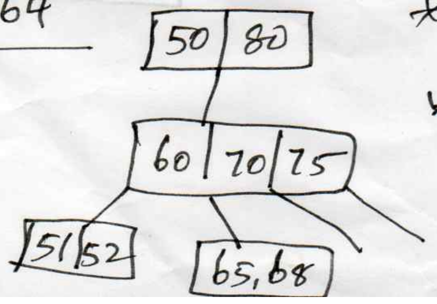
→ Two method

① leaf node contains more than min key

② leaf node contains min key → 3 cases .

leaf node contains more than min key

Delete 64



* to search root node to that node

* leaf node contain more than min key

* so simply delete 64

* does't voilate B Tree

Delete 23

↳ leaf node contains min. key.

→ it can't delete 23 immediatly.

→ it violate B Tree properties.

3 cases

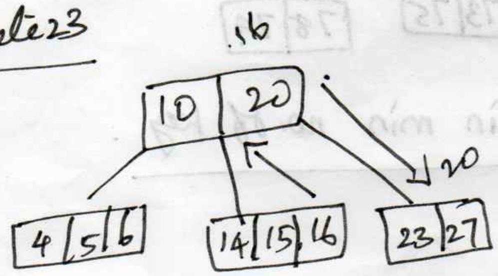
① Borrow immediate left node (sibiling)
↳ contains more than min key.

② Borrow immediate right node.
↳ contains more than min key.

③ merging Left s~~x~~ sibiling & R sibiling.

case 1 Borrow immediate left node

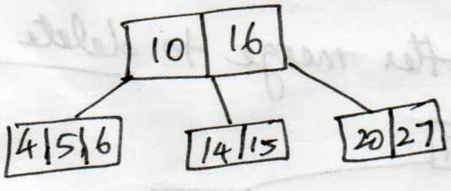
Delete23



→ LS~~T~~ contains more than min key (not directly borrow)

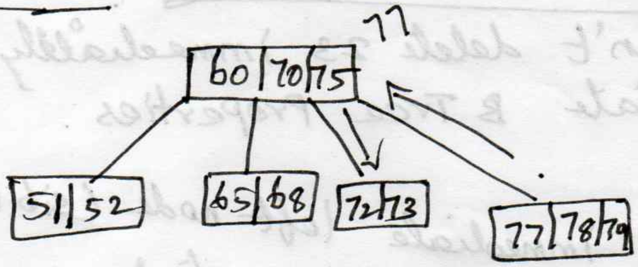
→ LST max value is transferred to parent and ~~[illegible]~~ value of parent ex to delete node.

→ After deletion of 23.



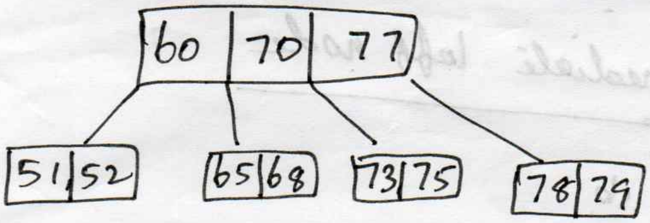
case 2: Borrow immediate Right Sibiling.

Delete 72



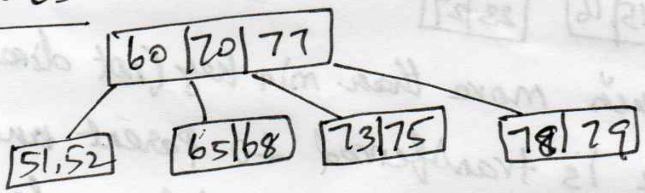
→ right sibiling having more than min key.

→. RS~~T~~ min value tx to parent and value of parent tx to delete node.



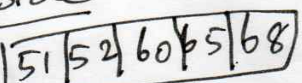
case 3: merging LS RS. in min no. of key.

Delete: 65

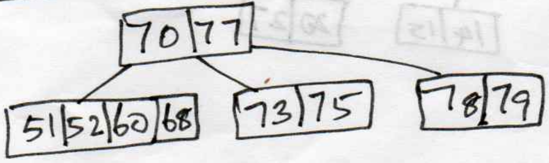


* either LS or RS can be merged with delete node along with parent node also merged

Left side



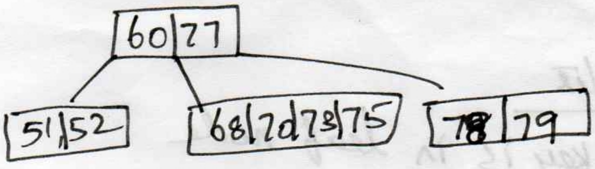
→ After merge to delete 65



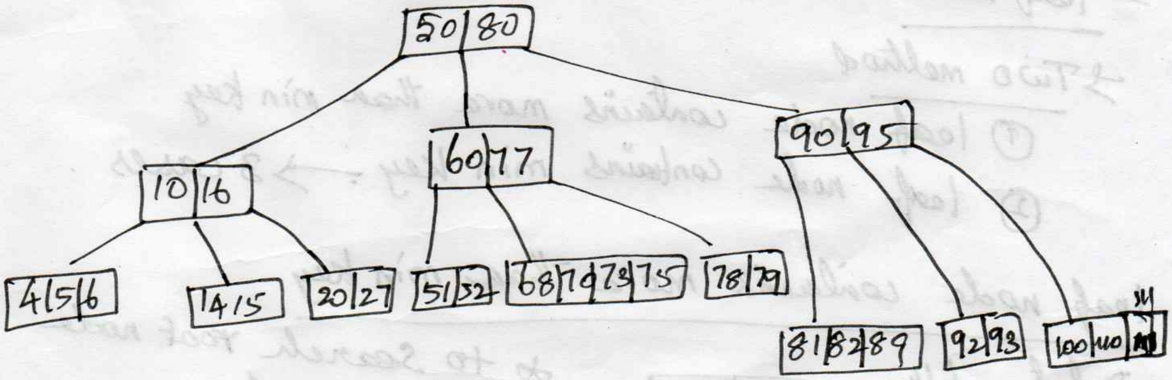
Right side

65 68 70 73 75 → After merge to delete 65

→ also 70 is merged (parent)



Final O/P after First possibilities



second possibilites : if target key is in internal node :

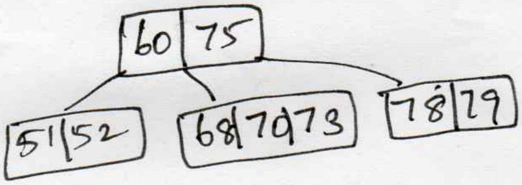
Two case
1. inorder predecessor
2. inorder Successor .

Delete 77 (inorder successor → min key .

Inorder predecessor

→ 75 → that node contains more than minkey.

→ 75 is replace with delete element 77 .



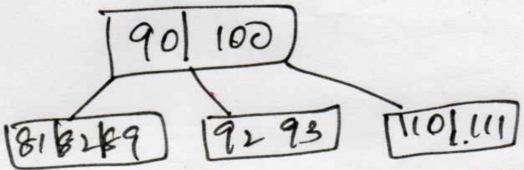
Delete 95

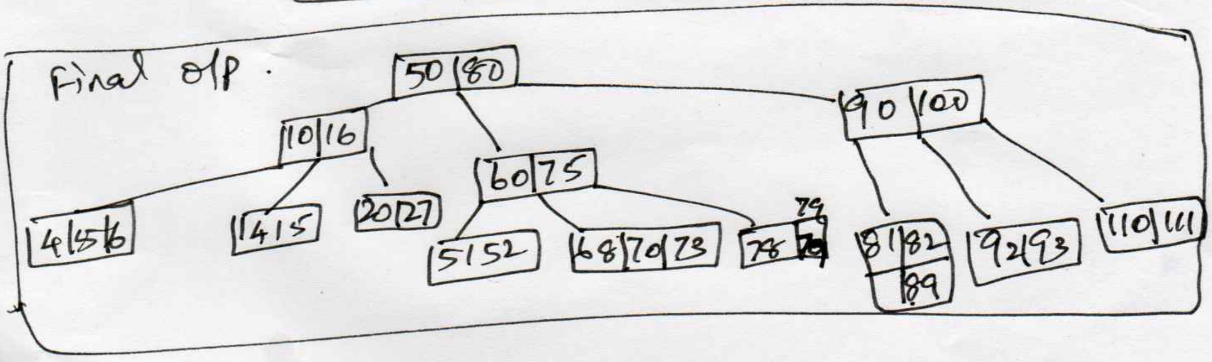
inorder successor contains more than min key .

Inorder predecessor → min key ✗

100

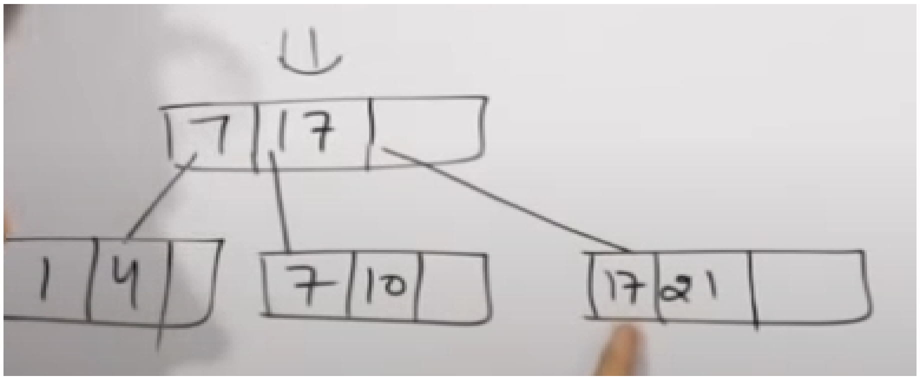
so we replace target key with Inorder successor .





B+ Tree

- B+ Tree is an extension of B Tree which allows efficient insertion, deletion and search operations.
- In **B Tree**, 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(index value).
- The leaf nodes of a B+ tree are linked together in the form of singly linked lists to make the search queries more efficient.
- B+ Tree 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.
- The internal nodes of B+ tree are often called index nodes. B+ tree of order 3 is shown in the following figure.



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.

B Tree VS B+ Tree

SN	B Tree	B+ Tree
1	Search keys can not be repeatedly stored.	Redundant search keys can be present.
2	Data can be stored in leaf nodes as well as internal nodes	Data can only be stored on the leaf nodes.
3	Searching for some data is a slower process since data can be found on internal nodes as well as on the leaf nodes.	Searching is comparatively faster as data can only be found on the leaf nodes.
4	Deletion of internal nodes are so complicated and time consuming.	Deletion will never be a complexed process since element will always be

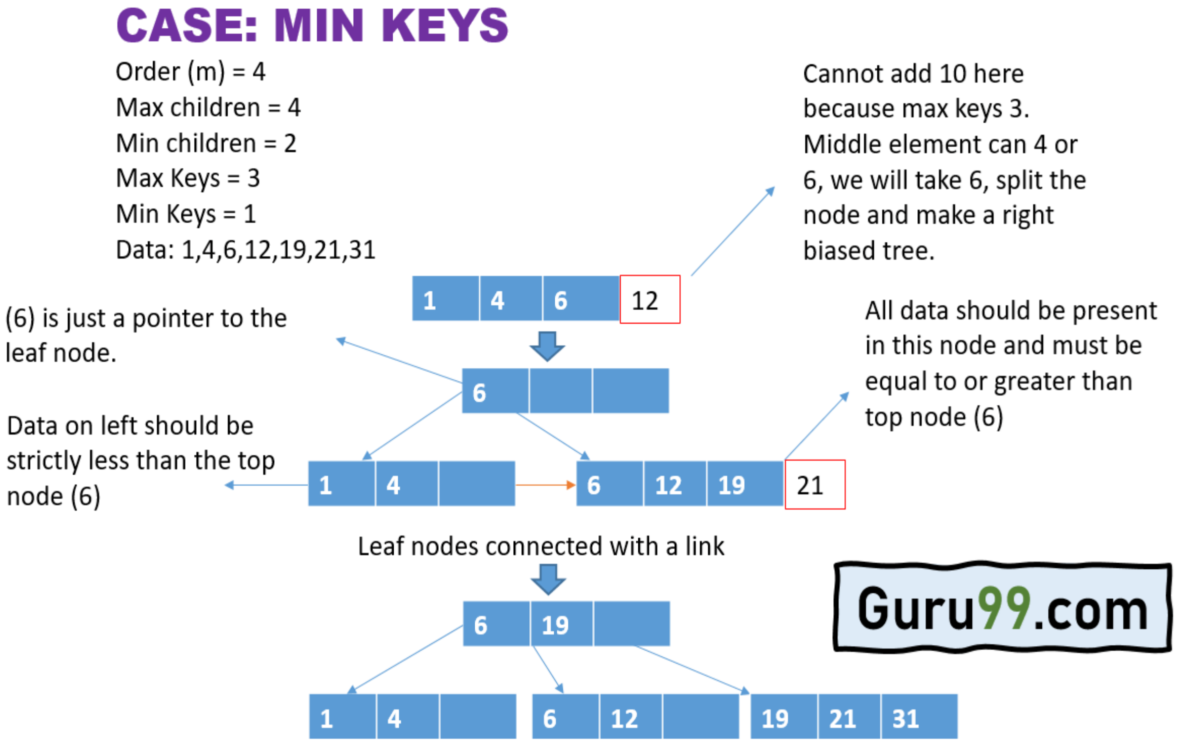
		deleted from the leaf nodes.
5	Leaf nodes can not be linked together.	Leaf nodes are linked together to make the search operations more efficient.

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.

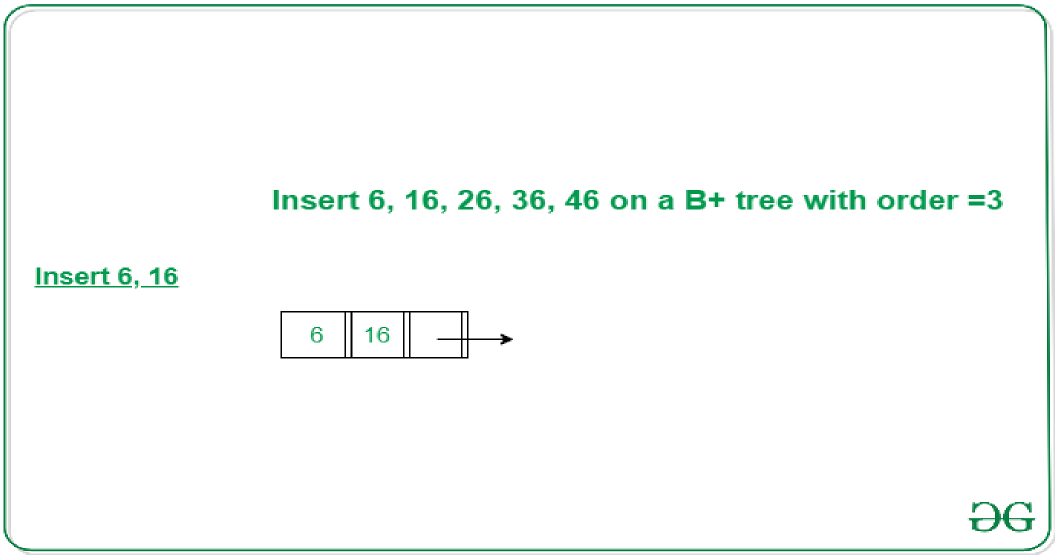

CASE: MIN KEYS
Order (m) = 4
Max children = 4
Min children = 2
Max Keys = 3
Min Keys = 1
Data: 1,4,6,12,19,21,31
Cannot add 10 here because max keys 3. Middle element can 4 or 6, we will take 6, split the node and make a right biased tree.
1
4
6
12
(6) is just a pointer to the leaf node.
6
All data should be present in this node and must be equal to or greater than top node (6)
Data on left should be strictly less than the top node (6)
1
4
6
12
19
21
Leaf nodes connected with a link
Guru99.com
6
19
1
4
6
12
19
21
31

Example 2:

Insert the following key values 6, 16, 26, 36, 46 on a B+ tree with order = 3.

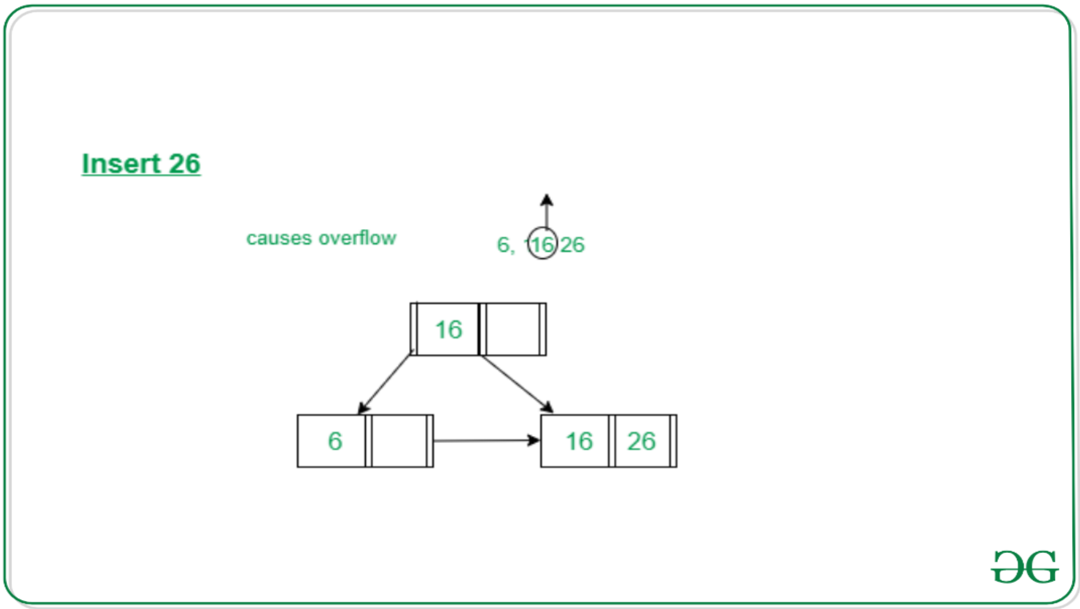
m=3
Max children=3

Min children=2
Max key=2
Min key=1



Insert 6, 16, 26, 36, 46 on a B+ tree with order =3

Insert 6, 16

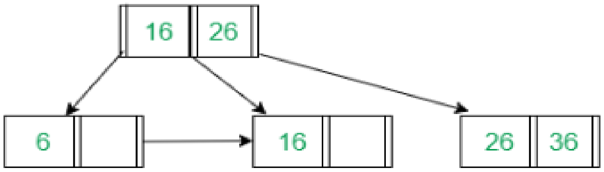


Insert 26

causes overflow

Insert 36

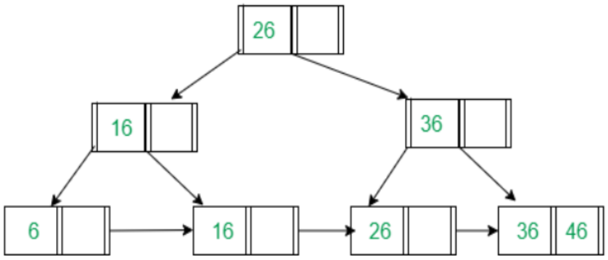
16, 26, 36



Insert 46

26, 36, 46 in leaf node

16, 26, 36 in root node



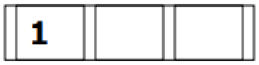
Example 3:

Insert the following key values 1 3 5 7 9 2 4 6 8 10 on a B+ tree with order = 4.

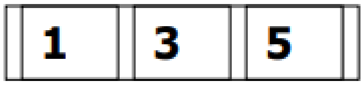
m=4
Max children=4
Min children=2
Max key=3
Min key=1

Insert 1, 3, 5, 7, 9, 2, 4, 6, 8, 10

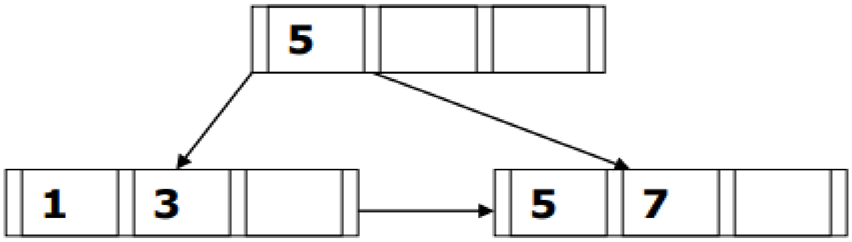
- Insert 1



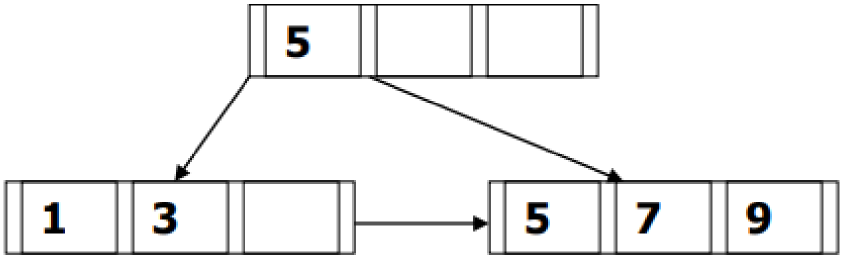
- Insert 3, 5



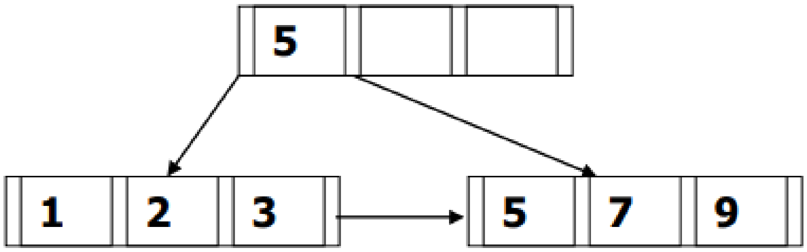
- Insert 7



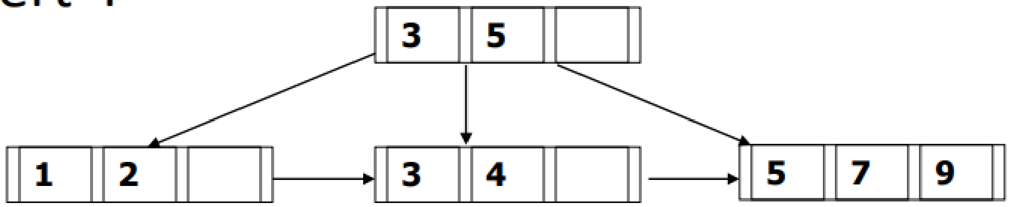
- Insert 9



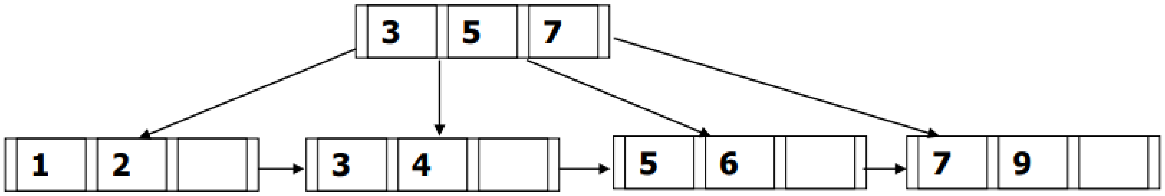
- Insert 2



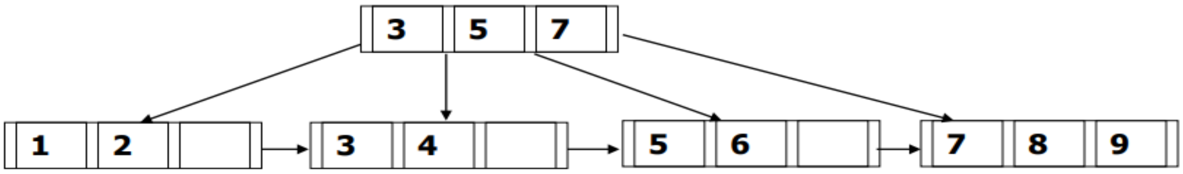
- Insert 4



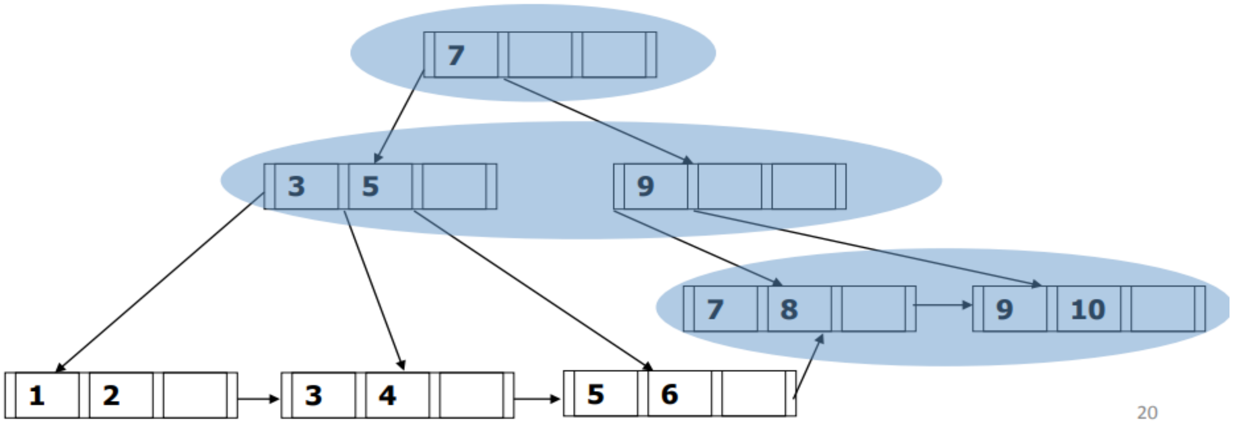
- Insert 6



- Insert 8



- Insert 10



H.W
1. To construct B+ tree order 4 ,   inserting values 1 4 7 10 17 21 31 25 19 20 28 42
2. To construct B+ tree order 4,   inserting values   2 4 7 10 17 21 28
3. To construct B+ tree order 5, ,   inserting values 7 10 1 23 15 17 9 11 39 35 8 40 25

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.

CASE: MIN KEYS

Order (m) = 4

Max children = 4

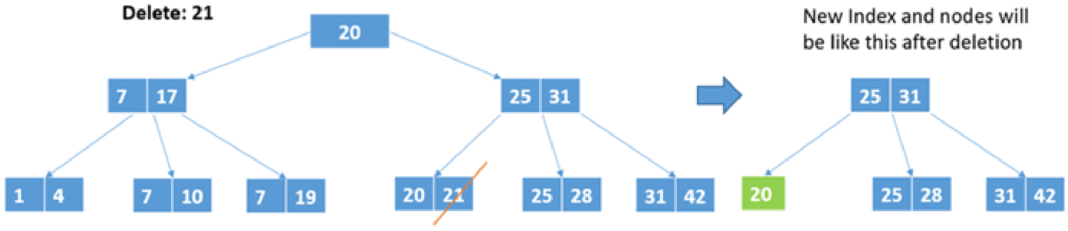
Min children = 2

Max Keys = 3

Min Keys = 1

Data: 21,31,20,10,7,25,42

Delete: 21



New Index and nodes will be like this after deletion

To delete 21, we have to check at both the Index and the Leaf node level for the data 21 and delete from both places.

21 is present only at leaf level so we can simply delete it from here.

CASE: MIN KEYS

Order (m) = 4

Max children = 4

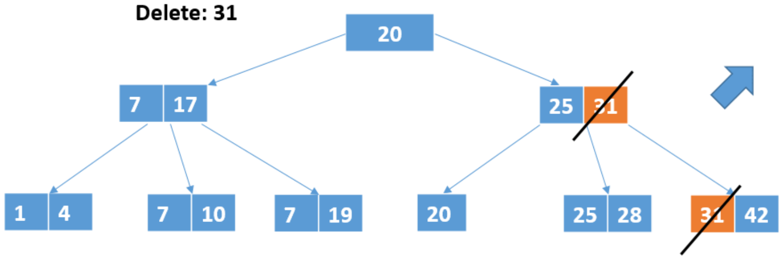
Min children = 2

Max Keys = 3

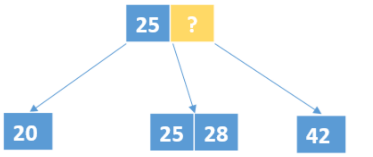
Min Keys = 1

Data: 21,31,20,10,7,25,42

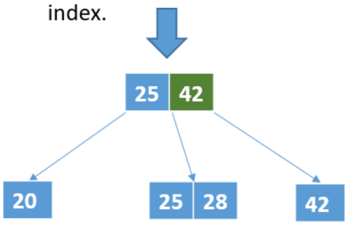
Delete: 31



New Index and nodes will be like this after deletion



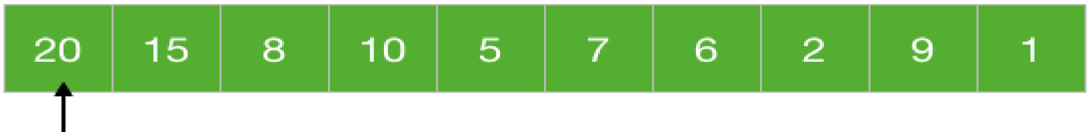
For the empty Index, we will look at the right child and take the minimum value and place it in the index.



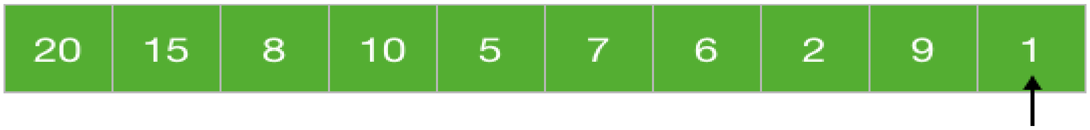
To delete 31, we have to check at both the Index and the Leaf node level for the data 31 and delete from both places.

Priority queues

- Priority queue is a special kind of queue data structure which will have precedence over jobs.
- Priority Queue Data Structure is a regular Queue Data Structure with additional properties
  - Each element has a priority associated with it
  - An element with high priority is served before an element with low priority
  - If two elements have the same priority, they are served according to the order in which they are enqueue
- Thus, a max-priority queue returns the element with maximum key first whereas, a min-priority queue returns the element with the smallest key first.



Maximum key is returned first in the max-queue



Minimum key is returned first in the min-queue

- Priority queues are used in many algorithms like Huffman Codes, Prim's algorithm, etc. It is also used in scheduling processes for a computer, etc.

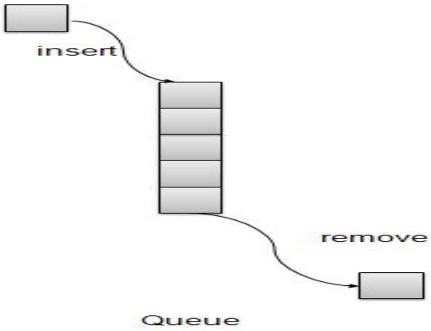
Basic Operations

1. **Insertion operation**:

   It is similar to enqueue, where we insert an element in the priority queue.

2. **DeleteMin operation:**

   It is equivalent of dequeue, where we delete at minimum element from priority queue.



Implementations:

- They are several ways for implementing priority queue.

1. Linked list- simple linked list implementation of priority queue to perform insertion at front and delete the minimum element.
2. Binary search tree - to perform insertion at front and delete the minimum element.
3. Binary heap – efficient way of implementing priority queue.

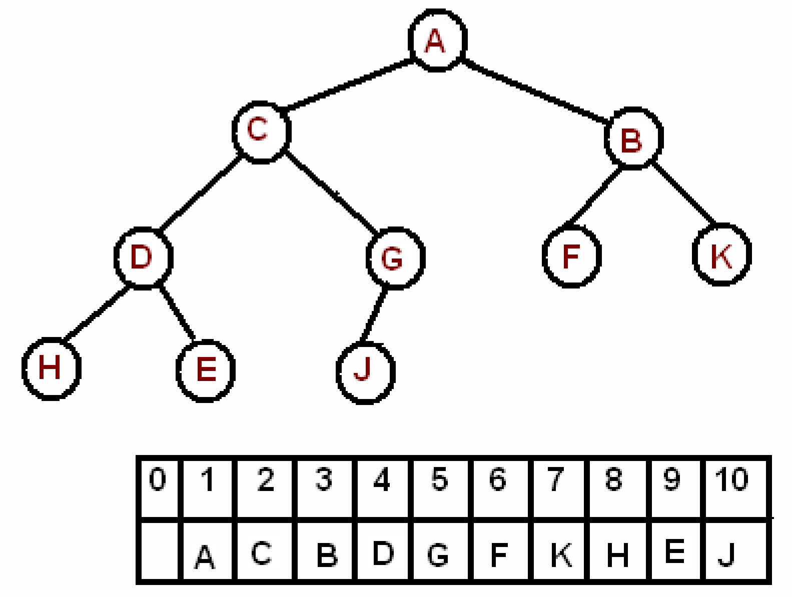
Binary heap:

- Binary heap is merely referred as heaps. A heap is a tree-based data structure in which all the nodes of the tree are in a specific order.

- Heap have two properties namely,
  - Structure property
  - Heap order property

Structure property

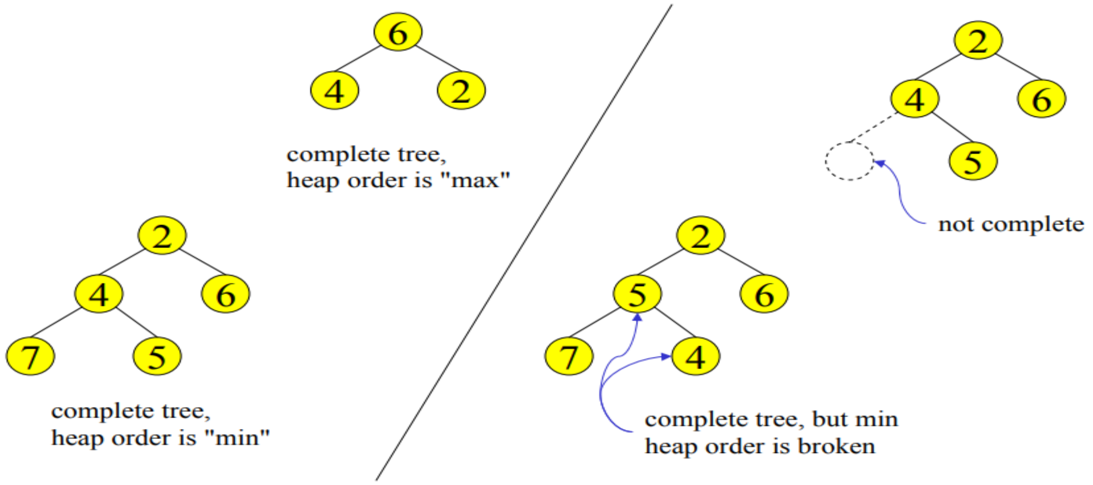
- A binary heap is a binary tree (NOT a BST) that is the tree is completely filled except possibly the bottom level, which is filled from left to right. such a tree is known as a complete binary tree.
- Example of a complete binary tree.



0	1	2	3	4	5	6	7	8	9	10
	A	C	B	D	G	F	K	H	E	J

- For any element in array position i, the left child is in position 2i, the right child is in position 2i+1 and the parent is in position i/2.

Examples



complete tree,
heap order is "max"

not complete

complete tree,
heap order is "min"

complete tree, but min
heap order is broken

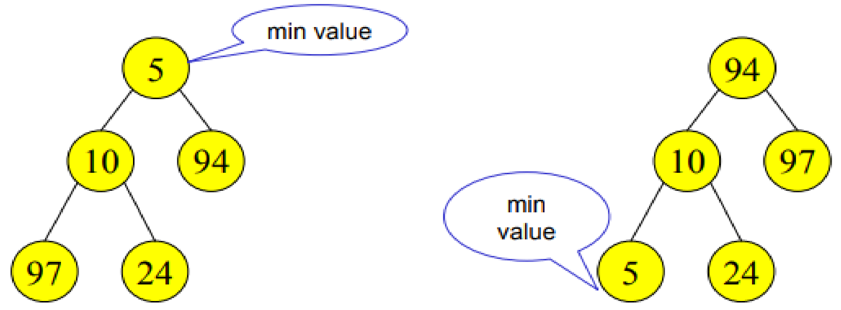
Binary Heaps 9

Heap order property

- In a heap, for every node X, the key in the parent of X is smaller than or equal to the key in X. For example, a complete binary tree that has the heap order property.

Binary Heap vs Binary Search Tree

Binary Heap | Binary Search Tree



Parent is less than both left and right children

Parent is greater than left child, less than right child

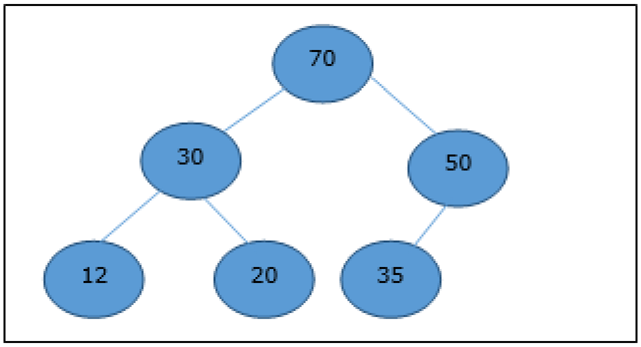
Binary Heaps 7

A binary heap can be classified further as either a max-heap or a min-heap based on the ordering property.

1. Max-Heap

- In this heap, the key value of a node is greater than or equal to the key value of the highest child.

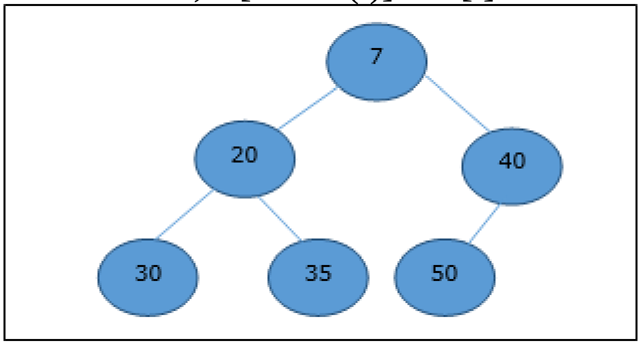
Hence, ***H[Parent(i)] ≥ H[i]***



2. Min Heap

n mean-heap, the key value of a node is lesser than or equal to the key value of the lowest child.

Hence, H[Parent(i)] ≤ H[i]

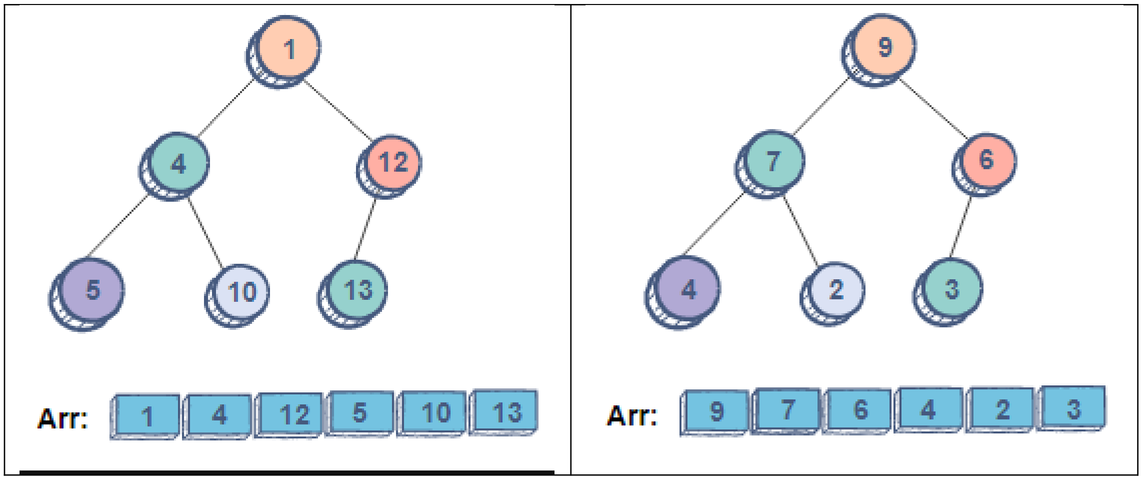


Heapify

- Heapify is the process of creating a heap data structure from a binary tree. It is used to create a Min-Heap or a Max-Heap.

Difference

Min-Heap	Max-Heap
The root node has the **minimum** value.	The root node has the **maximum** value.
The value of each node is equal to or greater than the value of its parent node.	The value of each node is equal to or less than the value of its parent node.
A complete binary tree.	A complete binary tree.



Arr: 1 4 12 5 10 13

Arr: 9 7 6 4 2 3

Operations on Max Heap

The following operations are performed on a Max heap data structure...

1. **Insertion**
2. **Deletion**

Insertion Operation in Max Heap

Insertion Operation in max heap is performed as follows...

Step 1 - Insert the **newNode** as **last leaf** from left to right.

Step 2 - Compare **newNode value** with its **Parent node**.

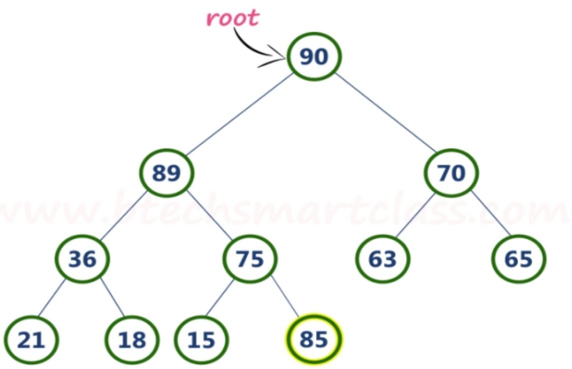
Step 3 - If **newNode value is greater** than its parent, then **swap** both of them.

Step 4 - Repeat step 2 and step 3 until newNode value is less than its parent node (or) newNode reaches to root.

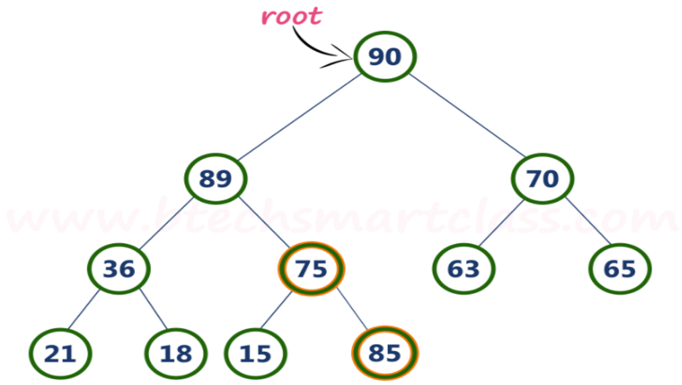
Example

Consider the above max heap. **Insert a new node with value 85.**

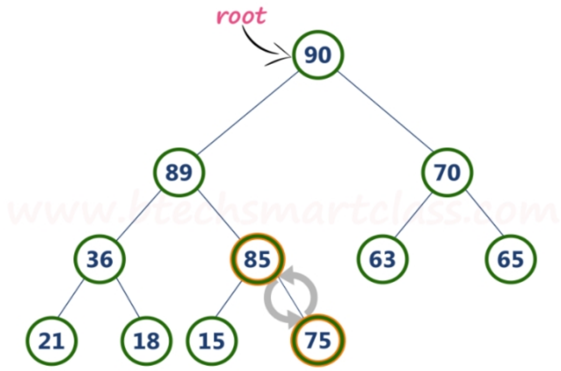
Step 1 - Insert the **newNode** with value 85 as **last leaf** from left to right. That means newNode is added as a right child of node with value 75. After adding max heap is as follows...



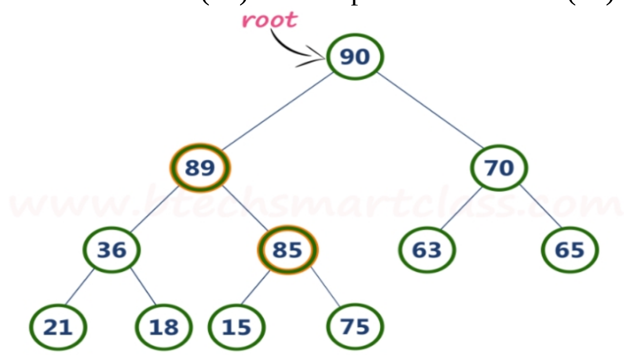
Step 2 - Compare newNode value (85) with its Parent node value (75). That means 85 > 75



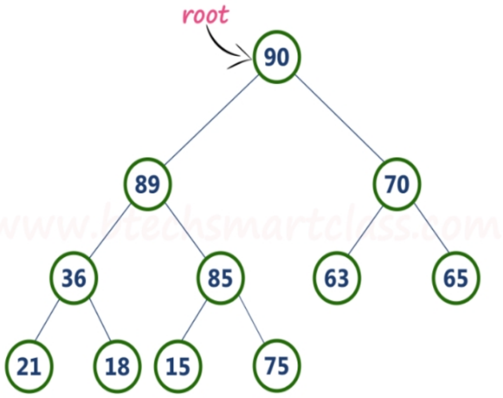
Step 3 - Here newNode value (85) is greater than its parent value (75), then swap both of them. After swapping, max heap is as follows..



step 4 - Now, again compare newNode value (85) with its parent node value (89).



Here, newNode value (85) is smaller than its parent node value (89). So, we stop insertion process. Finally, max heap after insertion of a new node with value 85 is as follows...



Deletion Operation in Max Heap

In a max heap, deleting the last node is very simple as it does not disturb max heap properties.

Deleting root node from a max heap is little difficult as it disturbs the max heap properties. We use the following steps to delete the root node from a max heap...

Step 1 - Swap the **root** node with **last** node in max heap

Step 2 - Delete last node.

Step 3 - Now, compare **root value** with its **left child value**.

Step 4 - If **root value is smaller** than its left child, then compare **left child** with its **right sibling**. Else goto **Step 6**

Step 5 - If **left child value is larger** than its **right sibling**, then **swap root** with **left child** otherwise **swap root** with its **right child**.

Step 6 - If **root value is larger** than its left child, then compare **root value** with its **right child** value.

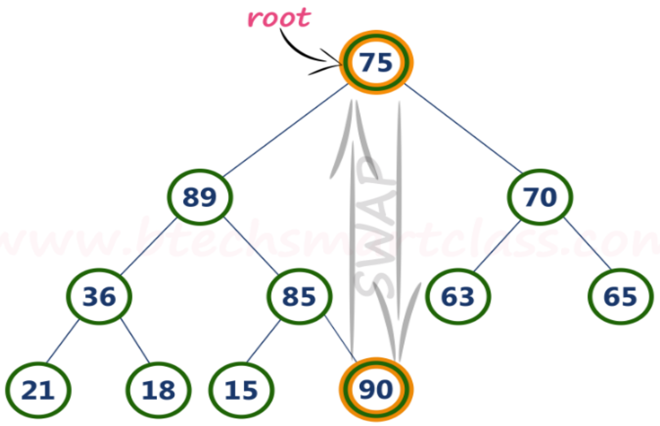
Step 7 - If **root value is smaller** than its **right child**, then **swap root** with **right child** otherwise **stop the process**.

Step 8 - Repeat the same until root node fixes at its exact position.

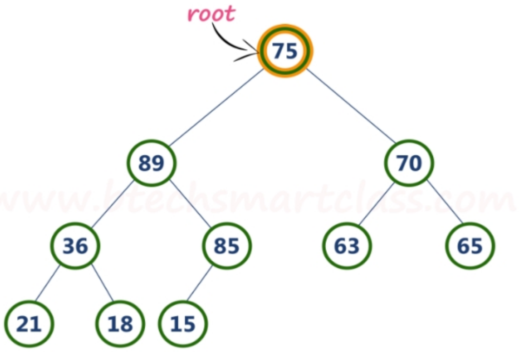
Example

Consider the above max heap. **Delete root node (90) from the max heap.**

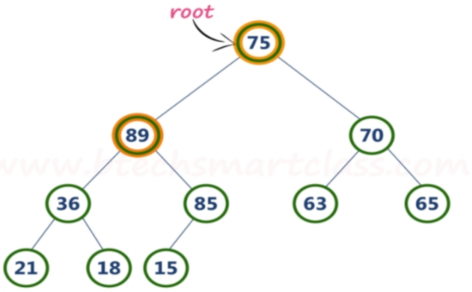
- **Step 1 - Swap** the **root node (90)** with **last node 75** in max heap. After swapping max heap is as follows...



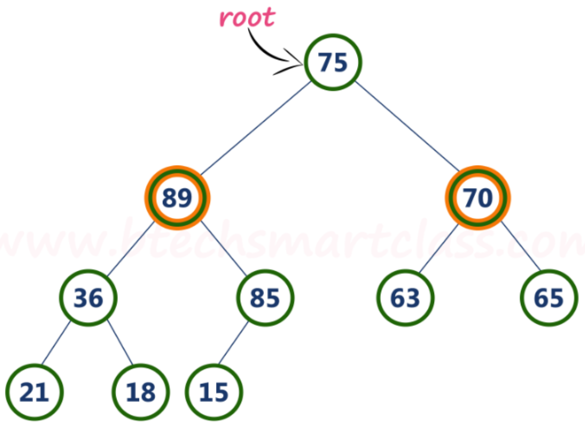
- **Step 2 - Delete** last node. Here the last node is 90. After deleting node with value 90 from heap, max heap is as follows...



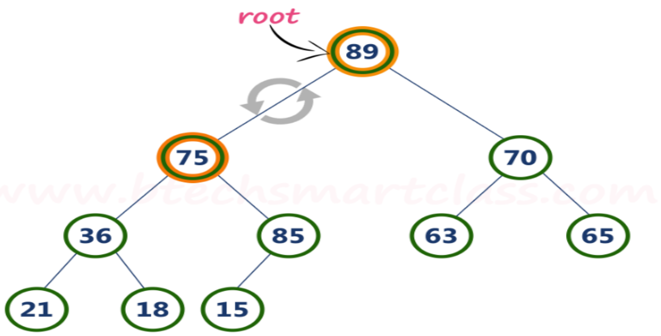
Step 3 - Compare root node (75) with its left child (89).



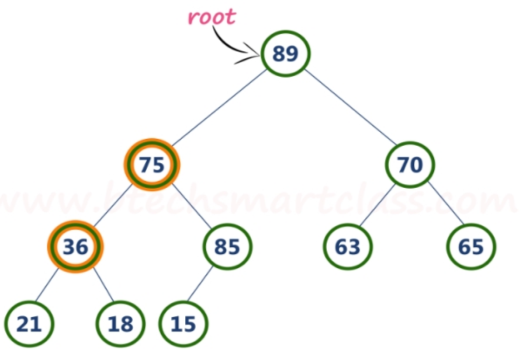
Here, root value (75) is smaller than its left child value (89). So, compare left child (89) with its right sibling (70).



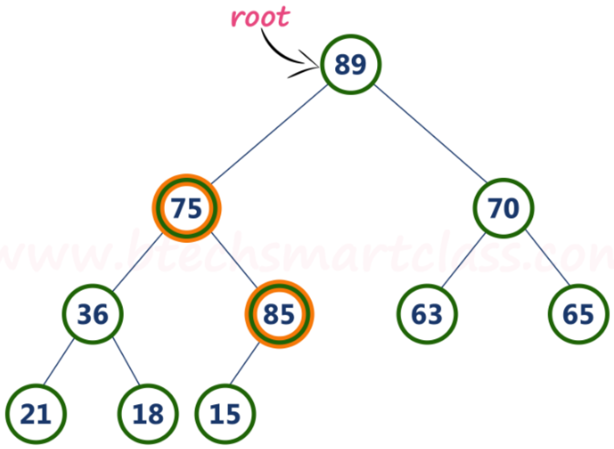
Step 4 - Here, left child value (89) is larger than its right sibling (70), So, swap root (75) with left child (89).



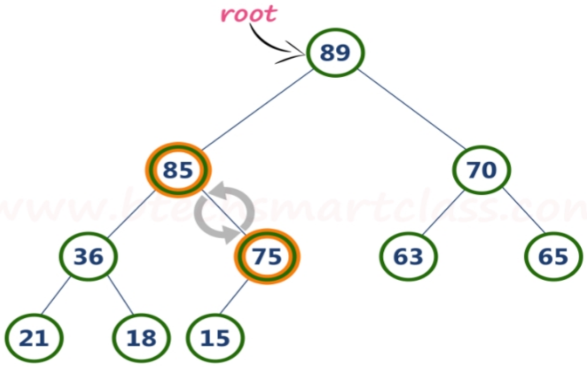
Step 5 - Now, again compare 75 with its left child (36).



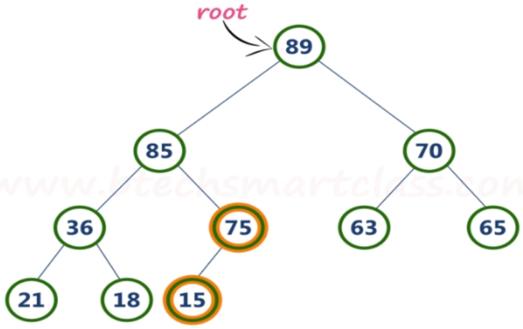
Here, node with value 75 is larger than its left child. So, we compare node 75 with its right child 85.



Step 6 - Here, node with value 75 is smaller than its right child (85). So, we swap both of them. After swapping max heap is as follows...

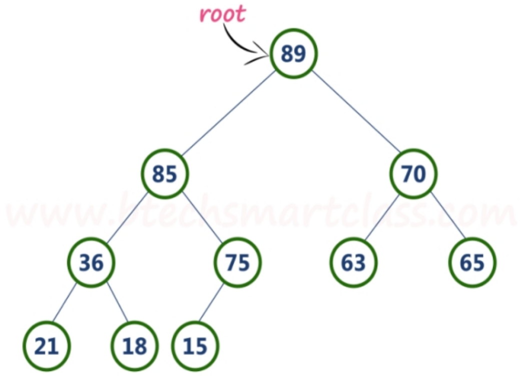


Step 7 - Now, compare node with value 75 with its left child (15).

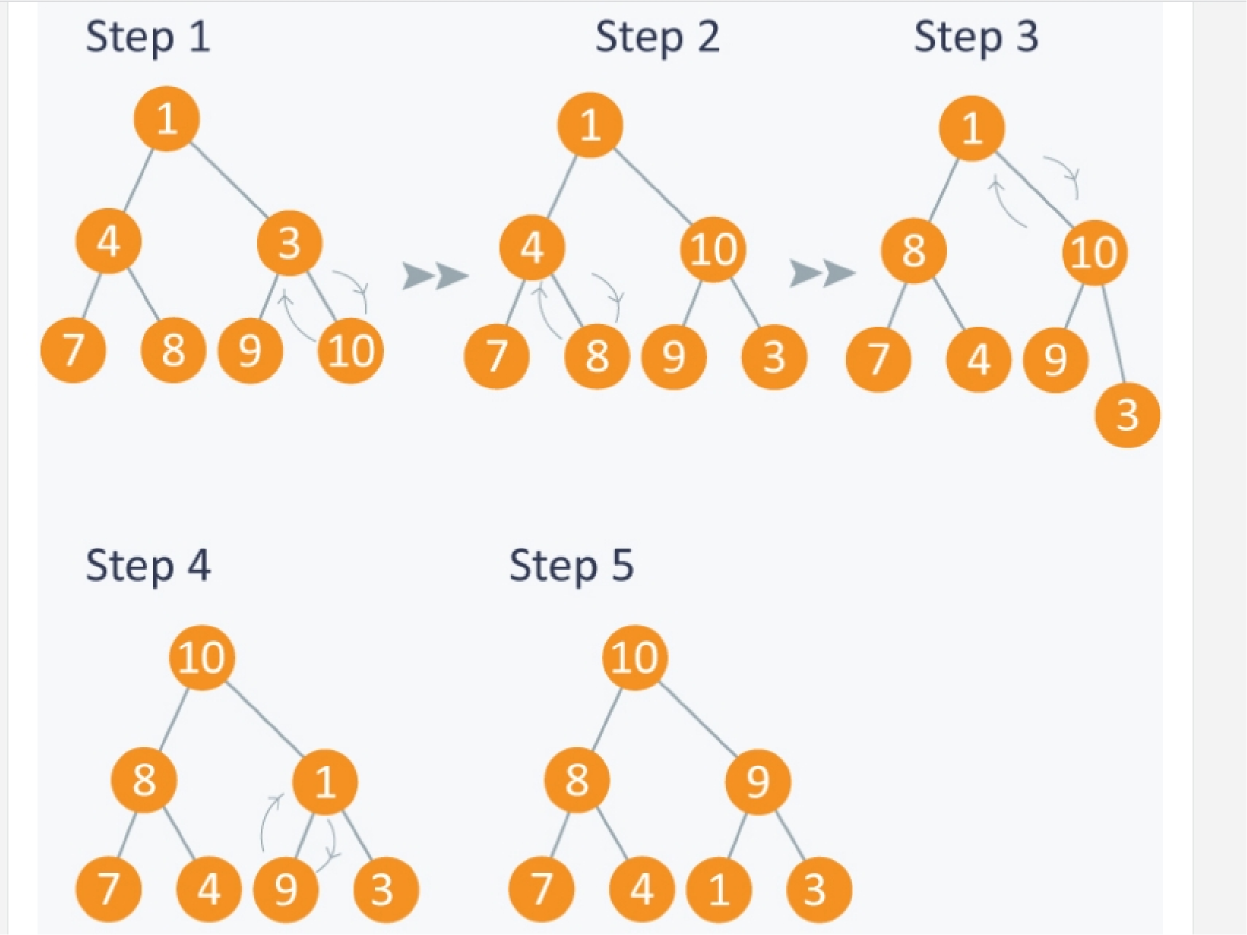


Here, node with value 75 is larger than its left child (15) and it does not have right child. So we stop the process.

Finally, max heap after deleting root node (90) is as follows...

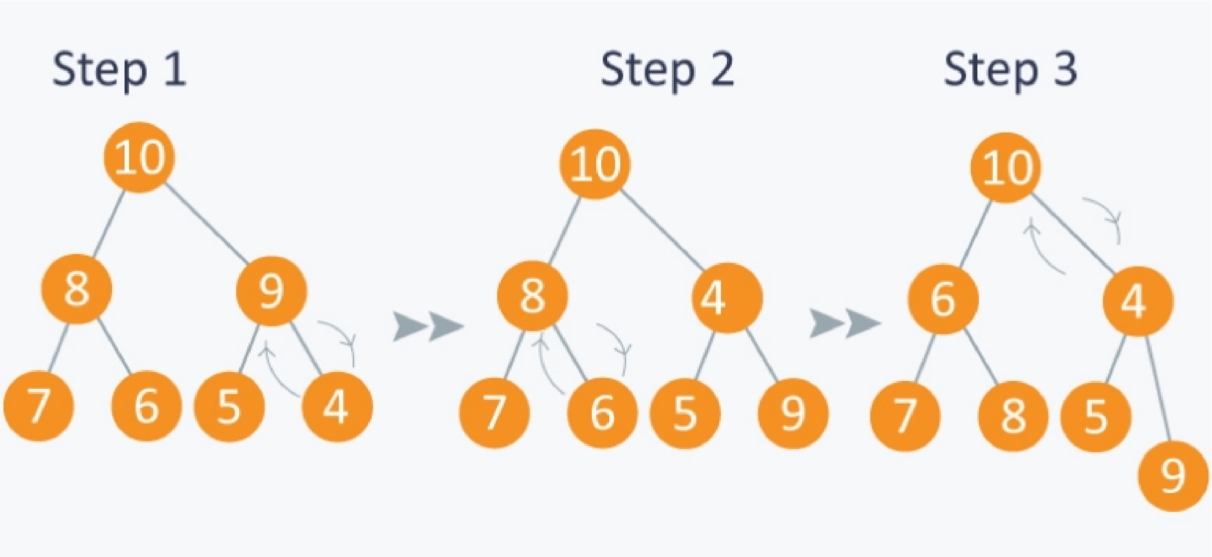


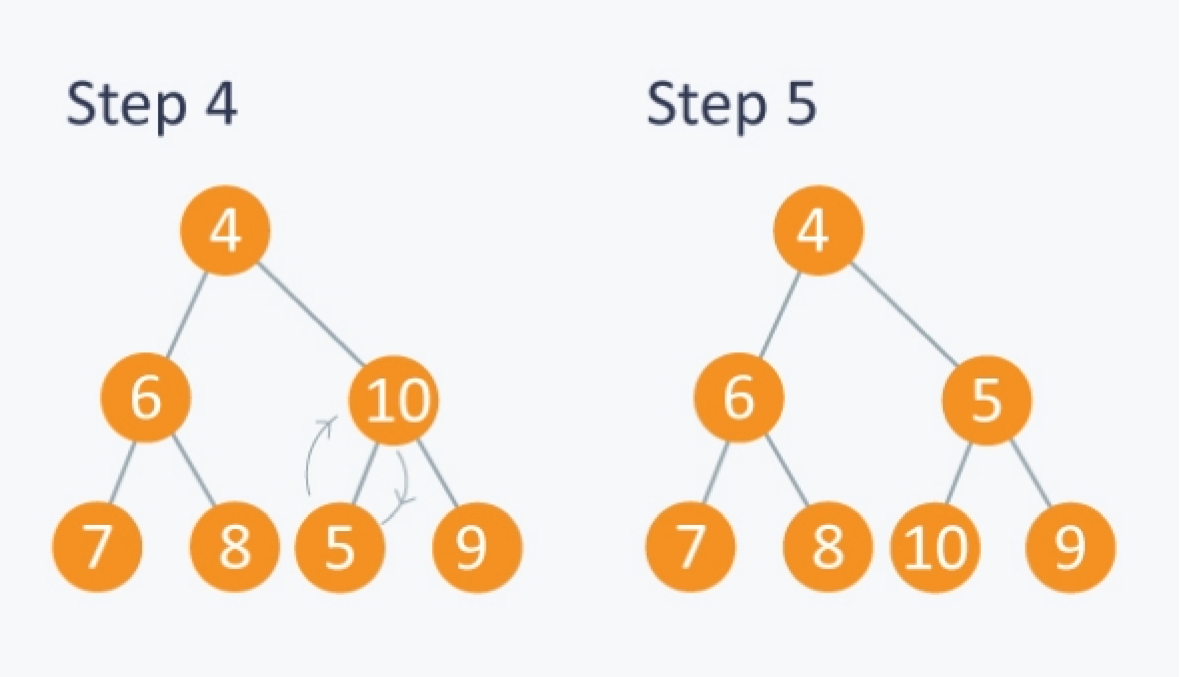
example 2: Consider elements in array {1 4 3 7 8 9 10 }



Min heap

Consider elements in array {10, 8, 9, 7, 6, 5, 4} .





Applications of priorty queue:

- A priority queue is typically implemented using Heap data structure.

Applications:

- Dijkstra’s Shortest Path Algorithm using priority queue: When the graph is stored in the form of adjacency list or matrix, priority queue can be used to extract minimum efficiently when implementing Dijkstra’s algorithm.

- Prim’s algorithm: It is used to implement Prim’s Algorithm to store keys of nodes and extract minimum key node at every step.
- Data compression : It is used in Huffman codes which is used to compresses data.
- **Artificial Intelligence** : A* Search Algorithm : The A* search algorithm finds the shortest path between two vertices of a weighted graph, trying out the most promising routes first. The priority queue (also known as the fringe) is used to keep track of unexplored routes, the one for which a lower bound on the total path length is smallest is given highest priority.
- Heap Sort : Heap sort is typically implemented using Heap which is an implementation of Priority Queue.

Binomial heap or queue

- Binomial heap is a heap which is similar to a binary heap but also support quick merging of two heaps.
- This is achieved by using a special data structure we hae is called ***binomial queue*****.**

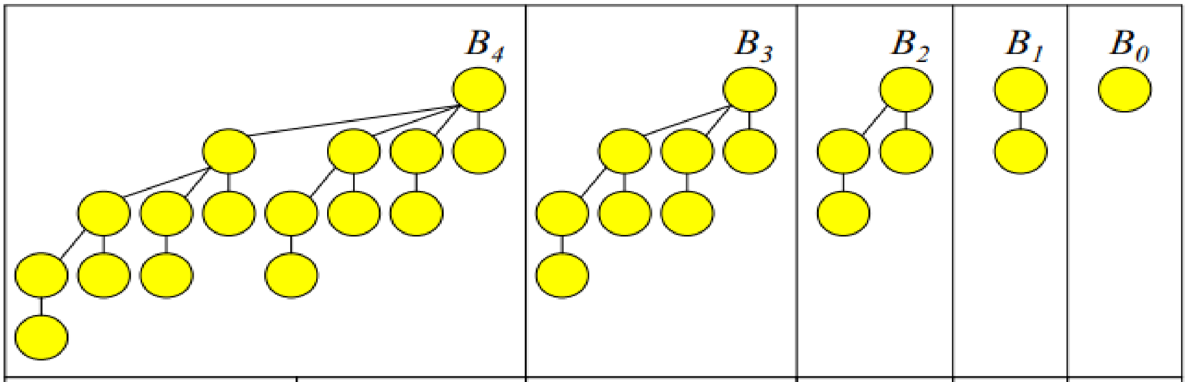
***Binomial queue*:**

- Binomial queues are a collection of heap-ordered trees. Each of the heap-ordered trees is called a *binomial tree.*
  - Not just one tree, but a collection of trees
  - each tree has a defined structure and capacity
  - each tree has the familiar heap-order property
- Binomial queue called as binomial heap, generally satisfy min heap property.
- Assume binomial tree represented as B_K .That means, BT of order k contains 2^k nodes.
  - suppose order k -> 2^k

0 ->2^0 =1 element
1 ->2^1 =2 element
2 ->2^2 =4 element
3 ->2^3 =8 element

- The number of nodes in a tree of depth d is exactly 2^d.

Binomial Queue with 5 Trees



depth	4	3	2	1	0
number of elements	$2^4 = 16$	$2^3 = 8$	$2^2 = 4$	$2^1 = 2$	$2^0 = 1$

Structure of binomial heap

- a binomial heap is implemented as a set of binomial tree that satisfy the binomial heap properties,
    - each binomial tree in a heap obeys the min heap property(p<=c)
    - there can only be either one or zero binomial trees for each order, including zero order.

Operations in binomial queue:

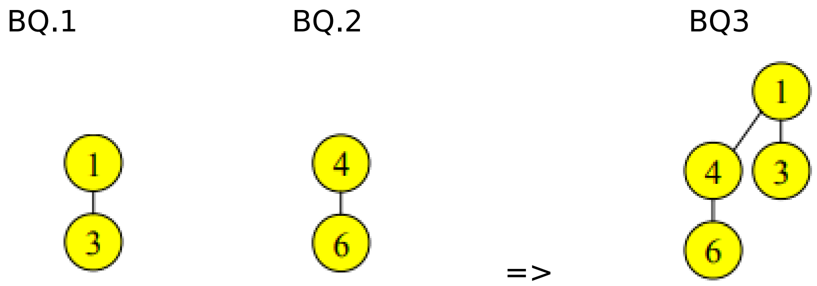
1. merge
2. insertion
3. find min
4. delete

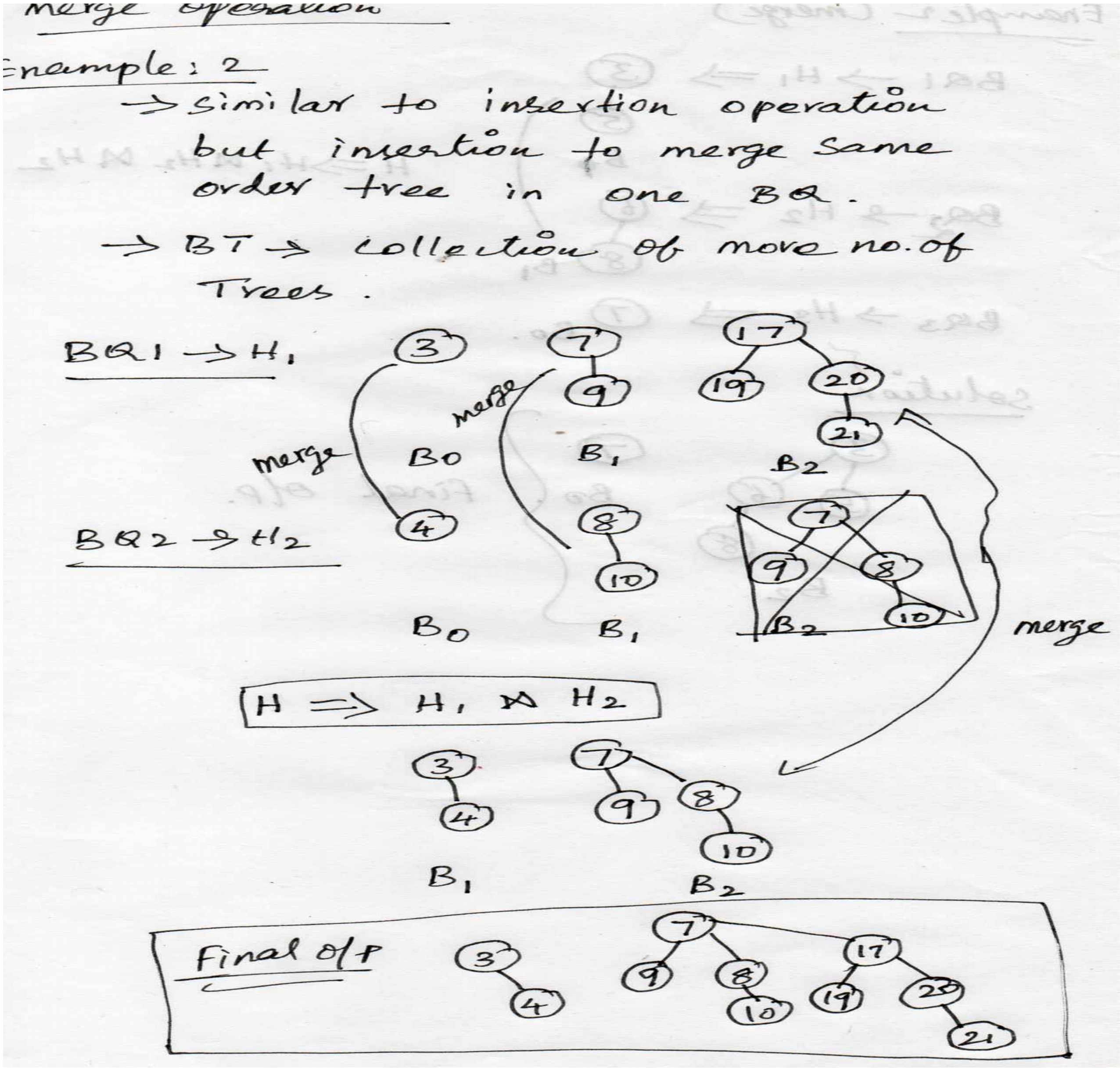
Merge operation:

- Simple operation is the merging of two binomial tree of same order within a binomial heap.
    - $B_k = B_{k-1} + B_{k-1}$

- As their root node is the smallest element within the tree, by comparing the two keys, smaller of them is the min key and becomes the new root node.
- Then the other tree becomes a subtree of the combined tree.

Example





merge operation

Example: 2

→ similar to insertion operation but insertion to merge same order tree in one BQ.

→ BT → collection of more no. of Trees.

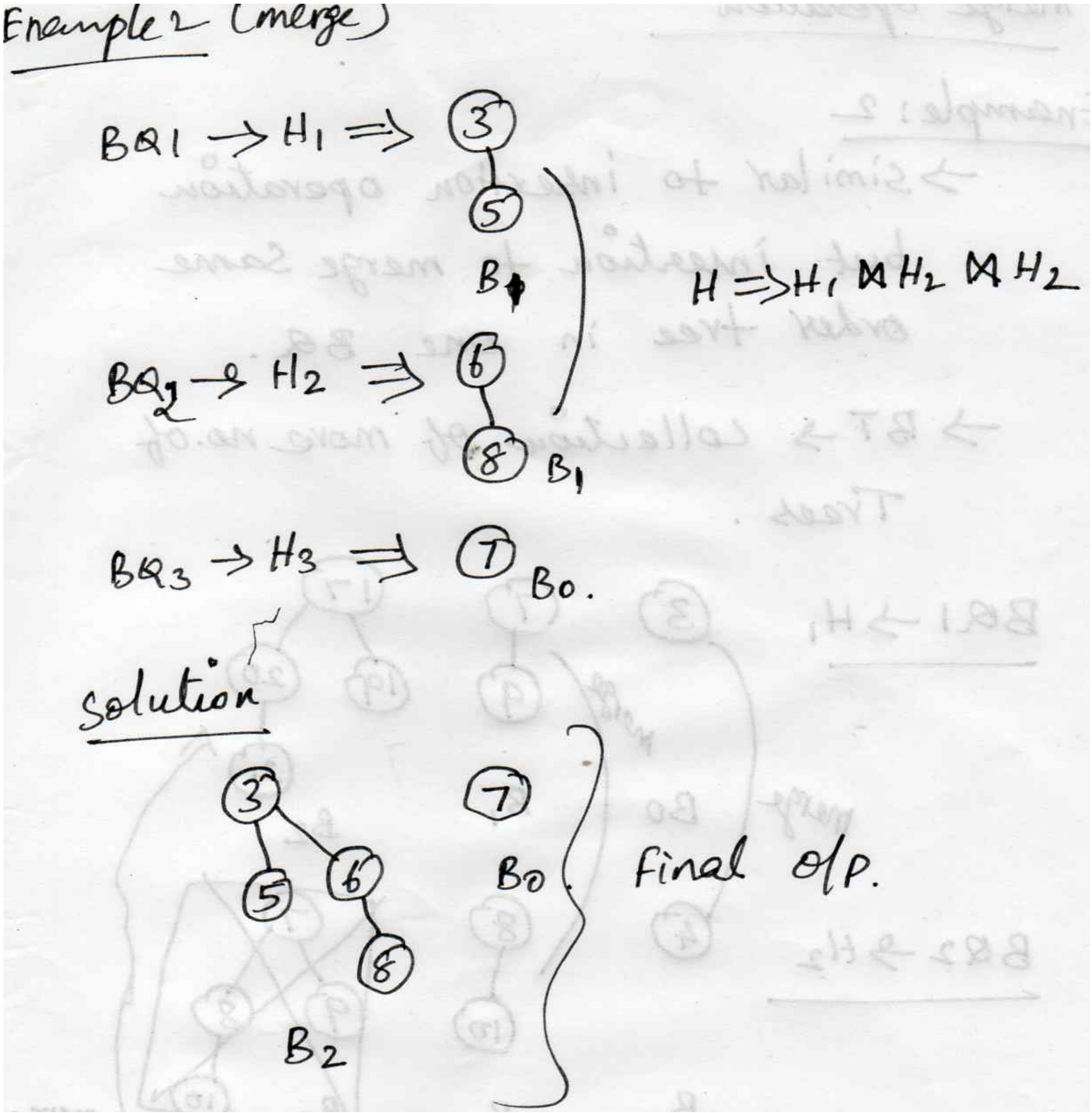
BQ1 → H1
3
7
9
17
19
20
21
merge
merge
B0
B1
B2
BQ2 → H2
4
8
10
7
9
8
10
B2
B0
B1
merge

$\boxed{H \Rightarrow H_1 \bowtie H_2}$

3
4
7
9
8
10
B1
B2

Final o/p

3
4
7
9
8
10
17
19
20
21

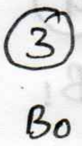


Insertion operation:

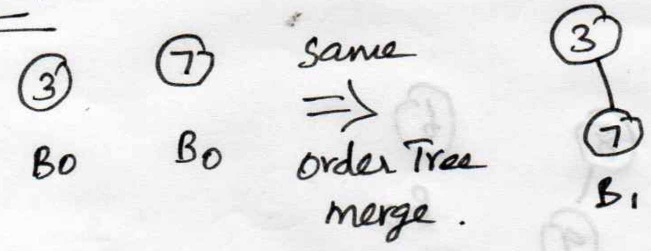
- Insertion is just a special case of merging, since we merely create a one-node tree and perform a merge.

Insertion 3, 7, 4, 11, 8, 9, 6, 10, 12

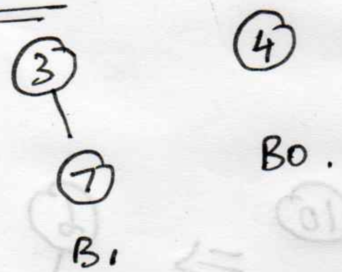
Insert 3



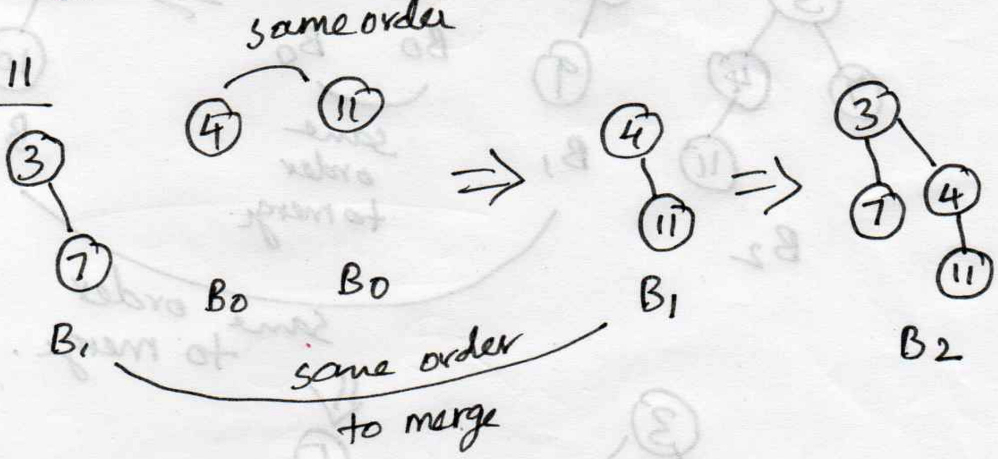
insert 7



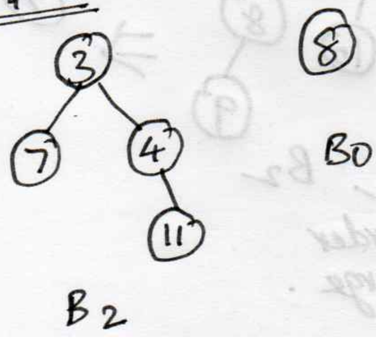
Insert 4



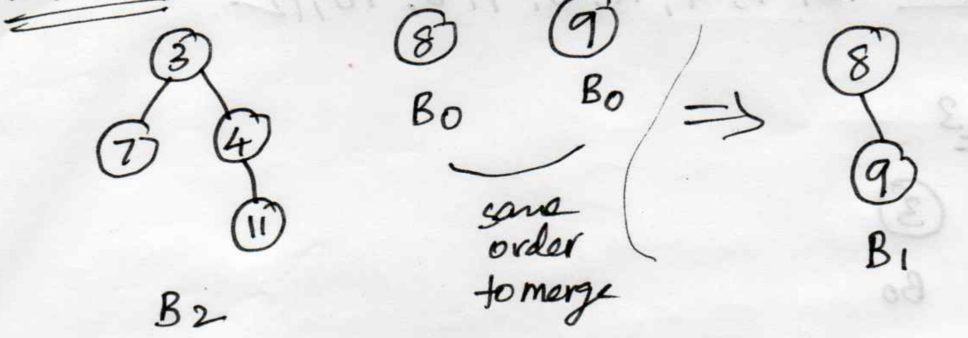
Insert 11



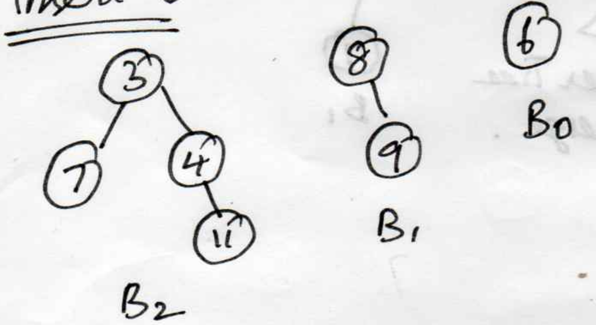
Insert 8



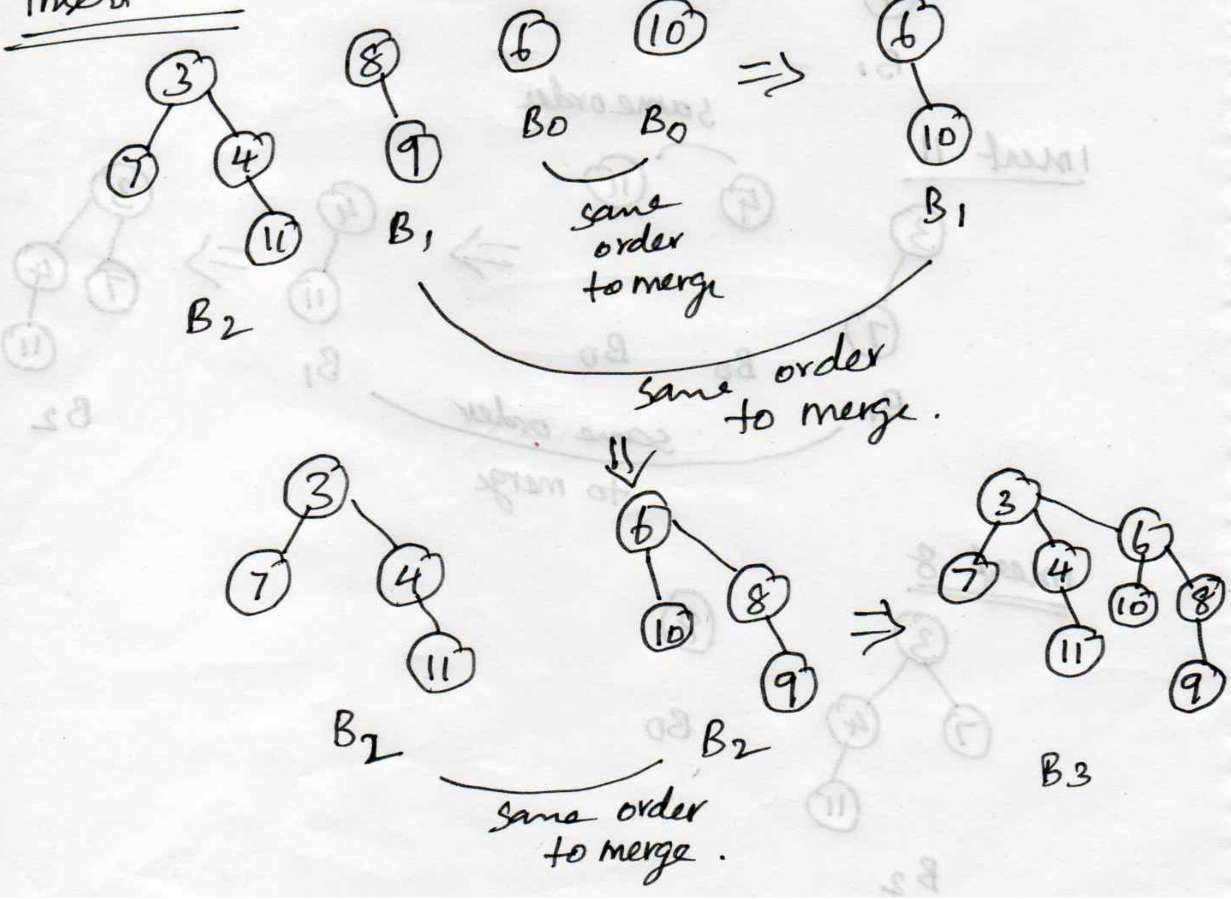
Insert 9

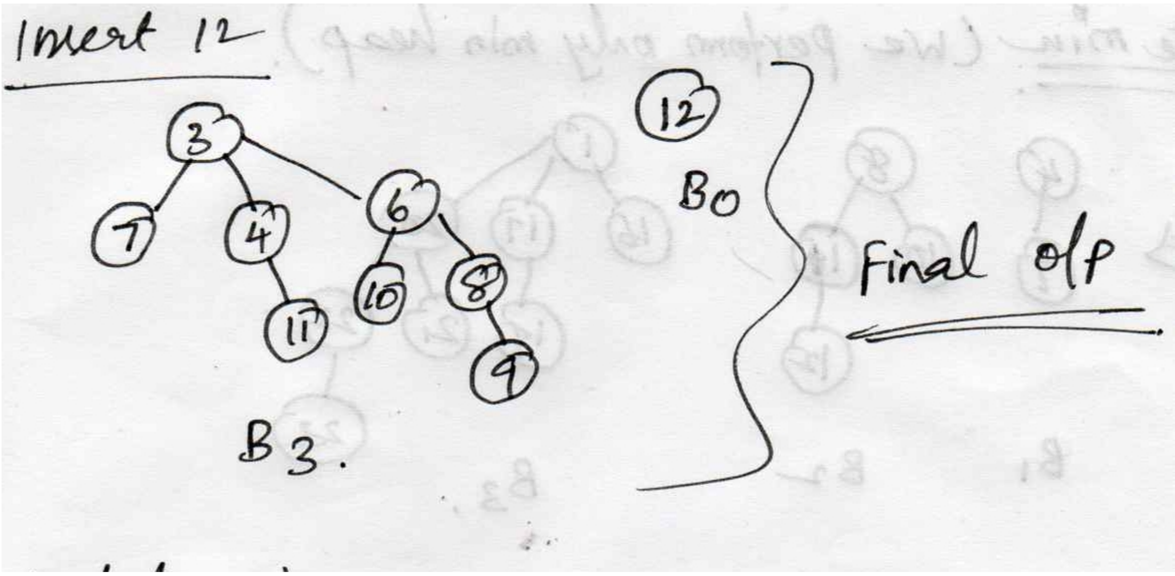


Insert 6



Insert 10





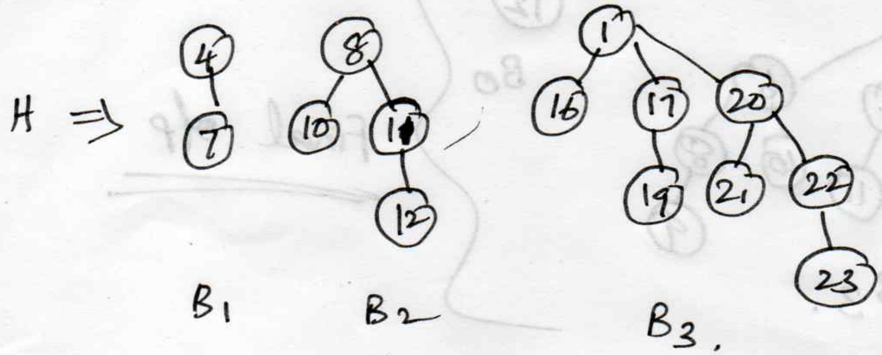
Find Min operation:

- To find min element of the heap, only compare the root of the binomial heap.

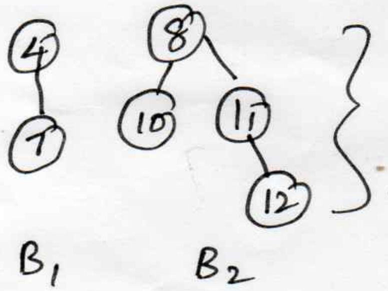
Delete min operation:

- Delete min element from the heap
- First find the element, remove it from its binomial tree and obtain a list of its subtrees
- Then transform this list of subtree into a separate binomial heap by reordering them from smallest to biggest order.

Delete min (we perform only min heap).

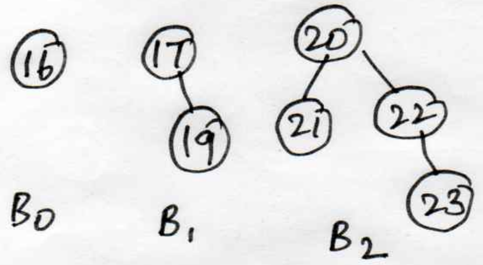


H' ⇒ To check root value, because root is min value.



BQ without deleted key.

H'' ⇒ BQ with deleted key.

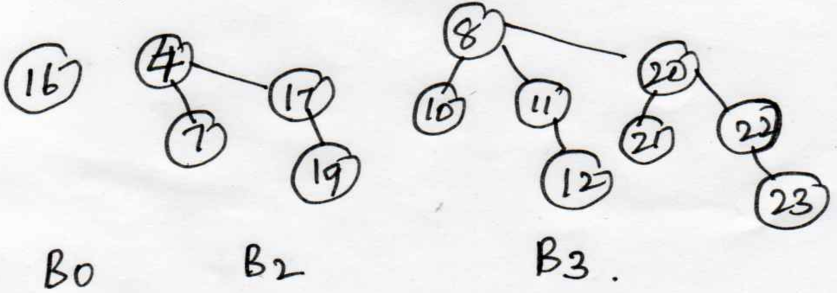


B3 = 8
if delete one element
B3 = 7.

B0	B1	B2
1	2	4.

* B

merge operation H' ⋈ H''


16
4
7
17
19
8
10
11
12
20
21
22
23
B0
B2
B3.