Unit-2:- Trees Part-1

Trees:-

A tree is a very useful data structure and it is a non-linear data structure.

A tree is a finite set of one (or more nodes, such that

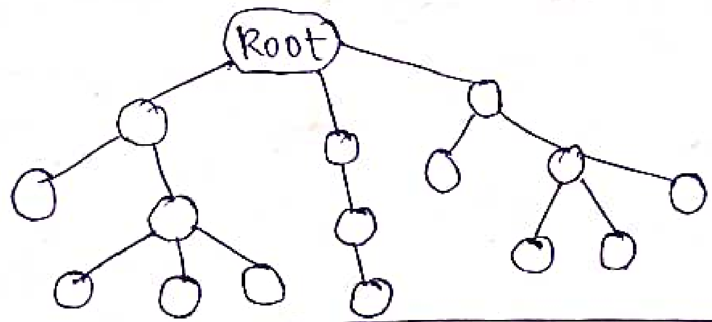
(i) There is a specially designed node called the root.

(ii) the remaining nodes are positioned in to 'n' disjoint sets $(n>0)$

i.e, $T_1, T_2, T_3, \ldots, T_n$.

where, each T_i $(i = 1, 2, \ldots, n)$.

$T_1, T_2, T_3, \ldots, T_n$ are called sub trees of the root.



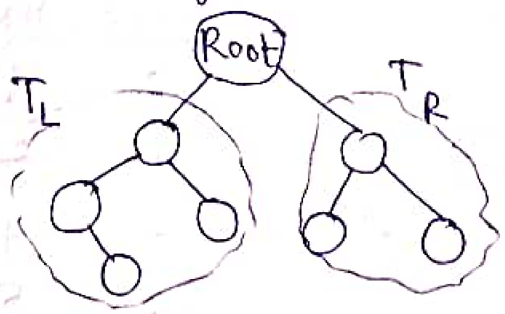
Binary Tree:-

(1) A binary tree is a special form of a tree. A binary tree is more important and frequently used in various applications of computer science.

A binary tree can also be defined as a finite set of nodes, such that Tree (T) is empty. 'T' contain a specially designed node called the root of 'T' and the remaining nodes of 'T' forms two disjoint sets of binary tree.

i.e, T_L and T_R which are called the left sub tree and the right sub tree.

Ex:- Binary Tree



There are 2 major differents b/w a tree and a binary tree.

(i) A tree can be never be empty, but a binary tree may be empty.

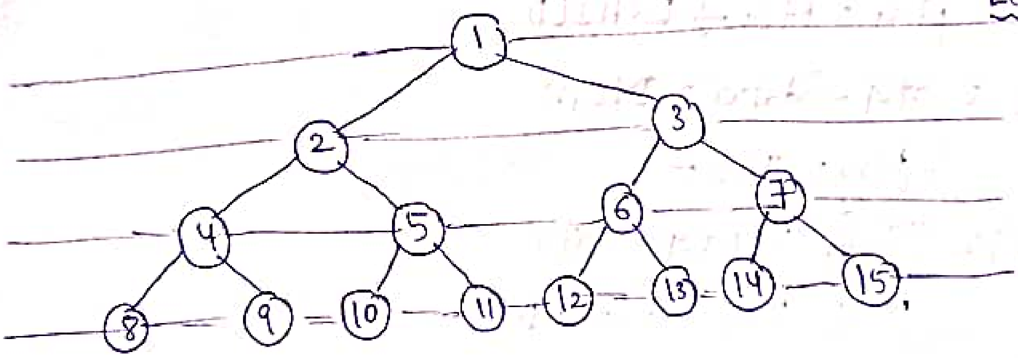
(ii) A binary tree may have at most two childrens. Where a tree can have many number of childrens.

(iii) Two special situations of a binary tree. i.e, Fully Binary tree., and Complete Binary Tree,

---

Fully Binary Tree:-

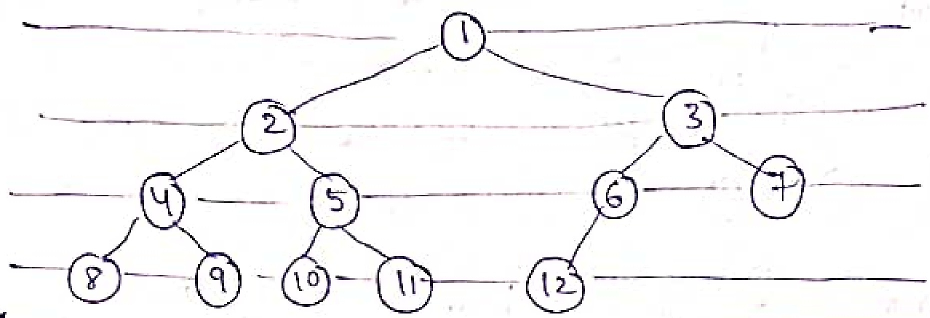
A binary tree, is a fully binary tree, if it contains the maximum possible number of nodes at all levels.



Levels	Nodes
0	1
1	2
2	4
3	8

Complete Binary Tree:-

A binary tree is said to be complete binary tree if all its levels except possibilly the last level have the maximum number of possible nodes and all the nodes in the last level appears as for left as possible.



Levels	nodes
0	1
1	2
2	4
3	5

Searching a Binary Search Tree:-

2) Searching a data in a BST is much faster than searching data in array & linked list. So, many applications which are used to perform searching operation. This data structure is used to store data. Suppose, in a BST(T) ITEM is the item of search and we will assume that the tree is represented using a linked structure.

Algorithm Search_BST

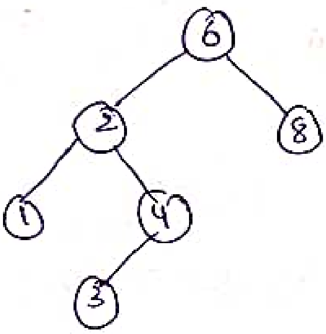
```
1). ptr = ROOT, flag = FALSE
2) while (ptr ≠ NULL) and (flag = FALSE) do
    case : ITEM < ptr -> data
              ptr = ptr -> LCHILD
    case : ptr -> data = ITEM
              flag = TRUE
    case : ITEM > ptr -> data
              ptr = ptr -> RCHILD
    end case
  end while
  if (flag = TRUE) then
        print "ITEM has found at the node", ptr
  else
     print "ITEM doesn't exist search is unsuccessful".
  end if
stop.
```

Here, we start from root (R) then if ITEM is lesser than the value in the root node -R we proceed to its left child.
If ITEM is greater than the value in the root-R we proceed to its right child. The process will be continued till the ITEM is not found (or we reach a dead end.

⑤ Insertion in BST:-

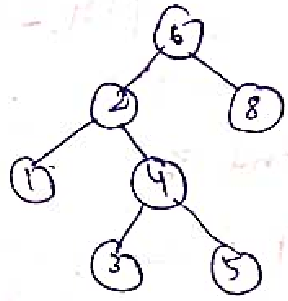
The insertion operation on a BST is conceptually very simple. It is just one step more than the search operation. To insert a node with data that is ITEM into a tree. The tree is requested to be searched starting from the root node. If ITEM is found no insertion. If ITEM is not found then it is to be inserted at the end.

Ex:-



⇒ ITEM = 4
ITEM = Root
no insertion

⇒ ITEM = 5.



Algorithm Insert-BST

ptr = ROOT, flag = FALSE

while (ptr ≠ NULL) and (flag = FALSE) do

case : ITEM < ptr → Data

ptr1 = ptr

ptr = ptr → LCHILD

```
case : ITEM > ptr -> data
    ptr1 = ptr
    ptr = ptr -> RCHILD
    ptr -> data = ITEM
        flag = TRUE
        print ("ITEM already exist")
    ~~end case~~
      exit
   end case
end while
If (ptr = NULL) then
  new = Getnode (NODE)
  new -> data = ITEM
  new -> LCHILD = NULL
  new -> RCHILD = NULL
 if (ptr1 -> data < ITEM) then
      ptr1 -> RCHILD = new
 else
     ptr 1 -> LCHILD = new
 end if
end if
stop.
```

Deleting a node in BST:-

This operation is slightly more complicated than the previous two operations. "T" is a BST and ITEM is the information which has to be deleted from "T". If that existed in the tree.

Suppose N be the node which contains the information ITEM & assume that PARENT of N denotes the parent node of N & SUCC(N) denoted the in order successor of the node N.
(nodes comes after N. Then the deletion of node N depends on the number of children. There are 3 cases of situations.

case-i: N is the leaf node

case-ii: N has exactly one child.

case-iii: N has two childs.

case-i:- N is deleted from the T by simply setting the pointer of N in the parent node.
i.e, PARENT(N) by null value.

case-ii:- N is deleted from the T by simply replacing the pointer of N in parent node.
i.e, PARENT(N) by the pointer of the only child of N.

case-iii:- N is deleted from the T by first deleting success SUCC(N) from T and then replacing the data content in node N by the data content in node SUCC(N). Reset the left child of the parent of SUCC(N) by the right chid of SUCC(N).

Algorithm:-

```
Algorithm Delete-BST
ptr = ROOT, flag = FALSE
while (ptr ≠ NULL) and (flag = FALSE) do
    Case : ITEM < ptr → Data
        ptr1 = ptr
        ptr = ptr → LCHILD
```

```
case : ITEM > ptr -> Data
        ptr1 = ptr
        ptr -> RCHILD
case : ptr -> Data = ITEM
        flag = TRUE
        print " ITEM already exist"
        exit
      end case
  end while
 If (flag = FALSE) then
     print ("ITEM does not exist : No deletion"
       exit
   end if
 if (ptr -> LCHILD = NULL) and (ptr -> RCHILD = NULL) then
       case = 1
    else
       if (ptr -> LCHILD ≠ NULL) and (ptr -> RCHILD ≠ NULL) then
         case = 3
     else
         case = 2
     end if
 end if
    if (case = 1) then
      if (parent -> LCHILD = ptr) then
           parent -> LCHILD = NULL
          else
         parent -> RCHILD = NULL
    end if
Return Node (ptr)
      if (case = 2) then
      if (parent -> LCHILD = ptr) then
```

```
if (ptr→LCHILD=NULL) then
    parent → LCHILD = ptr → RCHILD
  else
    parent → LCHILD = ptr → LCHILD
else
  if (ptr→RCHILD = ptr) then
    if (ptr → RCHILD = NULL) then
        parent → RCHILD = ptr → RCHILD
    else
        parent → RCHILD = ptr → LCHILD
    end if
  end if
end if
Return Node (ptr)
end if
if (case=3) then
    ptr = SUCC (ptr)
    item = ptr → Data
    Delete_BST (item)
        ptr → Data = item
end if
Stop
```

(5) Traversal on Binary search Tree (BST):-

The traversal operation is a frequently used operation on a BST. By this operation we can visit each node in the tree exactly once. A fully traversal on a binary tree gives a linear ordering of data in the tree.

For example, if the binary tree contains an arithmetic expression then its traversal may gives up the expression

in infix notation, post fix notation, and prefix notation.
There are 6 possible ways:

(i) R T_L T_R (iv) T_R T_L R

(ii) T_L R T_R (v) T_R R T_L

(iii) T_L T_R R (vi) R T_R T_L

Here, T_L and T_R denotes the left and right sub trees of the root 'R'. It may notice that visit-1 and visit-4 are mirror symmetric and similarly remaining also from out of 6 possible traversals only 3 are fundamental they are preorder, inorder, postorder.

Pre order Traversal :-

In this traversal the root is visited first, then the left sub tree in pre order fashion such traversal can be defined as follows

(i) Visit the root node - R

(ii) Traverse the left sub tree of R in preorder.

(iii) Traverse the right sub tree of R in preorder.

Algorithm preorder

```
    ptr = Root
   If (ptr ≠ NULL) then
        visit (ptr)
        preorder (ptr → LC)
        preorder (ptr → RC)
   end if.
Stop.
```

Inorder Traversal :-

In this traversal before visiting the root node the left sub tree of root node is visited and then the root node and after visiting the root node the right sub tree of the root node is visited.
Visiting both the sub trees in the same manner as the tree itself such a traversal can be served as follows.

(i) Traverse the left sub tree of the root node are in inorder.
(ii) Visit the root node R.
(iii) Traverse the right sub tree of the root node are in inorder.

Algorithm Inorder
```
  ptr = Root
 if(ptr ≠ NULL) then
  Inorder (ptr → LC)
   visit (ptr)
  Inorder(ptr → RC)
  end if
stop
```

Post Order Traversal :-

In this traversal the root node is visited in the end that is first visit the left sub tree and then right sub tree and lastly the root. A definition for this type of traversal is defined as follows

(i) Traverse the left sub tree of the root node are in post order.

(ii) Traverse the right sub tree of the root node are in post order.
(iii) Visit the root node - R.

Algorithm Postorder

```
ptr = Root
if(ptr ≠ NULL) then
  postorder(ptr -> LC)
  postorder(ptr -> RC)
  visit(ptr)
end if
stop.
```

Height Balanced Binary Tree :-

A binary search tree is said to be height balanced binary search tree if all its nodes having a balance factor of 1, 0 or −1.

that is $|bf| = |h_L - h_R| \leq 1$.

bf = balance factor.

The balance factor (bf) of a binary tree is defined as

bf = Height of the left sub tree (h_L) − Height of the right sub tree (h_R).

It may be noted that a height balanced binary tree is always be a binary search tree and a complete binary search tree. is always height balanced. But the reverse may not be true.

The basic objective of the height balanced BST is to perform searching, Insertion and deletion operations effectively. These operations may not be with the minimum time. But the time involved is less than that of in an unbalanced BST. It may be realize that the searching time in the case of complete binary tree is minimum. But, insertion and deletion may not be with the minimum time always.

Ex:- bf ⑥ = 3 − 2 = 1

bf ② = 1 − 2 = −1

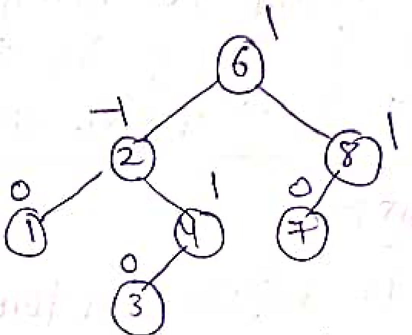
bf ⑧ = 1 − 0 = 1

bf ① = 0

bf ④ = 1 − 0 = 1. ⇐ It is height balanced tree.

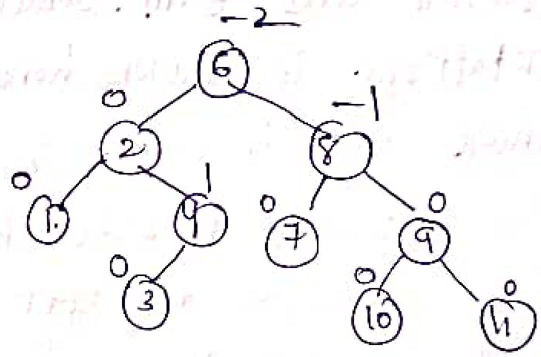
bf ③ = 0.

bf ⑦ = 0.



~~Delete node~~ ~~then~~

It is not height balanced tree ⟹



The following steps need to be adapted.

(i) Insert node in to a binary search tree:-

Insert the node in to its proper position following the properties of BST.

(ii) Compute the balance factor:-

On the path starting from the root node to the node newly inserted, compute the balance factor of each node.

(iii) Deside the pivot node:
On the path as traced in step-2 determine wheather the absolute value of any nodes balance factor is shifted from 1 to 2. If so the tree become unbalanced. the node which shifted from 1 to 2. mark as a special node called pivot node.

(iv) Balance the unbalance tree:
It is necessary to manipulate pointers created at the pivot node to bring the tree back in to height balance. This pointer manipulation is well known as AVL rotations.

(b) AVL Rotations:-

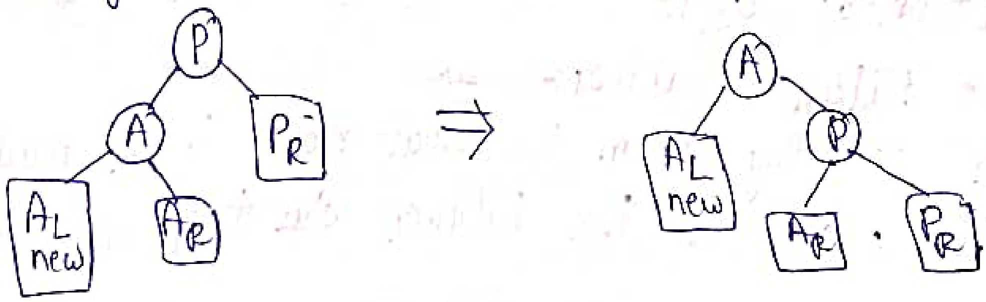
In order to balance a tree there was a method derived in 1962. by two Russian mathematicians G.M. Adelson Velskii and E.M. Lendis and the method named as AVL rotations in their honour.

There are 4 cases of rotation possible these are

Case 1: Left to Left Rotation
Case 2: Right to Right Rotation
Case 3: Left to Right Rotation
Case 4: Right to Left Rotation.

Case-1 : LL Rotation:-

Unbalance occur due to insertion in the left sub tree of the left child of the pivot.



Here, the following manipulation in the pointer takes place.

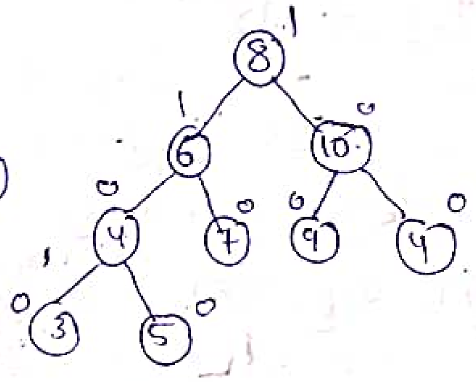
(i) Right sub tree (A_R) of the left child 'A' of pivot node (P) becomes the left child of 'P'.

(ii) 'P' becomes right child of 'A'.

(iii) Left sub tree (A_L) of 'A' remains as same.

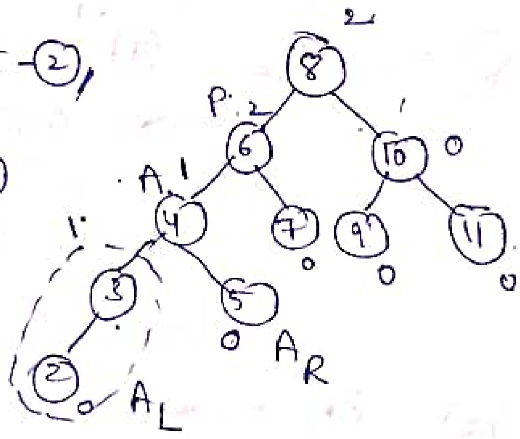
Ex:

①



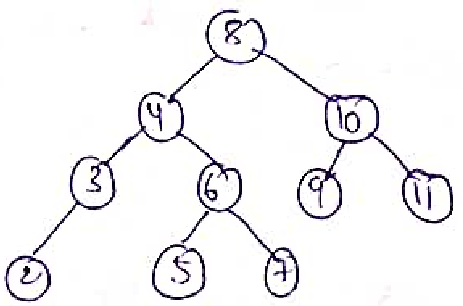
Insert ②,

②



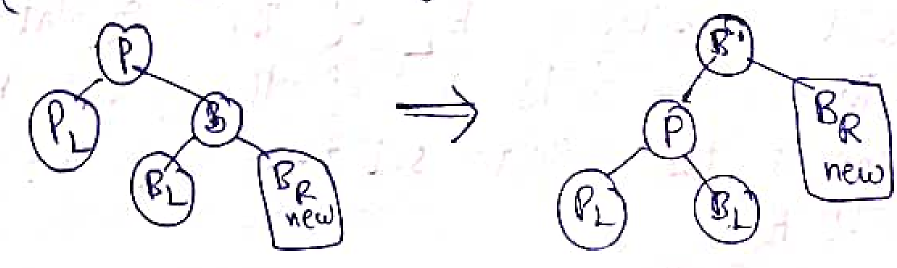
LL Rotation.

③



Case-2 :- RR Rotation:-

Unbalance occurs due to insertion in the right sub tree of the right child of the pivot node this case is the reverse and symmetric to case 1



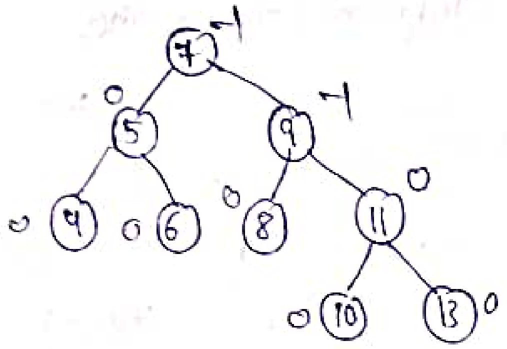
The following manipulation in pointers takes place.

(i) A left sub tree B_L of right child 'B' of pivot node (P) becomes the right sub tree of 'P'.

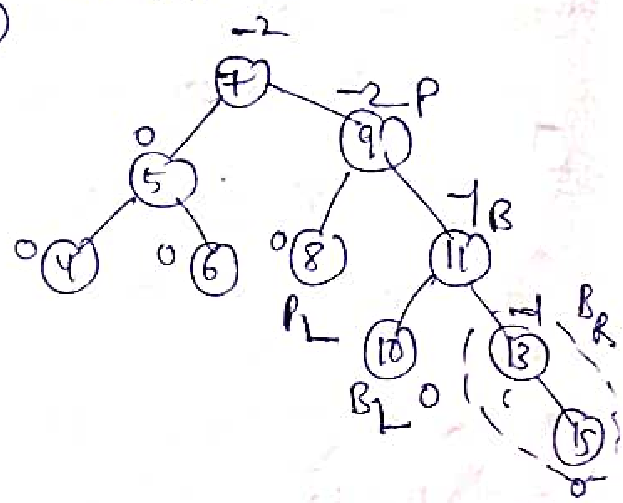
(ii) P becomes the left child of 'B'.

(iii) Right subtree B_R of B remains the same.

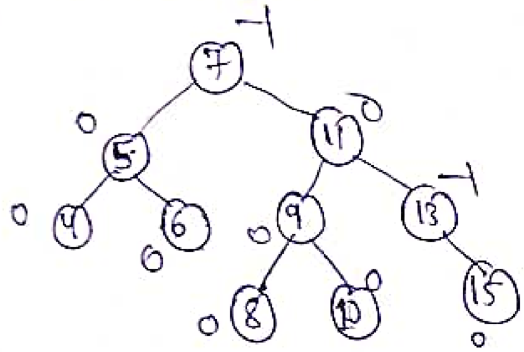
Ex:-

① 

② Insert -15



③ RR Rotation



Case-3 :- LR Rotation:-

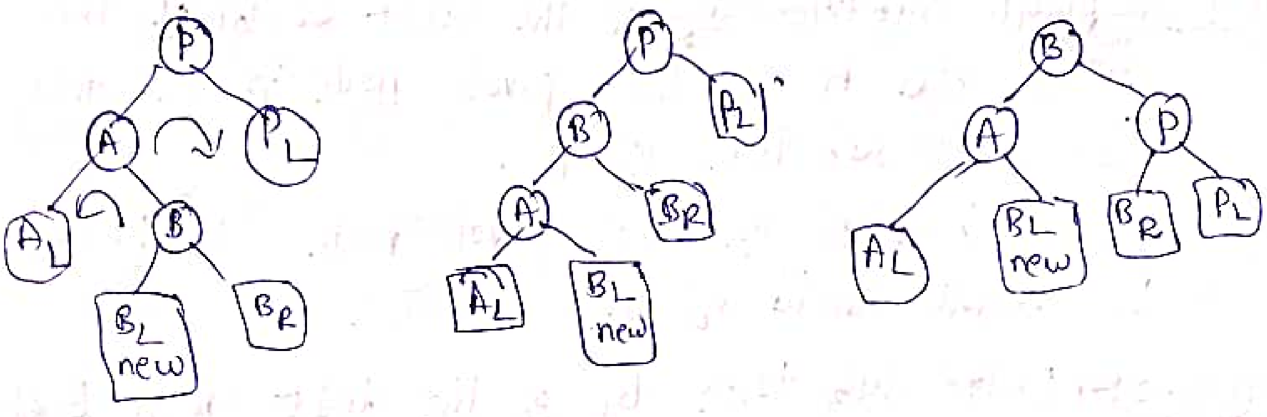
Unbalance occurs due to the insertion in the right subtree of the left child of the pivot node. This rotation involves two rotations of manipulation in pointer.

Rotation-1 :- (i) Left sub tree B_L of the right child B of the left child A of the pivot node P becomes the right sub tree of the left child A.

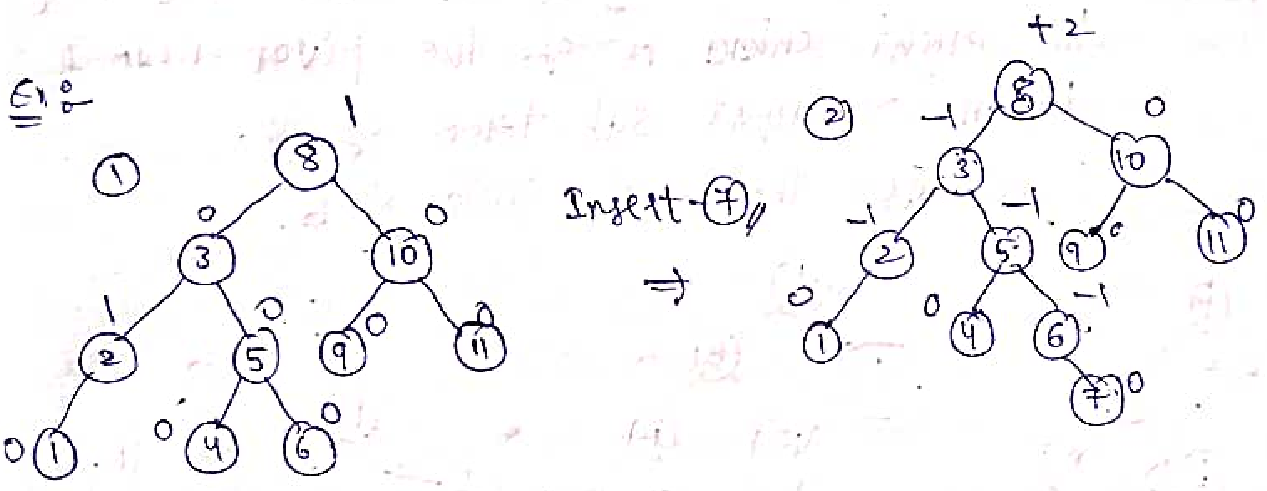
(ii) Left child A of the pivot node 'p' becomes the left child of B.

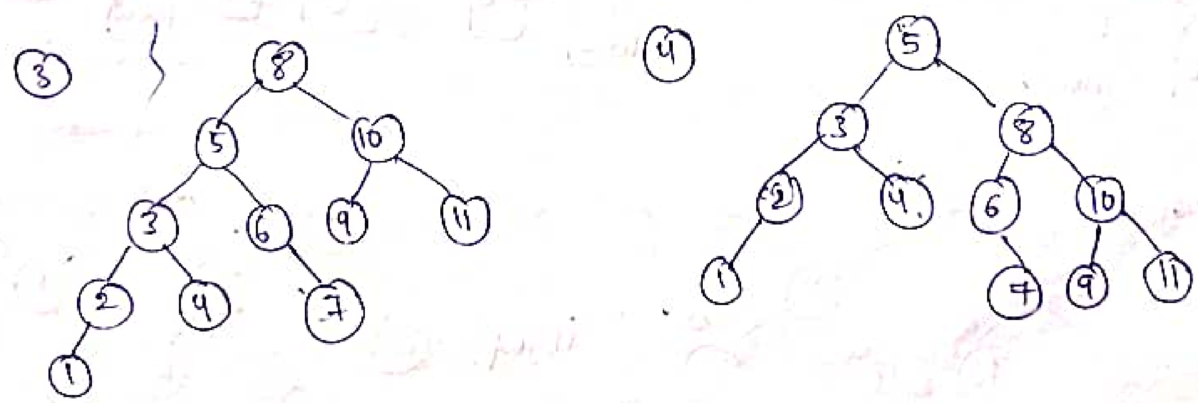
Rotation-2: (i) Right subtree B_R of the right child B of the left child A of the pivot node 'p' becomes the left subtree of P.

(ii) P becomes the right child of B.



Ex:-





Case-4:- RL Rotation:-

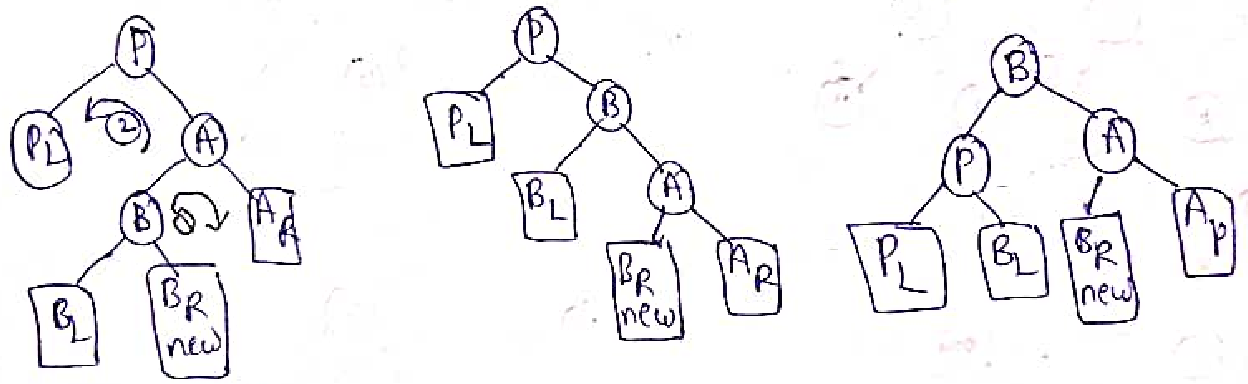
Unbalance occur due to the insertion into the left sub tree of the right child of the pivot node. This case is the reverse and symmetric to case '3'. This case is known as Right to Left Rotation. This rotation involves two rotations for the manipulation of pointers.

Rotation-1: (i) Right subtree B_R of the left child 'B' of the right child A of the pivot node 'p' becomes the left sub tree of 'A'.

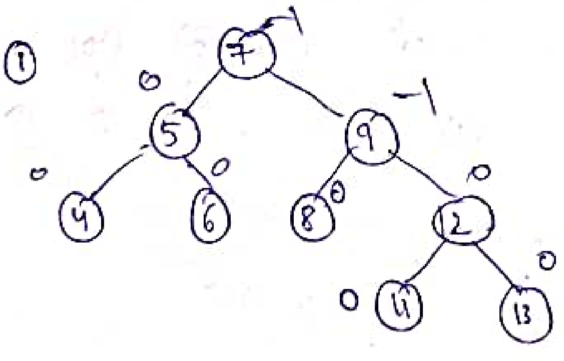
(ii) Right child 'A' of the pivot node 'p' becomes the right child of B.

Rotation-2: (i) Left sub tree B_L of the left child B of the right child A of the pivot node 'p' becomes right sub tree of 'p'.

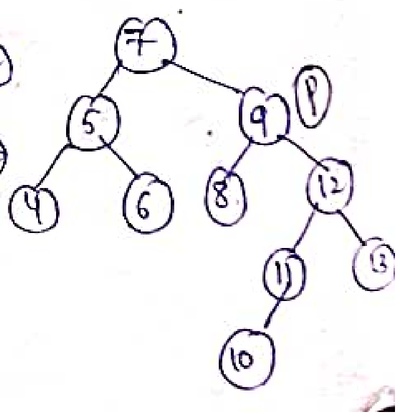
(ii) 'p' becomes the left child of B.



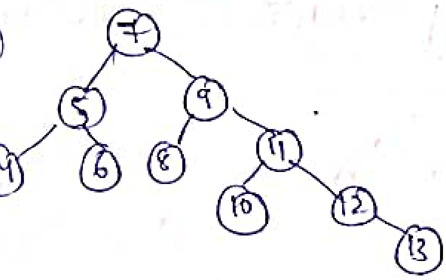
Ex: ①



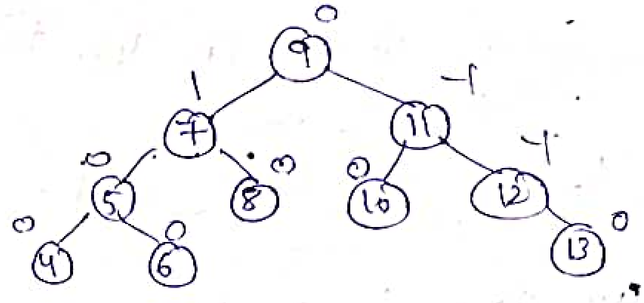
Insert (10). ⇒ ②

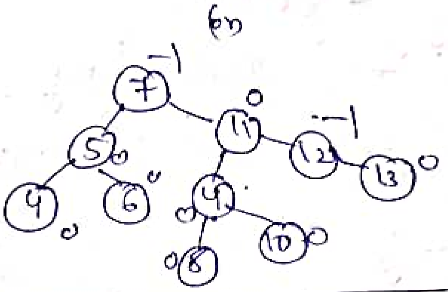


3



4





B-Trees :-

B-Trees is an extension of the m-way search tree. In order to improve the search efficiency, the tree should be balanced and if the m-way search tree is height balanced then it is called B-Tree. This structure is best suitable for maintaining the indexing of elements in it.

The main purpose of this indexing is to acclerate the search procedure. Binary Search Tree uses the concept of tree indexing, where each node contains a key value, pointers to the left sub tree and right sub tree.

BST is infact a 2-way search tree and this concept of tree indexing can be generalized for an m-way search tree $(m \geq 2)$.

i.e, (i) An m-way search tree T is a tree in which all the nodes of degree $(m \geq 2)$.

(ii) Each node in the tree may contain following attributes.

P_0	K_1	P_1	K_2	P_2	K_3	------	K_n	P_n

where, $1 \le n < m$

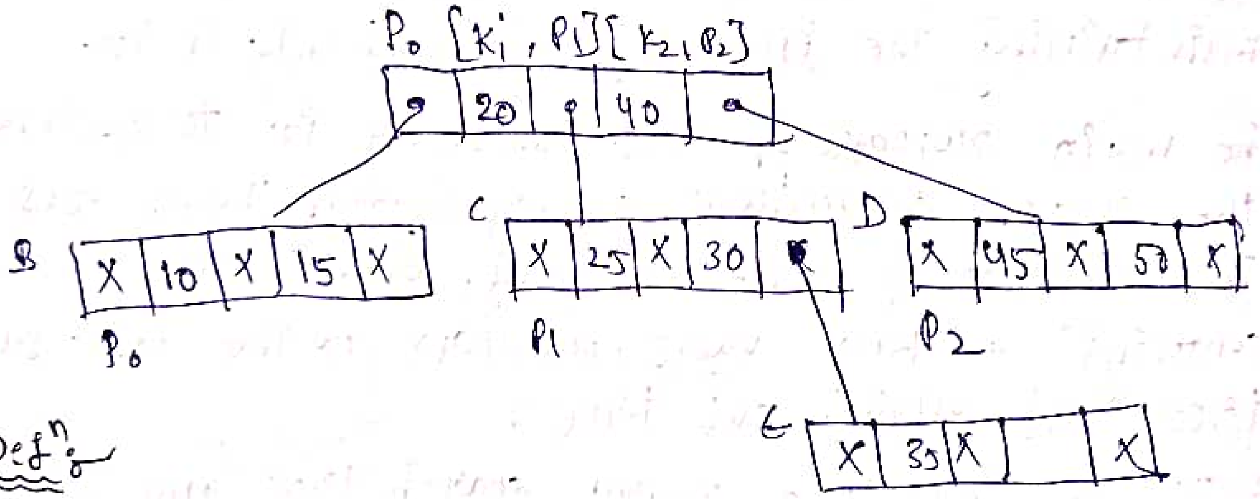
K_i $(1 \le i \le n)$ are key values in the node $(1 < i < k_{i+1})$

P_i $(0 \le i \le n)$ are pointers to the sub trees of tree(T).

All the key values in the sub trees pointed by P_i are less than the values K_{i+1} $(i = 0 \le i < n)$.

All the key values in the sub tree pointed by P_n is greater than K_i.

All the sub trees pointed by P_i are also the m-way search tree.



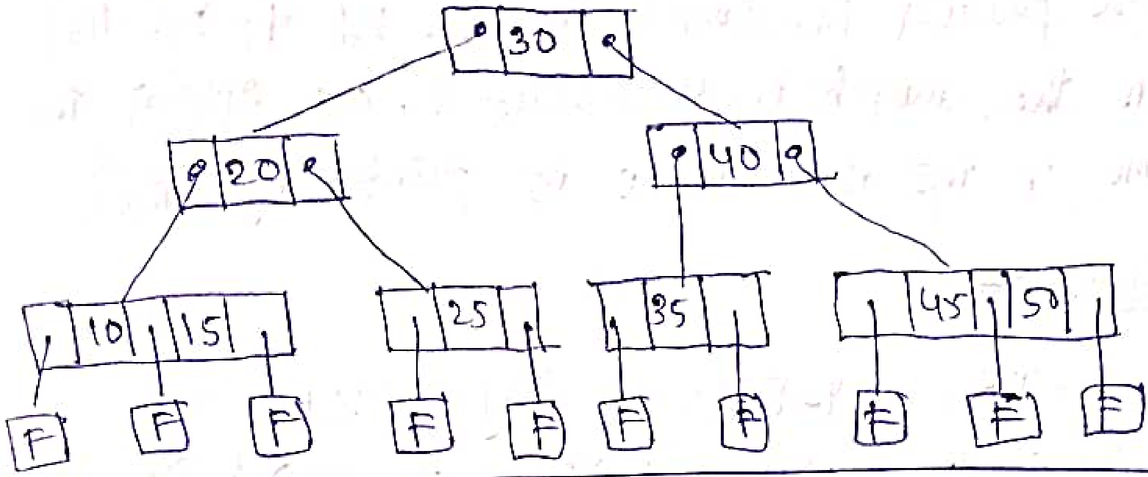
Def^n:

A B-Tree (T) of order 'm' is an m-way search tree. i.e, either it is empty or it satisfies the following properties.

(i) The root node has at least two childrens.

(ii) All the nodes other than the root node have at least $m/2$ childrens.

(iii) All failure nodes are at the same level.
A failure node is a node which can be reached during a search only if the value is not in the Tree.



Operations on B-Trees:-

The required operations are performed on B-Trees. i.e, Searching, Inserting, Deleting and Traversing.

Searching:-

Searching for a key value in a B-Tree is almost same as searching a key value in a BST, except in a BST we have to consider node which contain one key value, where as B-Tree of order-m contains atmost m-1 key values (m-childrens).

Suppose X be the item of search and there be a set of key values.
i.e, $K_1, K_2, \ldots, K_n$ and $n<m$.
First we have to perform a search over it for some i, such that $K_i \leq X \leq K_{i+1}$. The following two cases may occur.

case-i: $X = K_i$, then the search is successful and complete

case-ii: $K_i \le X < K_{i+1}$, Here no match is found then the search has to be proceeds in the sub tree whose pointer is stored in. P_i. If P_i is null then the search is unsuccessfull, else repeat the same in the node. While is pointed by P_i.

Algorithm:-

```
ptr = B.T.R , KEYS[0] = -∞ , flag = FALSE
While (ptr ≠ NULL) And (flag = FALSE) do
    For i=0 to (m-1) do
        PTRS[i] = ptr → Pi
    End For
    i=1, number=0
    While (ptr.Ki ≠ NULL) do
        KEYS[i]=ptr → Ki
        number = number + 1 , i=i+1
    End While
    KEYS[number+1] = +∞ , i=0
    While (X > KEYS[i+1])
        i=i+1
    End While
    If X=KEYS[i+1]
        flag = TRUE
        Return (flag , ptr → Pi , i+1)
        Exit
```

```
Else
    ptr = PTRS[i]
  EndIf
EndWhile
Stop.
```

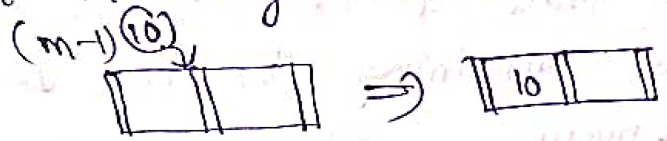
Insertion:-

To understand the insertion operation in B-Tree, it is better to construct a tree by successively inserting node starting from an empty B-Tree, with this construction we will able to highlight the various cases of insertion.

Assume that order of B-Tree is 3 and consider the elements are 10,20,30,40,50,60,70,80,90.

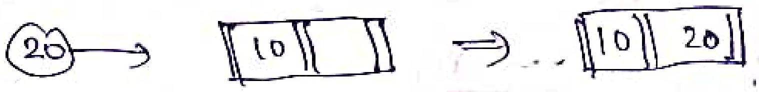
Insertion of 10:-

Initially the B-tree is empty. Get a node & insert the key value 10 in to it.



Insertion of 20:-

Before insertion remember that a node in a B-Tree of order 'm' can have almost 'm-1' key values. So, in this case the root node can hold the key value 20 after 10.



Insertion of 30:-

Before inserting 30 we should follow the steps. when a key value is to be inserted into a node which already has the maximum number of key values then the following steps need to be follow

(i) Insert the value x into the list of value in node in ascending order.

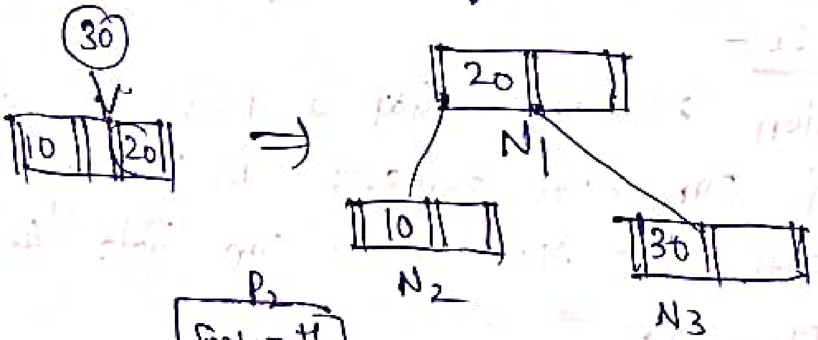
(ii) Split the list of values into 3 parts P_1, P_2 & P_3.

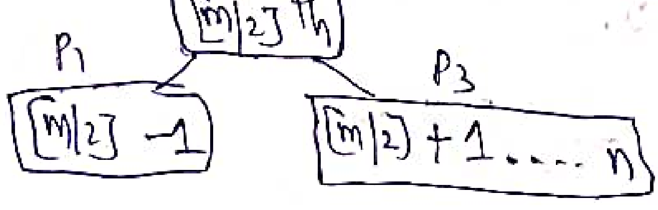
$P_1 \Rightarrow$ contains the first $[m/2] - 1$ key values.

$P_2 \Rightarrow$ Contains $[m/2]^{th}$ key value.

$P_3 \Rightarrow$ contains $[m/2] + 1$ upto 'n' key values.

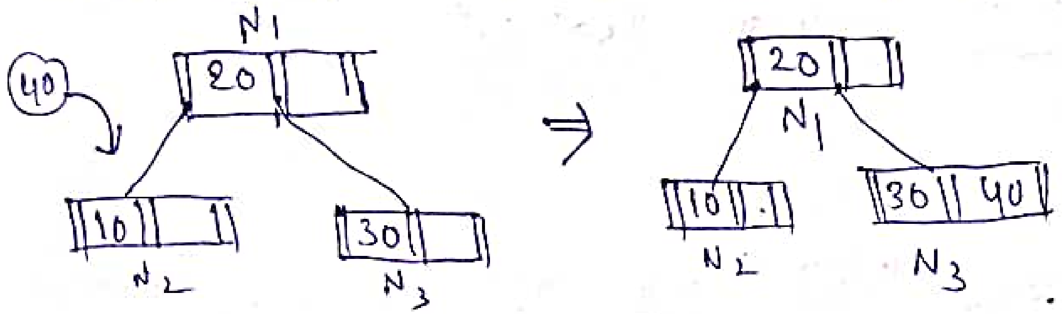
(iii) With this splitting, $\frac{m}{2}$ th value is to be inserted into the parent node of the current node if, the parent node is null then create a new node, In the place of current. Two nodes are to be allocated containing the key value in P1 and P3 respectively.





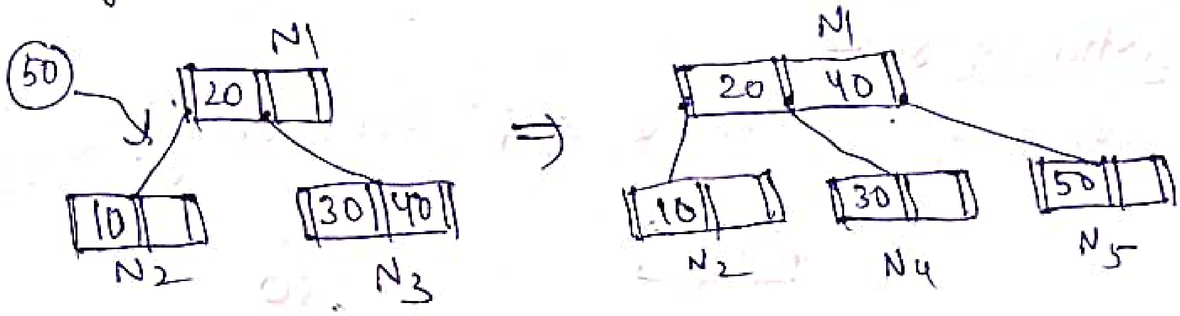
Insertion of 40:-

Insertion of 40 is the self explonatory.



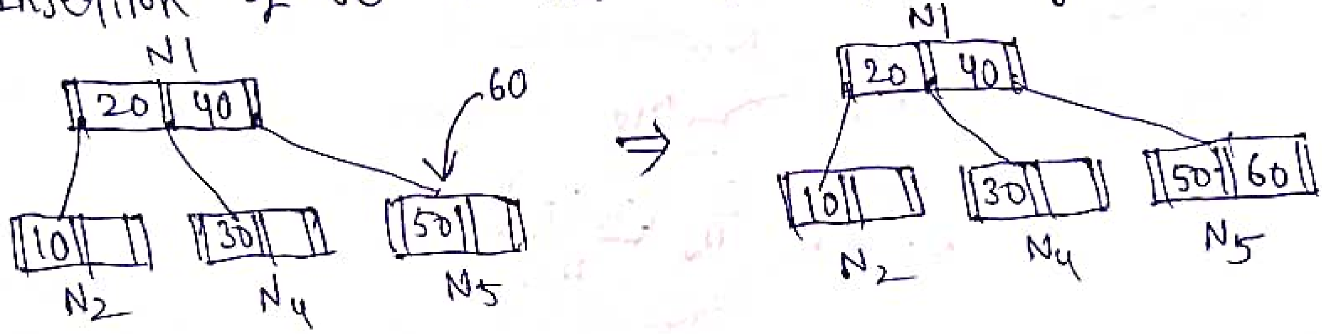
Insertion of 50:-

Insertion of 50 should go to node-3 (N3). But it is already full, so it will be split.



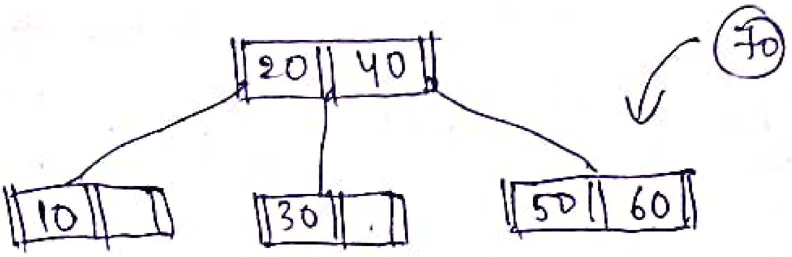
Insertion of 60:-

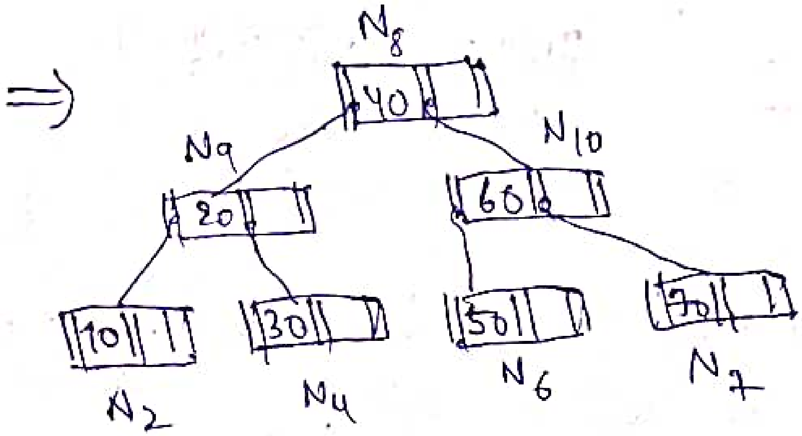
Insertion of 60 is the self explanatory.



Insertion of 70:-

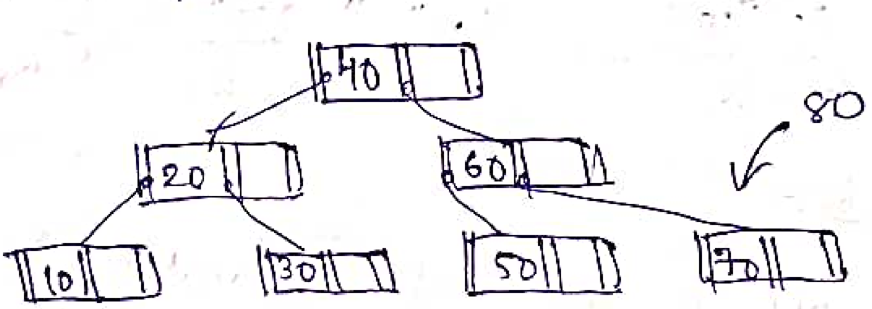
Insertion of 70 should go to node-5 (N5). Which is already full. So, it requires to split N5 in to N6, N7. This process request to insert 60 in to N1. which requiries another splitting of N1 in to N8, N9, N10.

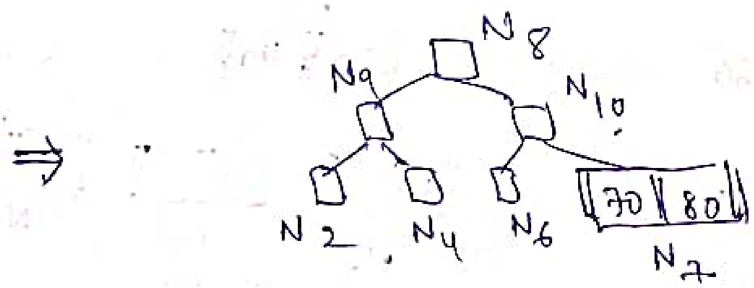




Insertion of 80:

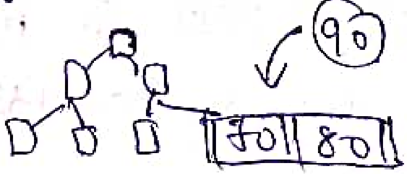
Insertion of 80 is a self explanatory.





Insertion of 90:

Insertion of 90 should go to node 7. But it is already full. So it will be split into N_{11} and N_{12}.



$\Rightarrow$