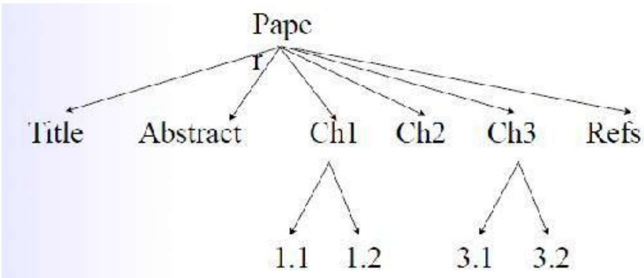
Unit II : Balanced Trees : AVL Trees: Maximum Height of an AVL Tree, Insertions and Deletions. 2-3 Trees: Insertion, Deletion, Priority Queues , Binary Heaps: Implementation of insert and delete min, creating heap.

Tree: Tree is non-linear data structure that consists of root node and potentially many levels of additional nodes that form a hierarchy.

- A tree can be empty with no nodes called the null or empty tree.
- A tree is a structure consisting of one node call the root and one or more subtrees.
- **Descendant:-** A node reachable by repeated proceeding form parent to child.
- **Ancestor:-** a node reachable by repeated proceeding from child to parent.
- **Degree:-** the number of sub-trees of a node, means the degree of an element (node) is the number of children it has. The degree of a leaf node is always 0(zero).
- **Siblings:-** Nodes with the same parent.
- **Height:-** number of nodes which must be traversed from the root to the reach a leaf of a tree.

  Examples

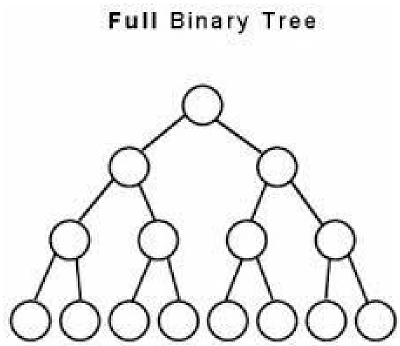
  Tree associated with a document



Binary Tree: - A binary tree is a tree data structure in which each node has at most two children, which referred as the left and right child.

Full Binary Tree or Complete Trees:

A binary tree of height is 'h' and contains exactly "2^h**-1**" elements is called full binary tree.

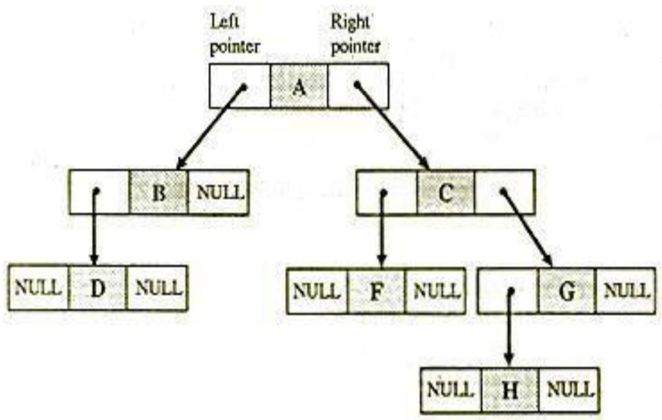


H=4 (levels+1 of root node)

n→number elements= $2^h-1=15$

(Another definition of full binary tree is, each leaf is same distance from the root)

Linked list representation of binary tree:




```
// Binary tree node structure
struct BinaryTreeNode
{
   int data;
   BinaryTreeNode *left, *right;
}*temp;
```

Operations on Binary Tree:
Create(), Insert(), Delete(), Size(), Inorder(), Preorder(), Postorder()

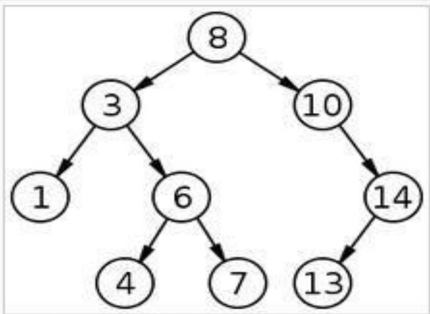
Binary Search Tree:
Binary search tree is also called ordered/sorted binary tree. Means Binary Search Tree is a node based binary tree data structure but it should satisfies following properties

- Every element (node) has a key or value & no two elements have the same key or value, therefore all keys are distinct.
- The left sub-tree of a node contains only nodes with key less than the root node's key value.
- The right sub-tree of a node contains only nodes with key greater than the root node's key value.
- The left and right sub-tree each must also be a binary search tree.
- A unique path exists from the root to every other node.

Binary search tree

Type	Tree	
Invented	1960	
Invented by	P.F. Windley, A.D. Booth, A.J.T. Colin, and T.N. Hibbard	
Time complexity in big O notation		
	Average	Worst case
Space	O(n)	O(n)
Search	O(log n)	O(n)
Insert	O(log n)	O(n)
Delete	O(log n)	O(n)

Example

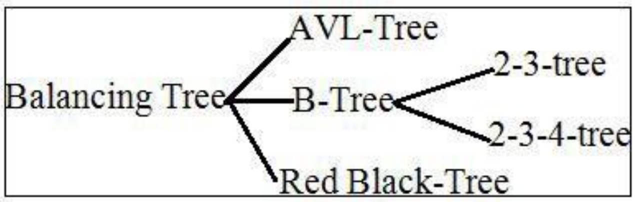


A binary search tree of size 9 and depth 3, with root 8 and leaves 1, 4, 7 and 13

Balanced Tree

Balancing or self-balancing (Height balanced) tree is a binary search tree.
Balanced tree is any node based binary search tree that automatically keeps its height (Maximum number of levels below the root) small in the face of arbitrary item insertion and deletion.

Use of Balanced tree:
Tree structures support various basic dynamic set operations including search, minimum, maximum, insert and deletion in the time proportional to the height of the tree.
Ideally, a tree will be balanced and the height will be "**log N**" where →number of nodes in the tree.
To ensure that the height of the tree is as small as possible for provide the best running time.
Examples of balancing tree



AVL Trees:

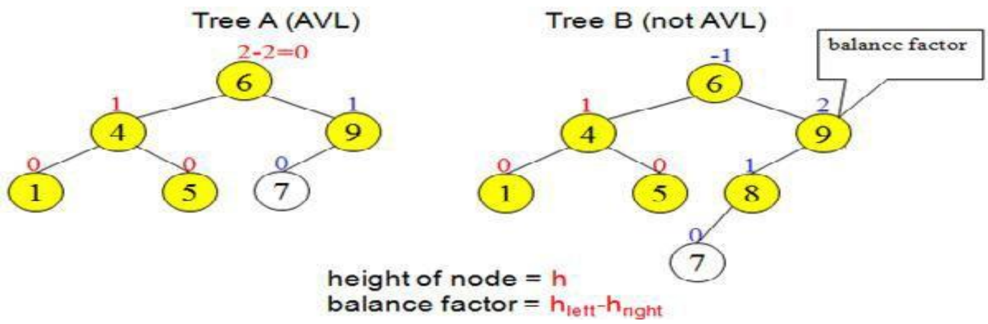
Introduction: An **AVL tree** (Adelson-Velskii and Landis' tree, named after the inventors) is a self-balancing binary search tree, invented in 1962

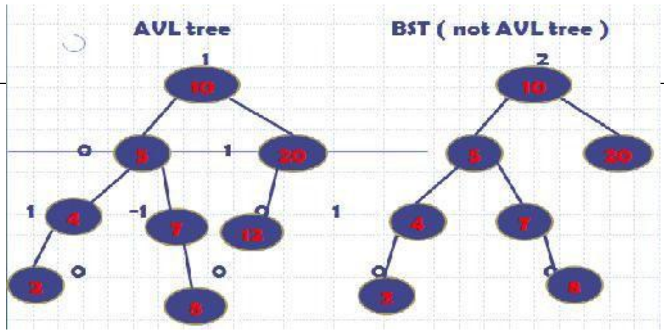
Definition: An AVL tree is a binary search tree in which the balance factor of every node, which is defined as the difference b/w the heights of the node's left & right sub trees is either 0 or +1 or -1 .

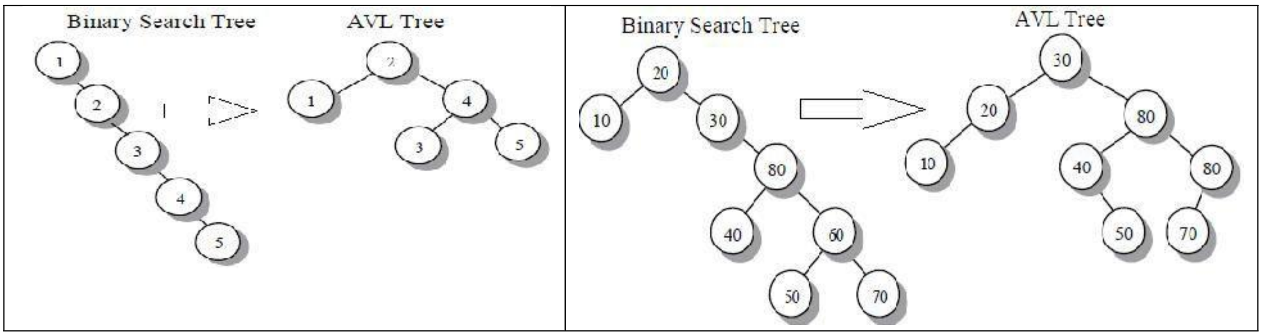
Balance factor = ht of left sub tree – ht of right sub tree.

Where ht=height

Example:







Structure or pseudo code for avl tree:

struct node
{ int data;
struct node *left,*right;
int ht;
}node;
	node *rotateright(node *);
	node *rotateleft(node *);
	node *RR(node *);
	node *LL(node *);
	node *LR(node *);
	node *RL(node *);

node *insert(node *,int);	int height(node *);
node *Delete(node *,int);	int BF(node *);

Inserting and Deleting on AVL Trees

Problem:

After insert/delete: load balance might be changed to +2 or -2 for certain nodes. _ re-balance load after each step

Requirements: re-balancing must have O (log n) worst-case complexity

Solution: Apply certain “rotation” operations

AVL tree insertion:

After inserting a node, it is necessary to check each of the node's ancestors for consistency with the rules of AVL. The balance factor is calculated as follows: balanceFactor = height (left subtree) - height(right subtree). If insertions are performed serially, after each insertion, at most one of the following cases needs to be resolved to restore the entire tree to the rules of AVL.

Let the node that needs rebalancing be α.

4 possible situations to insert in a tree

1. Insert into the **left** sub-tree of the **left** child
2. Insert into the **right** sub-tree of the **right** child
3. Insert into the **left** sub-tree of the **right** child
4. Insert into the **right** sub-tree of the **left** child

If an insertion of a new node makes an avl tree unbalanced, we transform the tree by a rotation. There are 4-types of rotation we have.

Single rotation < LL rotation / RR Rotation

Double Rotation < LR rotation / RL rotation

29

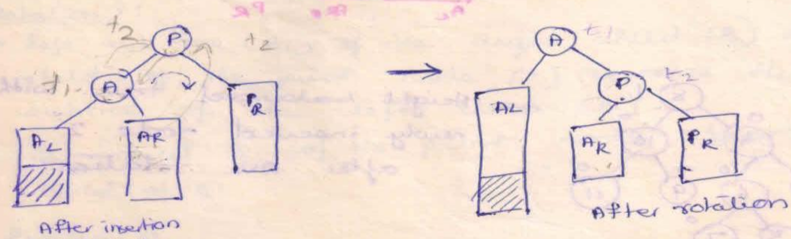
AVL ROTATIONS :

Case 1: Unbalance occured due to insertion in the left subtree of the left child of pivot node.

Manipulation of pointers takes place as

- Right subtree (A_R) of left child (A) of pivot node (P) becomes left subtree of P
- P becomes the right child of A
- Left subtree (A_L) of A remains the same.

This is called LEFT – TO – LEFT insertion



After insertion in left sub-tree of the left child of pivot node (P)

After rotation

→ AVL rotation when unbalance occurs due to the insertion in the left sub-tree of the left child of the pivot node (LEFT – TO – LEFT insertion)

ex :

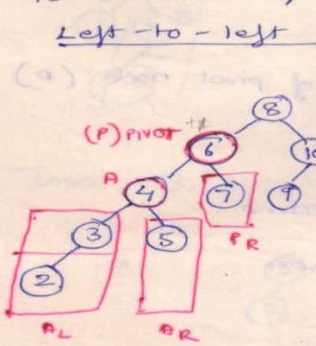
→ A height balanced binary tree as $|bf| \leq 1$

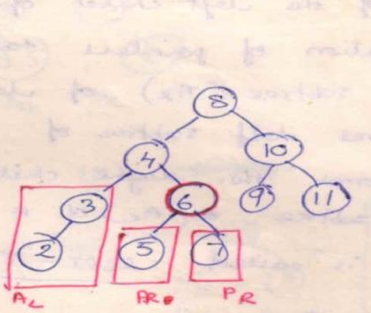
(b) After insertion of 2 into the tree

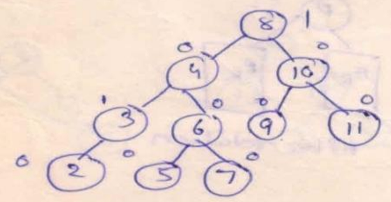
①

Here 6 is pivot node – as this is the nearest node from the newly inserted node whose balanced factor is switched from $\boxed{1 \text{ to } 2}$

Left-to-left insertion





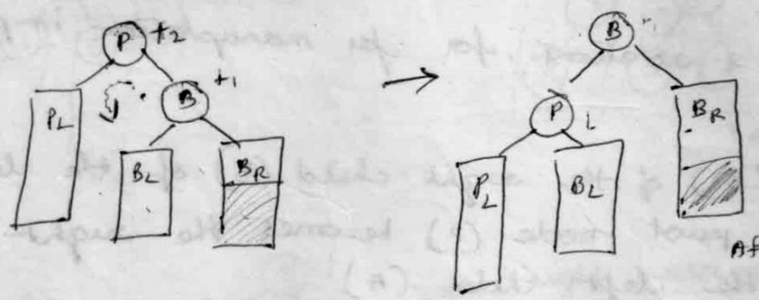


Height balanced tree with newly inserted node 2 after AVL rotation.

Case 2 : Unbalance occured due to the insertion in the right subtree of the right child of the pivot node.

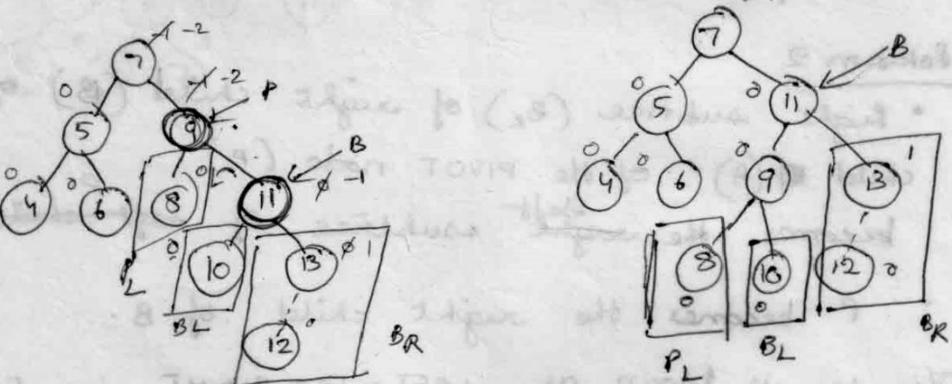
- Reverse & symetric to case 1.
- Left subtree (B_L) of right child (B) of pivot node (P) becomes the right sub tree of P.
- P becomes the left child of B.
- Right subtree (B_R) of B remains same.

This case is known as RIGHT-TO-RIGHT insertion.



After insertion in the right sub-tree of the right child of pivot node (P)

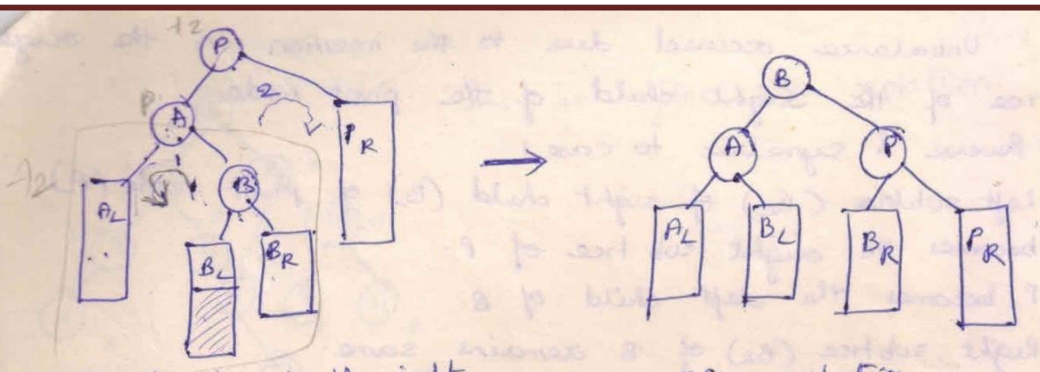
ex:-



12 is inserted which makes the tree unbalanced

After AVL rotation

Case 3 : Unbalance occured due to insertion in the right subtree of left child of the pivot node.
This is LEFT-TO-RIGHT insertion



After insertion in the right subtree of the left child of pivot node (P)

After rotation

Case 3 involves 2 rotations for ~~for~~ manipulation in pointer

Rotation 1:

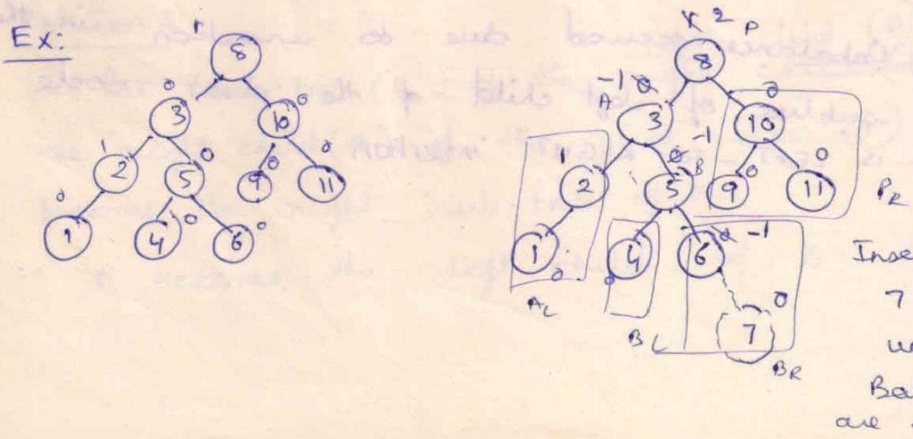
- Left sub-tree (B_L) of the right child (B) of the left child of the pivot node (P) becomes the right subtree of the left child (A).
- Left child A of the pivot node becomes the left child of B.

Rotation 2

- Right subtree (B_R) of right child (B) of left child of (A) of the PIVOT node (P) becomes the ~~right~~ left subtree of ~~left child of~~ P (A)
- P becomes the right child of B.

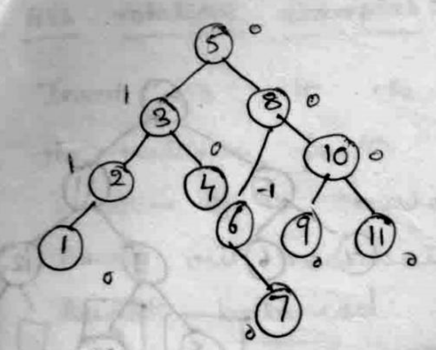
This case is known as LEFT-TO-RIGHT insertion.
The insertion occurs at B_R instead of B_L.

Ex:



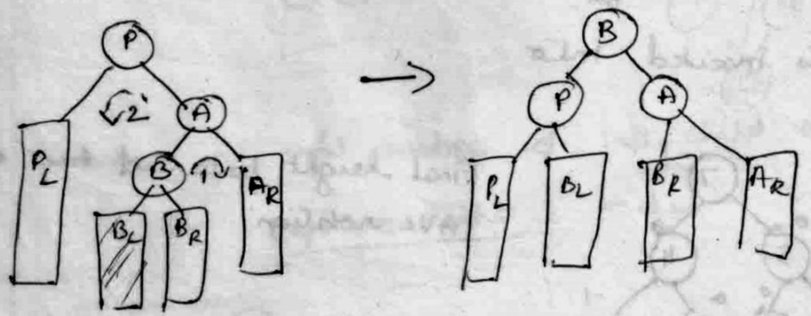
Insertion of 7 made tree unbalanced. Balanced factors are recomputed

This is after AVL rotation



Case 4: Unbalance occured due to insertion in the left-sub-tree of right child of the pivot node.

Reverse of case 3.
This is known as RIGHT-TO-LEFT insertion.



Rotation 1:

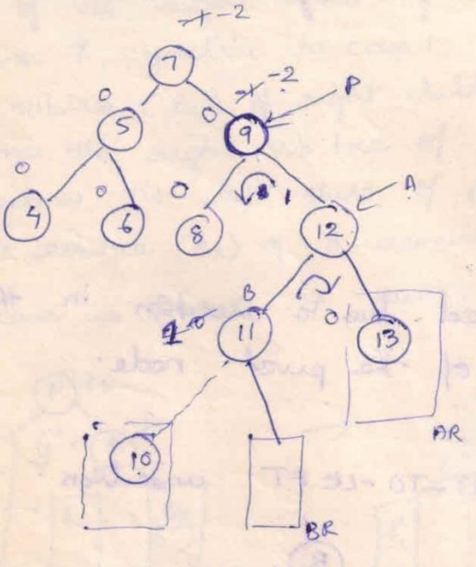
- The right sub tree (B_R) of the left child (B) of the right child (A) of the pivot node (P) becomes the left sub-tree of A
- Right child (A) of the pivot node (P) becomes the right child of B.

Rotation 2:

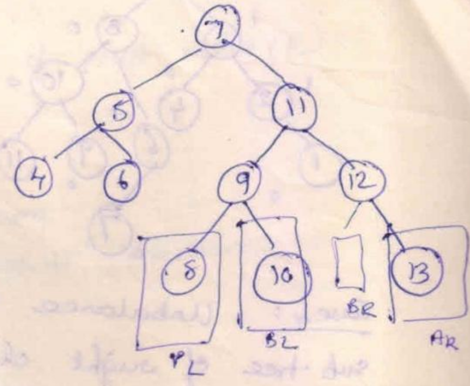
- Left sub-tree (B_L) of the ~~right child (B)~~ left child (B) of the right child (A) of the pivot node (P) becomes the right sub-tree of P.
- P becomes the left child of B

ex: This case is right-to-left insertion.

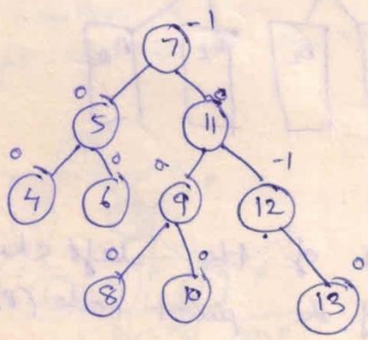
ex:



when 10 is inserted into the tree



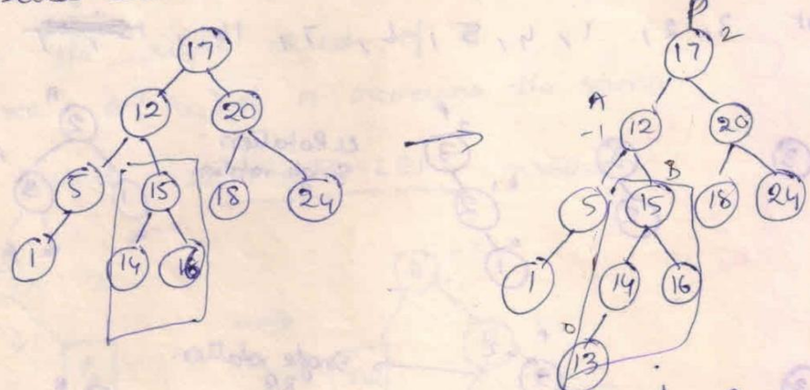
After AVL rotation



Final height balanced tree after AVL rotation

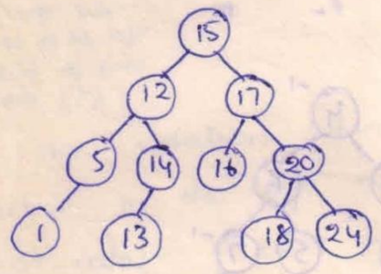
AVL rotations examples:

Insert 13 into the below tree and recompute the balance factor of each node. Show that insertion of 13 makes it an unbalanced tree. carry out necessary AVL rotations to make it height balanced.

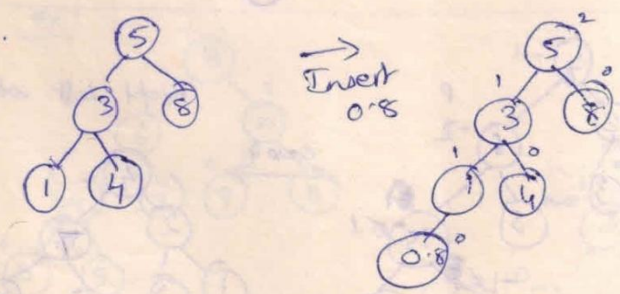


Insertion in right subtree of left child of P

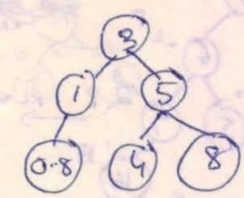
∴ Do Left to Right - dsl

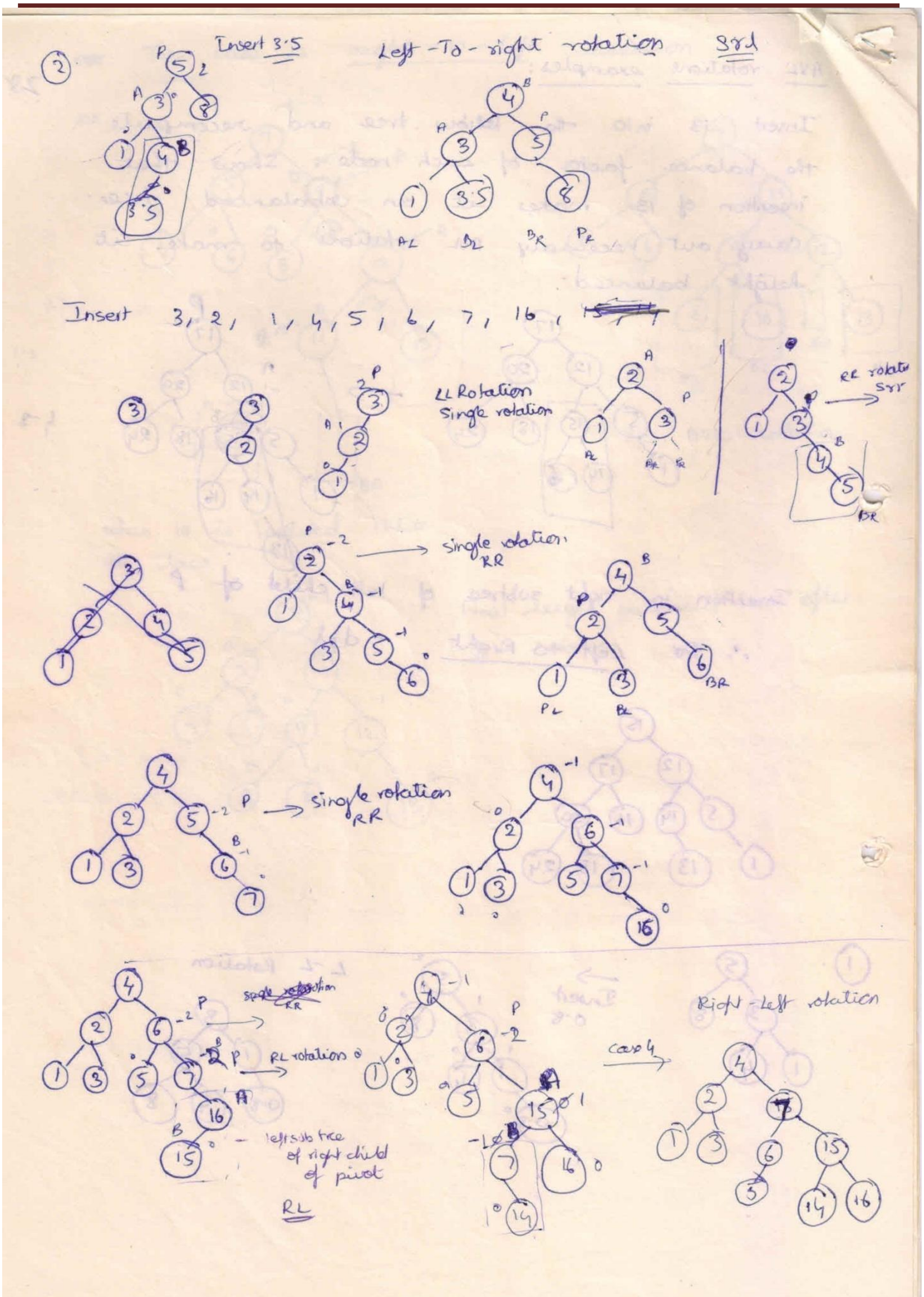


1.



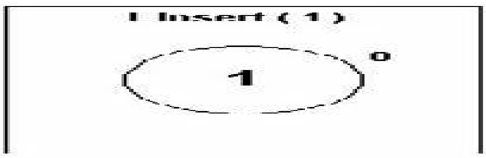
L-L Rotation

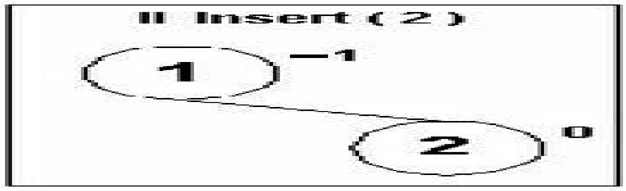


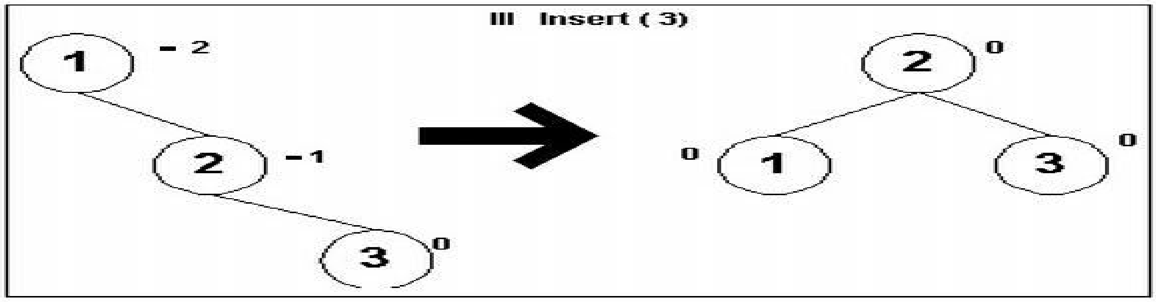

Insert 3.5
Left-To-right rotation
Insert 3, 2, 1, 4, 5, 6, 7, 16,
LL Rotation
Single rotation
RR rotation
single rotation RR
single rotation RR
RL rotation
left sub tree of right child of pivot
RL
case 4
Right-Left rotation

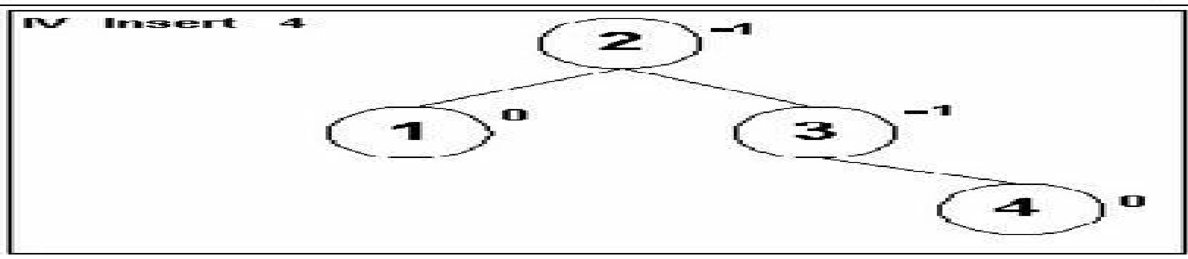
Example of Insertion of 1, 2, 3, 4, 5, 0, 7, 6 into an AVL Tree

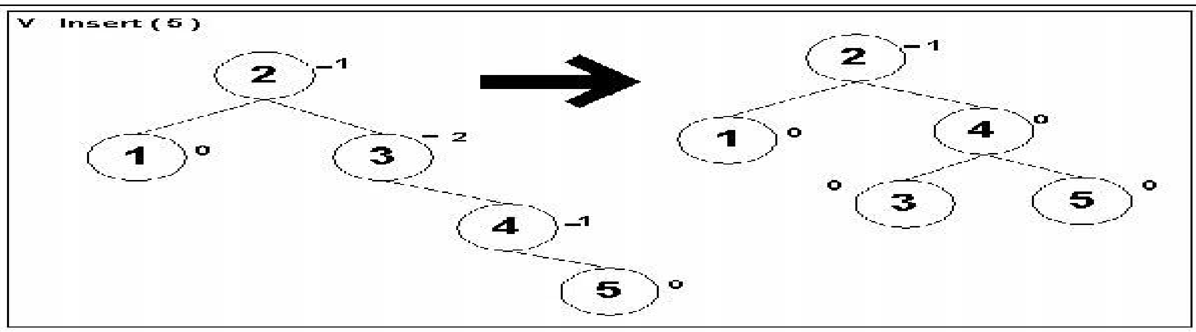
All insertions are **right-right** and so rotations are all **single rotate** from the **right**. All but two insertions require re-balancing:

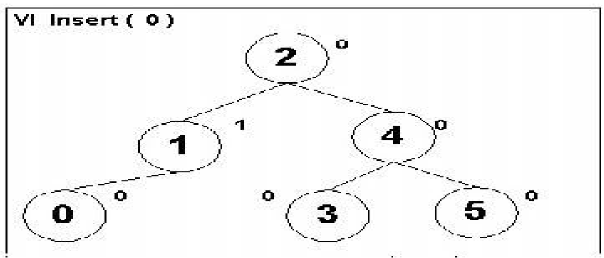


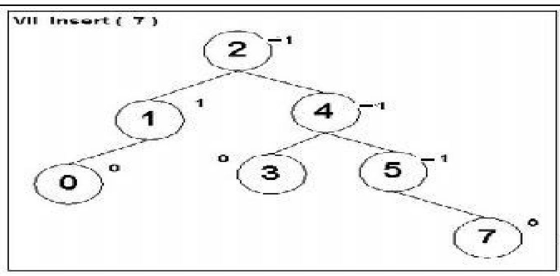




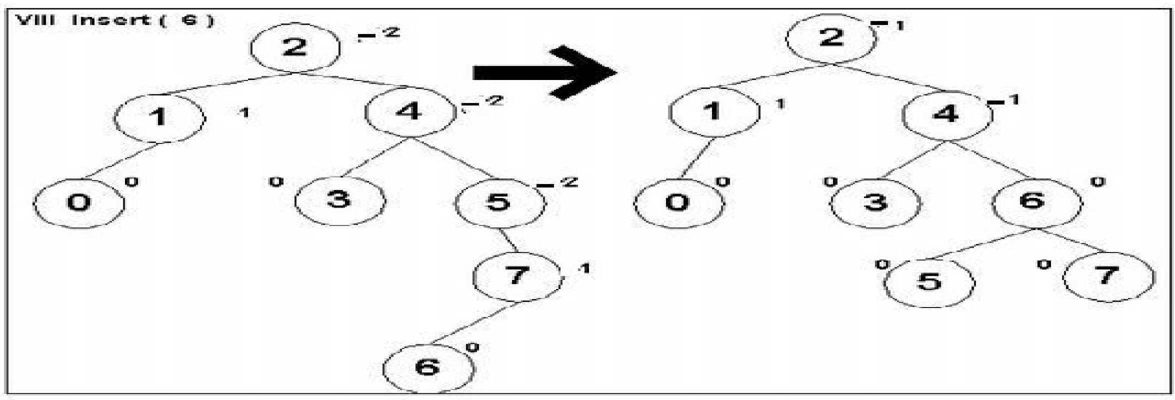




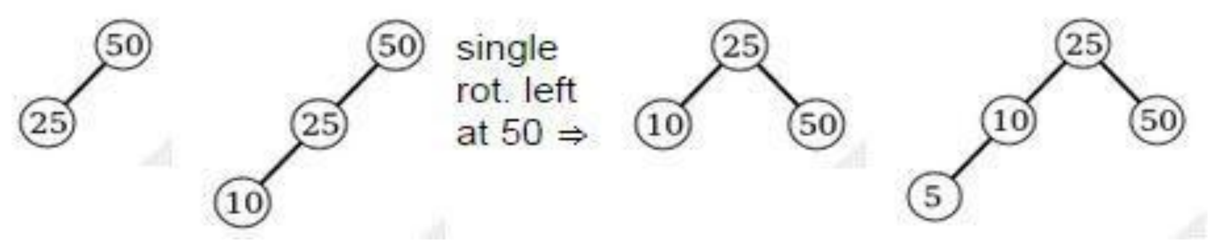


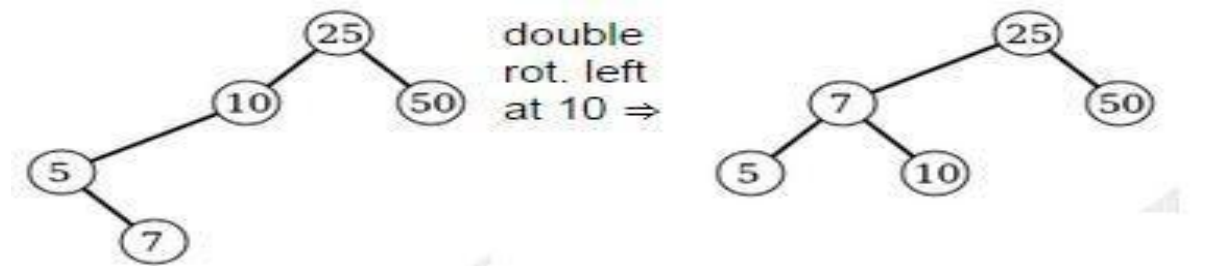


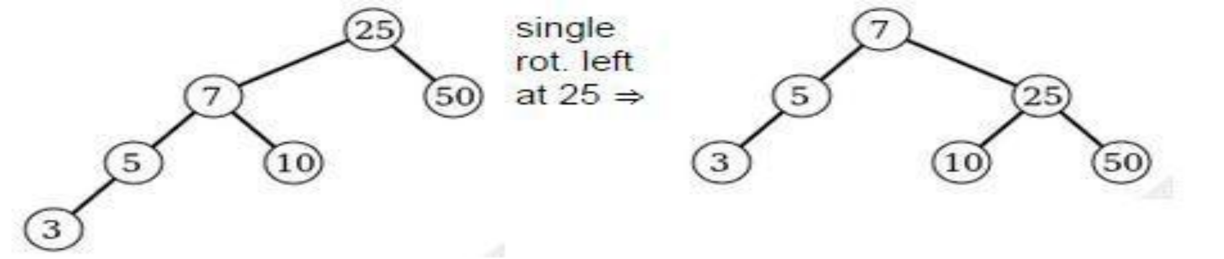
All insertions are **right-left** and so double rotations take place form left-right and right-to left

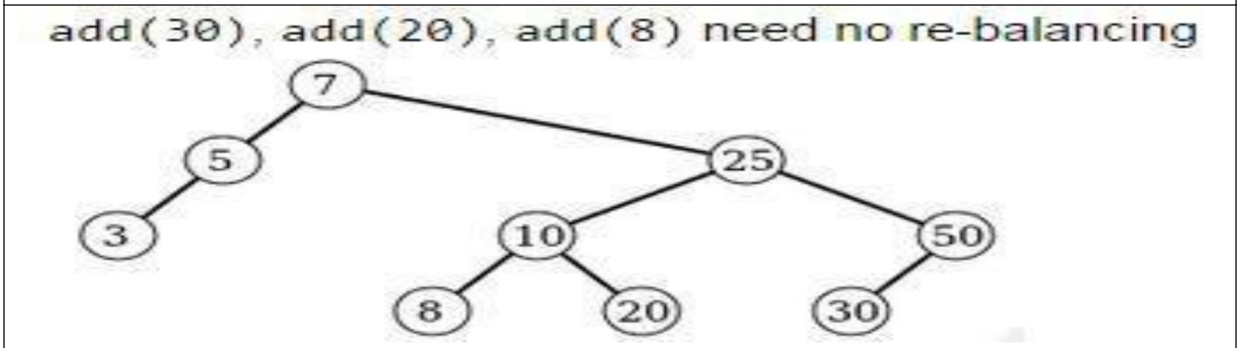


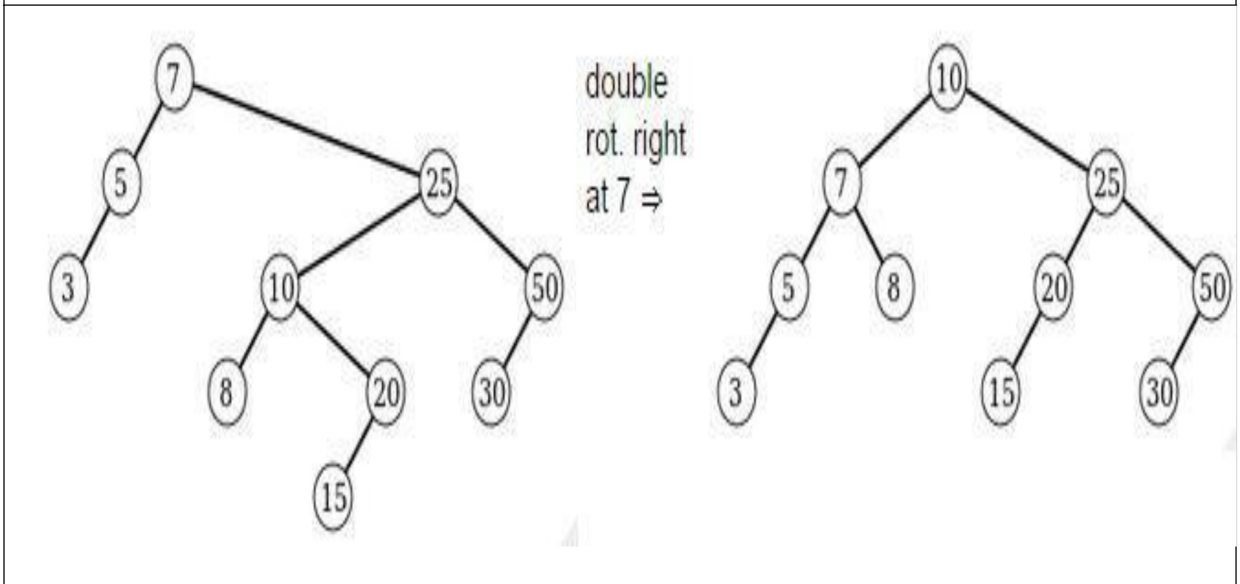
Another example. The insertion sequence is: 50, 25, 10, 5, 7, 3, 30, 20, 8, 15











```
node * insert(node *T,int x)
{
   if(T==NULL)
   {
      T=(node*)malloc(sizeof(node));
      T->data=x;
      T->left=NULL;
      T->right=NULL;
   }
   else
      if(x > T->data)               // insert in right sub-tree
      {
         T->right=insert(T->right,x);
         if(BF(T)==-2)
            if(x>T->right->data)
               T=RR(T);
            else
               T=RL(T);
      }
      else
         if(x<T->data)  // insert in left sub-tree
         {
            T->left=insert(T->left, x);
            if(BF(T)==2)
               if(x < T->left->data)
                  T=LL(T);
               else
                  T=LR(T);
         }
         T->ht=height(T);
         return(T);
}
```

```
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);
}
```

```
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);
}
```

```
node * RR(node *T)
{
   T=rotateleft(T);
   return(T);
}
```

```
node * LL(node *T)
{
   T=rotateright(T);
   return(T);
}
```

```
node * LR(node *T)
{
   T->left=rotateleft(T->left);
   T=rotateright(T);
   return(T);
}
```

```
node * RL(node *T)
{
   T->right=rotateright(T->right);
   T=rotateleft(T);
   return(T);
}
```

```
node * rotateleft(node *x)
{
   node *y;
   y=x->right;
   x->right=y->left;
   y->left=x;
   x->ht=height(x);
   y->ht=height(y);
   return(y);
}
```

```
node * rotateright(node *x)
{
   node *y;
   y=x->left;
   x->left=y->right;
   y->right=x;
   x->ht=height(x);
   y->ht=height(y);
   return(y);
}
```

Search Operation in avl tree:

Search operation of avl tree is same as the search operation of binary search tree. Means given element is checked with the root element,

- If the given element is match with the root element then return the value
- If the given element is less than the root element then the searching operation is continued at left sub-tree of the tree.
- If the given element is greater than the root element then the searching operation is continued at right sub-tree of the tree.

```
node *search(node *root, int key, node **parent) {
   node *temp;
   temp = root;
   while (temp != NULL) {
      if (temp->data == key) {
         printf("\nThe %d Element is Present", temp->data);
         return temp;
}
      *parent = temp;
       if (temp->data > key)
               temp = temp->lchild;
      else
              temp = temp->rchild;
   }
   return NULL;
}
```

Deletion of node in avl tree:

●Deletion:

■Case 1: if X is a leaf, delete X

■Case 2: if X has 1 child, use it to replace X

■Case 3: if X has 2 children, replace X with its inorder predecessor(and recursively delete it)

Algorithm:

Step 1: Search the node which is to be deleted.

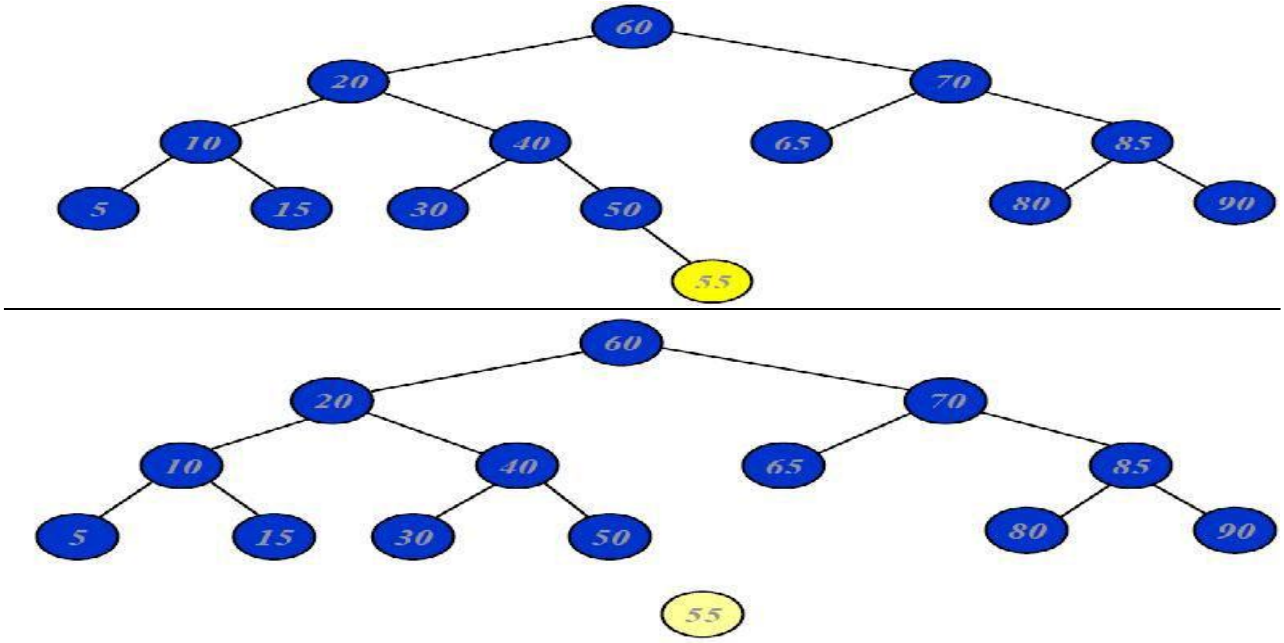
If the node to be deleted is a leaf node then simply delete that node and make be null

If the node to be deleted is not a leaf-node, i.e., that node have one or two children then that node must be swapped with its in order successor. Once the node is swapped we can remove the required node.

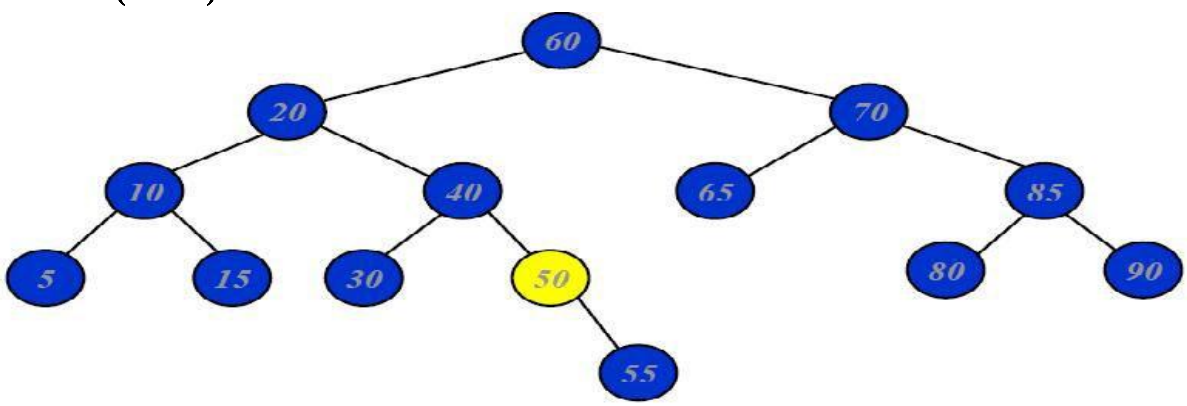
Step: 2:- Now we have to traverse back up the path towards the root node checking balance factor of every nod along that path.

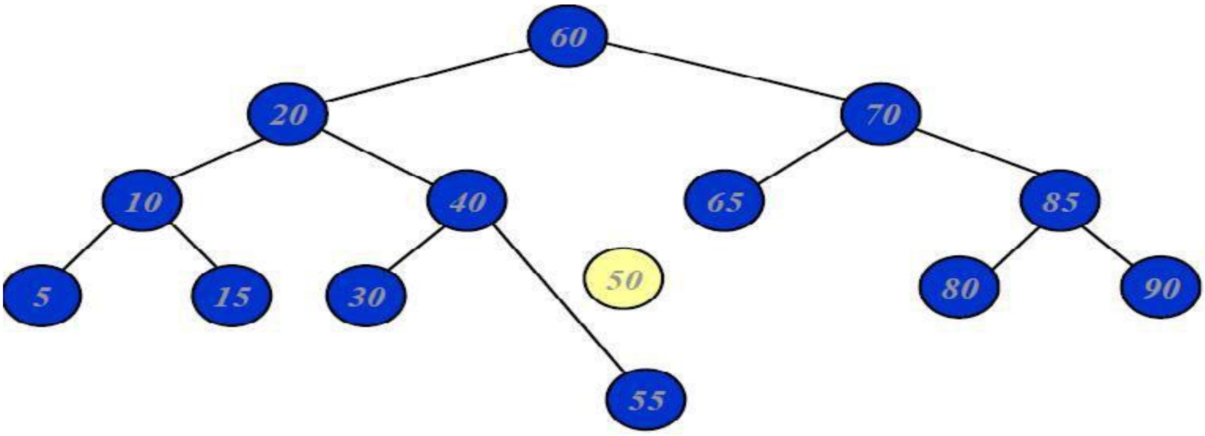
Step 3:-If we encounter unbalancing in some sub tree than balance that sub tree using appropriate single or double rotations.

Delete 55 (case 1)

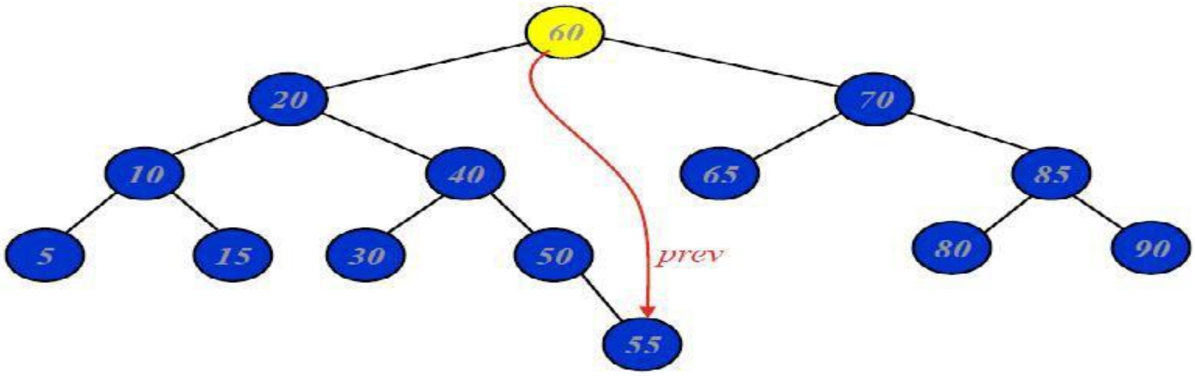


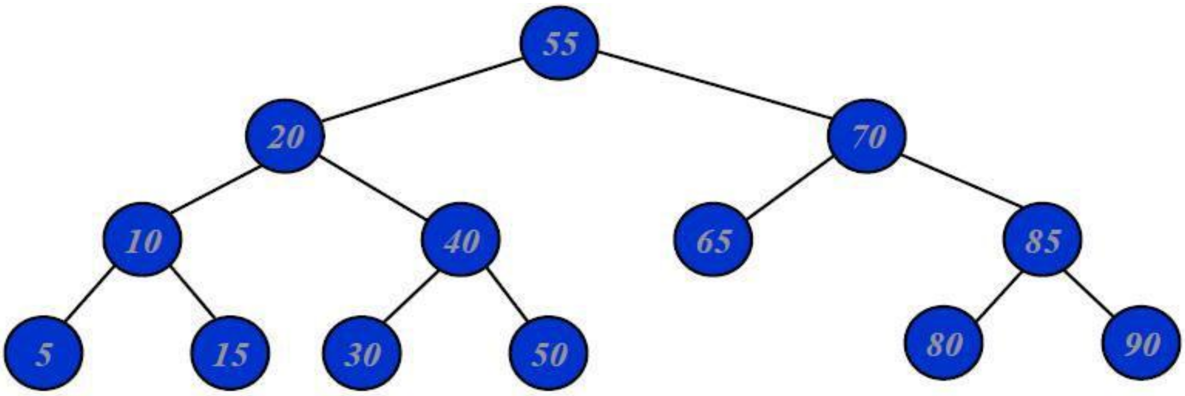
Delete 50 (case 2)



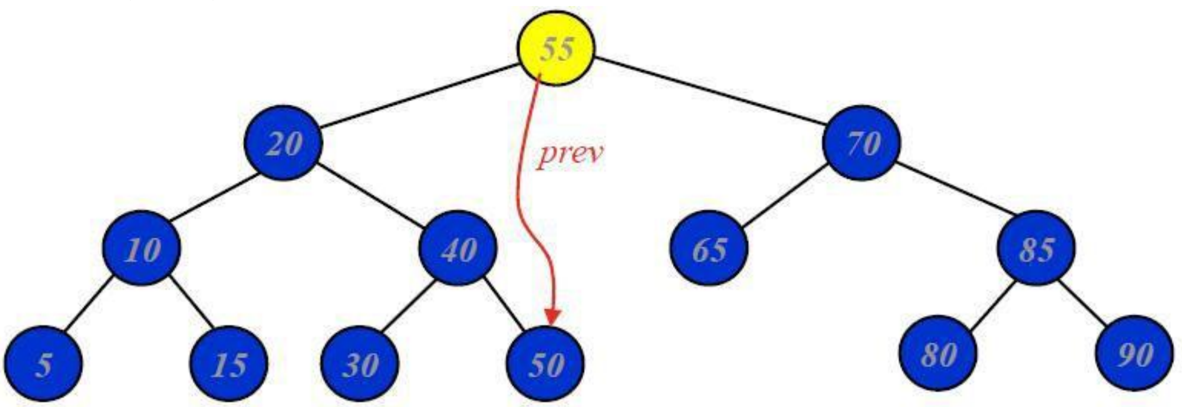


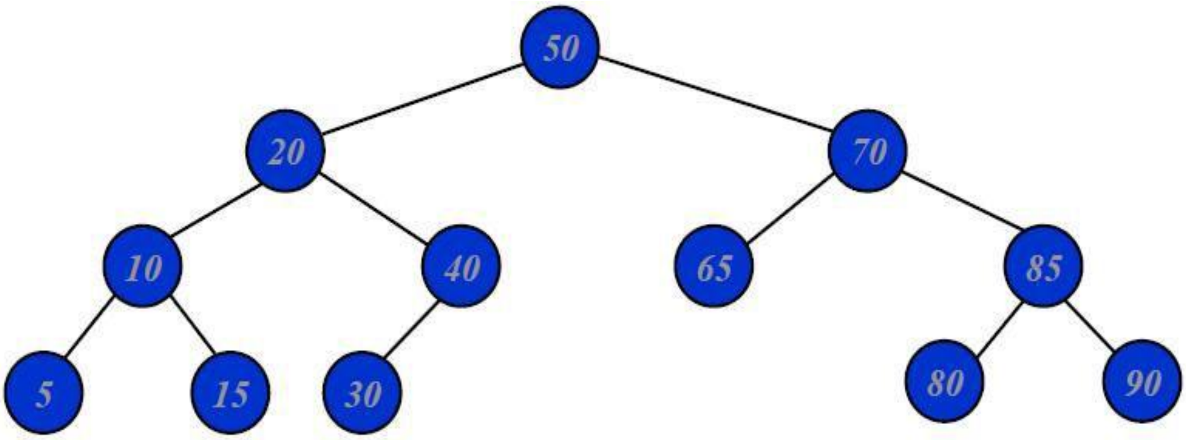
Delete 60 (case 3)



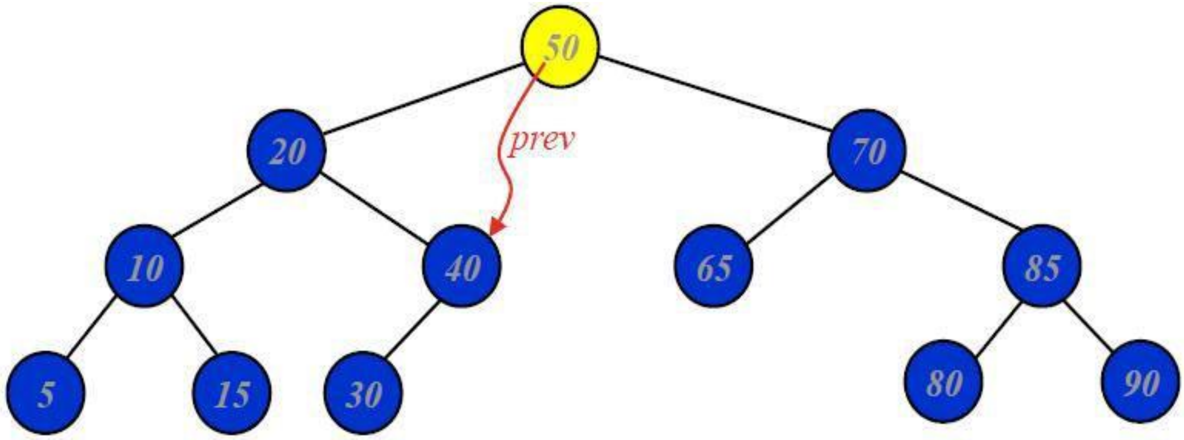


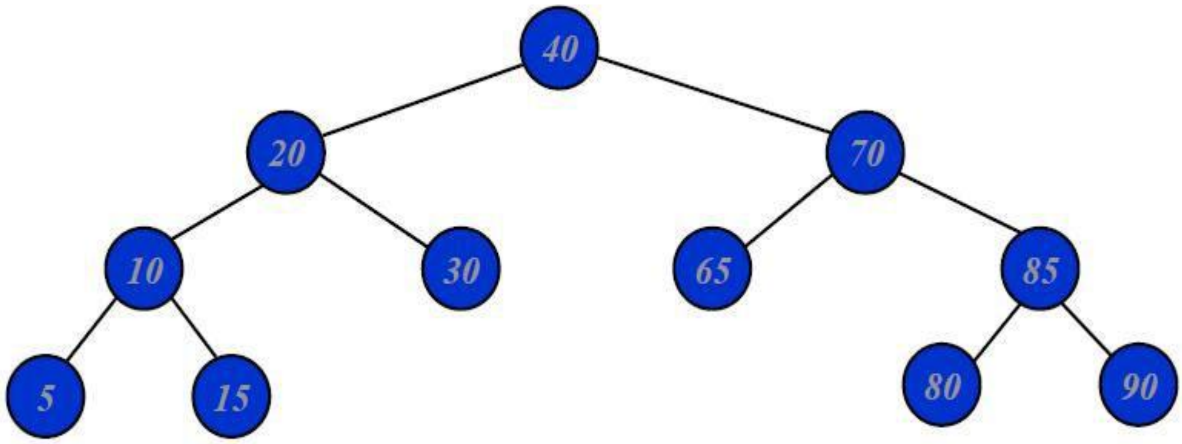
Delete 55 (case 3)



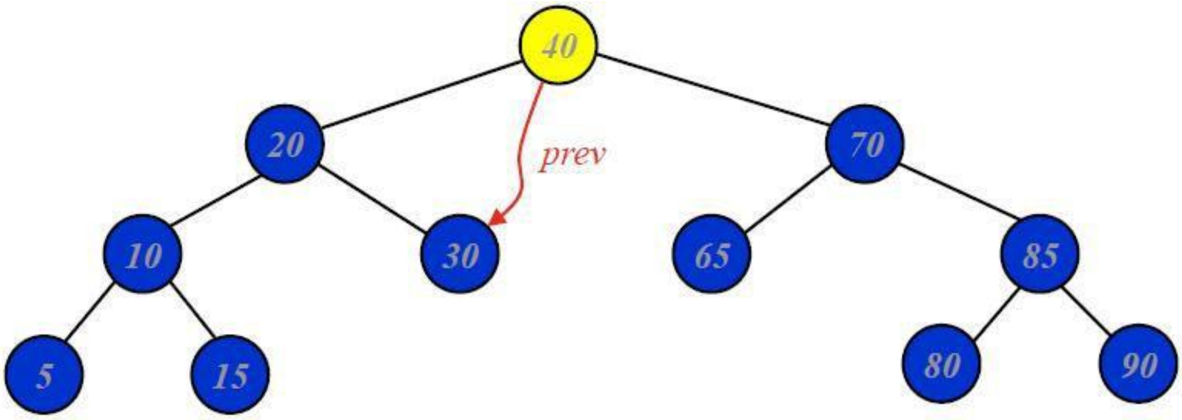


Delete 50 (case 3)

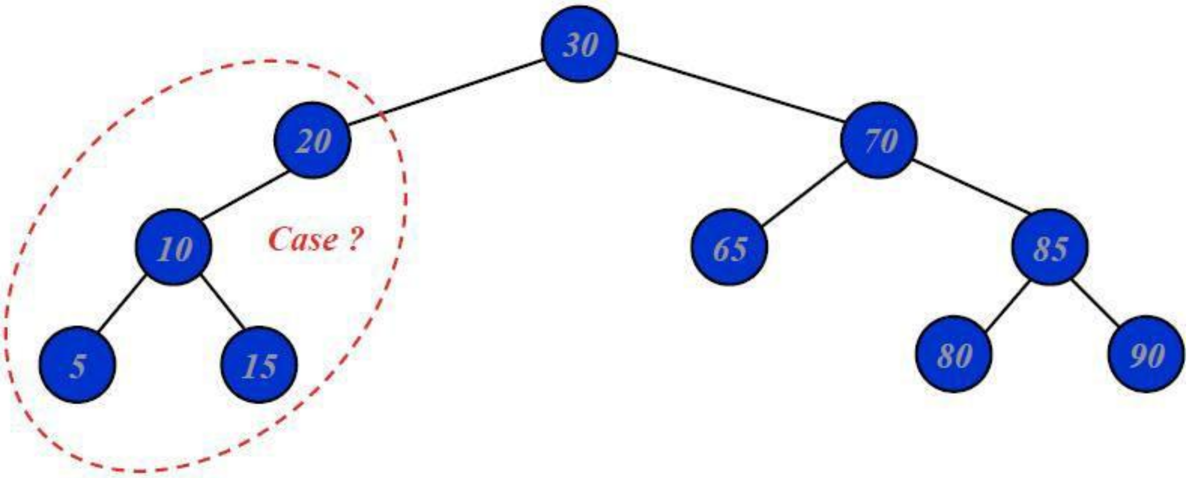




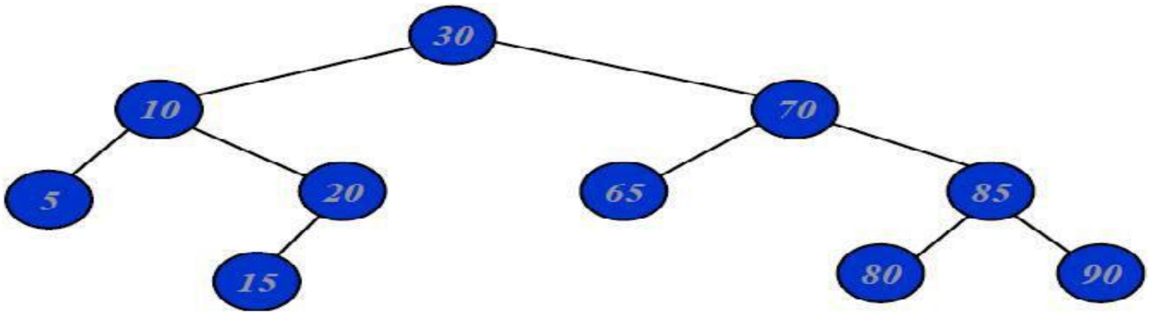
Delete 40 (case 3)



Delete 40 : Rebalancing



Delete 40: after rebalancing



```
node * Delete(node *T,int x)
{      node *p;
   if(T==NULL)     {        return NULL;    }
   else
      if(x > T->data)               // insert in right subtree
      {
         T->right=Delete(T->right,x);
         if(BF(T)==2)
            if(BF(T->left)>=0)
               T=LL(T);
            else
               T=LR(T);
      }
      else
         if(x<T->data)       {
               T->left=Delete(T->left,x);
               if(BF(T)==-2)//Rebalance during windup
                  if(BF(T->right)<=0)
                     T=RR(T);
                  else
                     T=RL(T);  }
         Else    {
            //data to be deleted is found
              if(T->right !=NULL)
              {  //delete its inorder succesor
                 p=T->right;
                 while(p->left != NULL)
                 p=p->left;
                 T->data=p->data;
                 T->right=Delete(T->right,p->data);
                 if(BF(T)==2)//Rebalance during windup
                 if(BF(T->left)>=0)
                     T=LL(T);
                  else
                     T=LR(T);
               }
              else
               return(T->left);
            }
   T->ht=height(T);
   return(T);
}
```

AVL tree

Type	Tree
Invented	1962
Invented by	G. M. (A)delson, (V)elskii & E. M.(L)andis

Time complexity in big O notation

	Average	Worst case
Space	O(n)	O(n)
Search	O(log n)	O(log n)
Insert	O(log n)	O(log n)
Delete	O(log n)	O(log n)

Red-Black Tree:

RED - BLACK Trees

29

- A red-black tree is an extended binary search tree in which the nodes and the edges from which these nodes emnate are either red or black and satisfy the following properties:

(i) The root node & external nodes are always black nodes.

(ii) [Red condition] - No 2 red nodes can occur consecutively on the path from root node to an external node.

(iii) [Black condition] - The no of black nodes on the path from the root node to the external node must be the same for all external nodes.

- As the colour of node is same as the colour of edge from which node emanates

- The red black condition may be expressed in terms of edges as well.

[Red Condition]: No 2 red pointers or edges can occur consecutively on the path from the root node to the an external node.

[Black Condition]: All paths from root node to external nodes must have same number of black pointers.

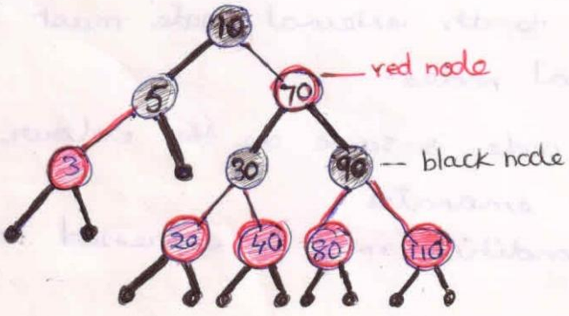
- All ~~nodes~~ pointers linking internal nodes with external nodes must be black.

→ The no of black nodes or edges on the path from a node to an external node is called the RANK of the node.

→ The rank of all external nodes is 0.

①

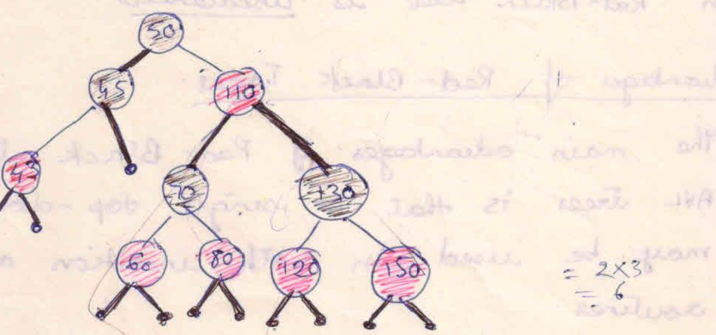
Red-black tree



Representation of red-black tree:

- A red-black tree is extended binary Tree.
- As colour of node plays domient role in def of red-black tree it is essential to record colour COLOUR.
- Another field to record colour, of 2 pointer enanating from the node.
- The INSERT, DELETE - unbalance the red-black tree.

Red- Black Tree Example: 25



Observations :

1. The Tree is an extended binary search tree with its root and external nodes to be black.

2. [Red Condition] - No 2 consecutive red nodes can occur is satisfied on all paths from root to external nodes.

3. [Black condition] - All root-to-external node paths must contain the same number of black nodes is also true.

- Every such path is given tree has exactly 2 black nodes.

Search - Same as searching a binary search tree.

Inserting :

Same process

1. The only concern is to determine to which colour the node must be set

- If the node is set to black :

Then the path from the root node to external node passing through the node would have one more black node.

∴ [Black Condition] - violated

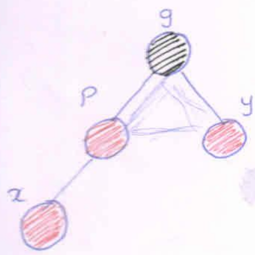
Alternative

- Set the node to red :

Violations in the Red-Black Property:
Rotations needed.

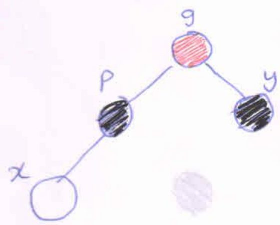
Case 1:

- If uncle node is RED
  change colour of
  - grand parent
  - parent
  - uncle

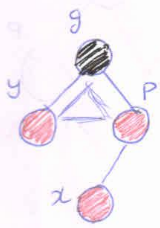


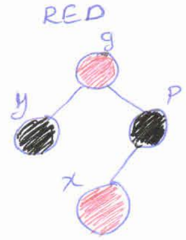
The newly inserted node is a red node - caused violation to RED condition: No 2 red nodes should come consequtively.

⇓ change colour



The newly insexted node creates violation
∴ Uncle node is RED



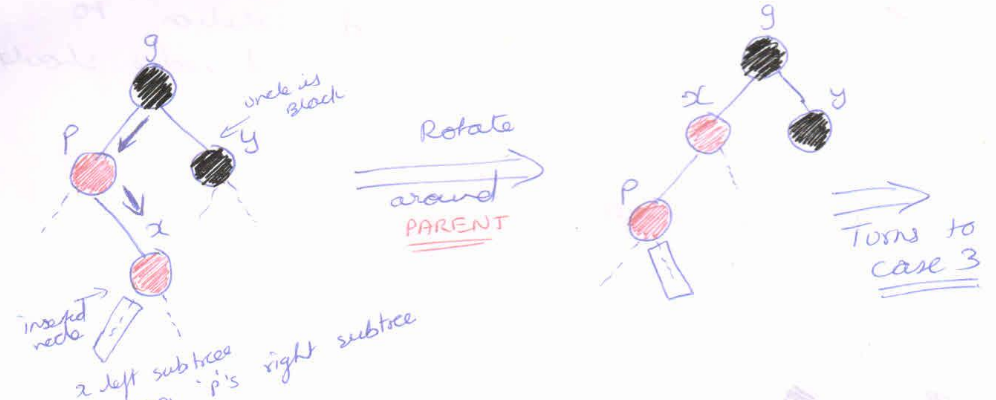


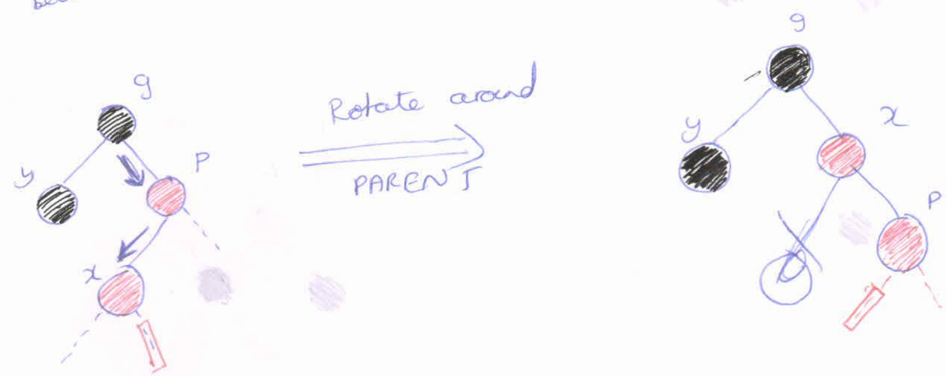
1

Case 2: The violating node is
→ Right child of left child
(OR)
→ Left child of Right child
} Inner circle

~~THEN~~ &
Uncle is BLACK

Rotate around PARENT node

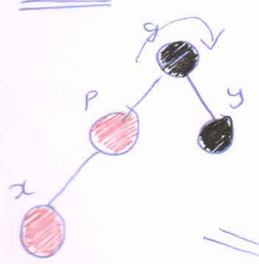




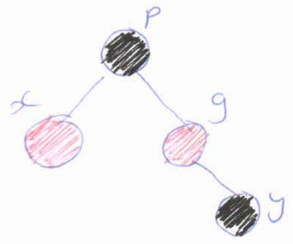
Case 3:

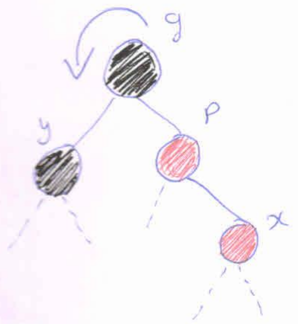
Uncle is Black and violating node is on
LEFT child of LEFT child
OR
RIGHT child of RIGHT child
} outside case

ROTATE Around grandparent
CHANGE colour of parent & grandparent

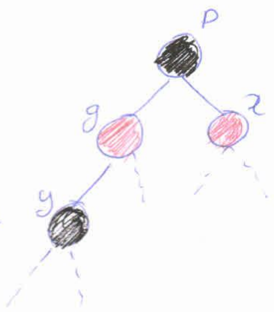


rotate around grand parent change colour of parent & grandparent

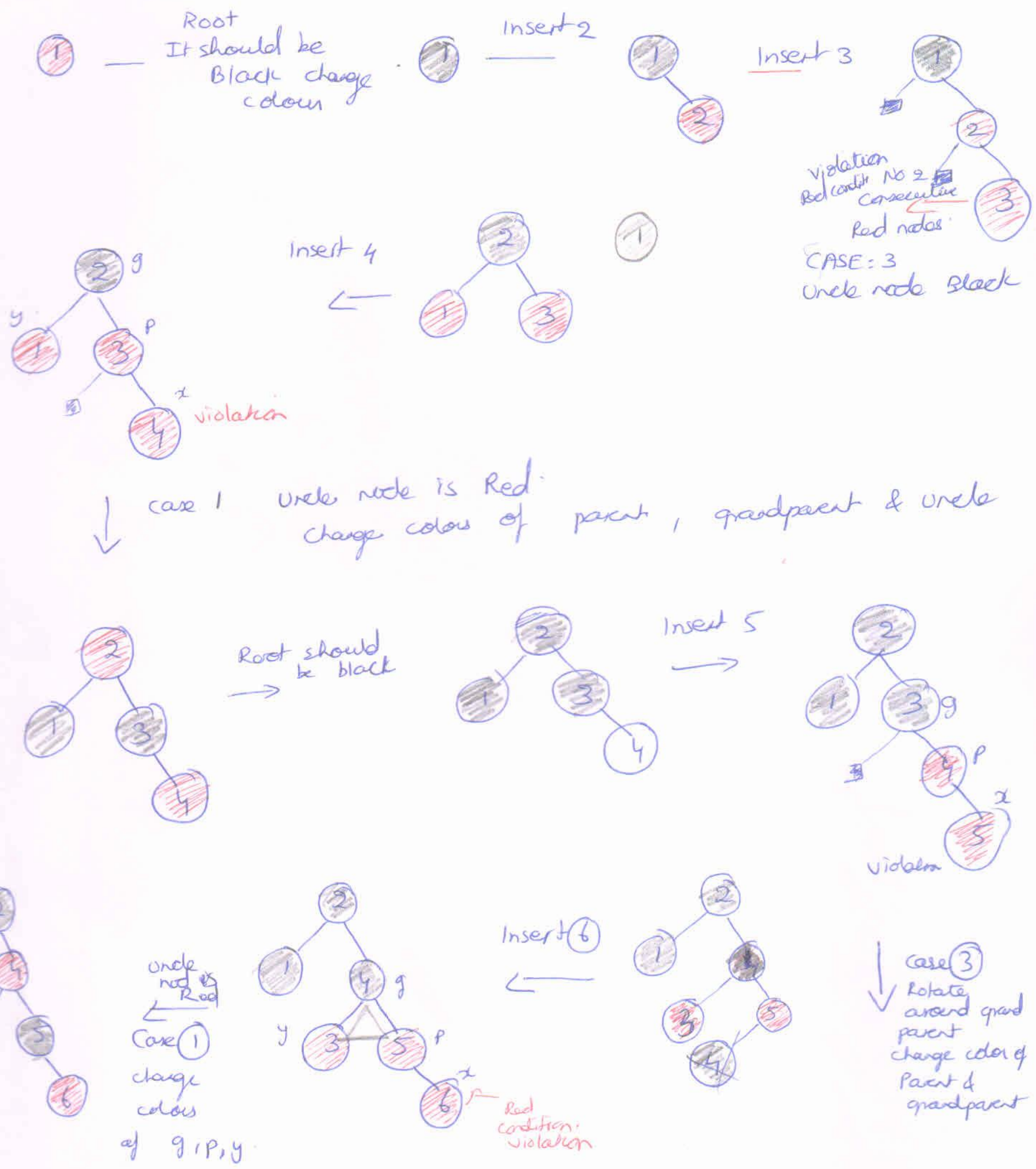


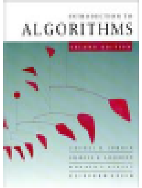


rotate around grand parent and change colour of parent and grandparent



Construct a RED-BLACK tree using
1, 2, 3, 4, 5, 6.





Pseudocode

```
RB-INSERT(T, x)
    TREE-INSERT(T, x)
    color[x] ← RED      ▷ only RB property 3 can be violated
    while x ≠ root[T] and color[p[x]] = RED
        do if p[x] = left[p[p[x]]]
            then y ← right[p[p[x]]]      ▷ y = aunt/uncle of x
                 if color[y] = RED
                  then ⟨Case 1⟩
                  else  if x = right[p[x]]
                          then ⟨Case 2⟩   ▷ Case 2 falls into Case 3
                        ⟨Case 3⟩
            else ⟨"then" clause with "left" and "right" swapped⟩
    color[root[T]] ← BLACK
```

<u>B-Tree:</u>

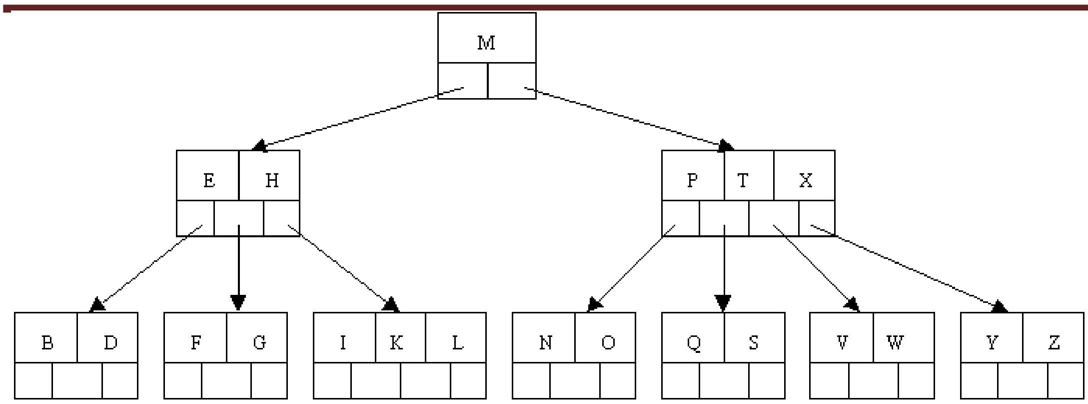
B-Tree is a self balancing search tree.

B-Tree is a tree data structure that keeps data sorted and allows searches, sequential access, insertions and deletions in logarithmic time (O(log n)).

Properties of B-Tree:

- The root has at least one key.
- All leaves (external node) are at the same level.
- Keys are stored in non-decreasing order.
- A B-tree of order M is a tree then
- The root is either a leaf or has between 2 and m Childs.
- Non-root nodes have at least $\lceil M/2 \rceil$ sub-trees.

Example:

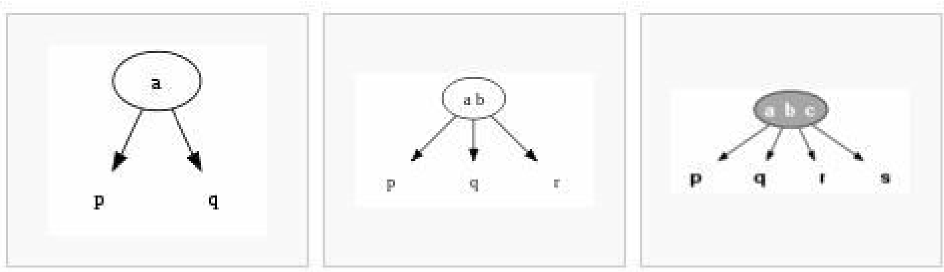


2-3-4 tree:

A B-tree of order 4 is known as a 2-3-4 tree.

A **2–3–4 tree** (also called a **2–4 tree**) is a self-balancing data structure that is commonly used to implement dictionaries. The numbers mean a tree where every node with children (internal node) has either two, three, or four child nodes:

- a 2-node has one data element, and if internal has two child nodes;
- a 3-node has two data elements, and if internal has three child nodes;
- a 4-node has three data elements, and if internal has four child nodes.

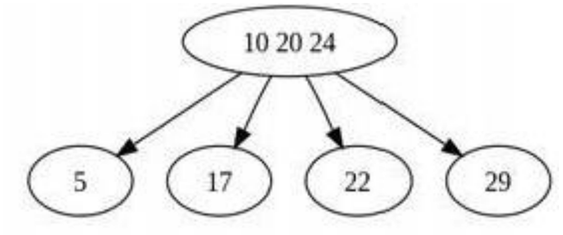


2-node 3-node 4-node

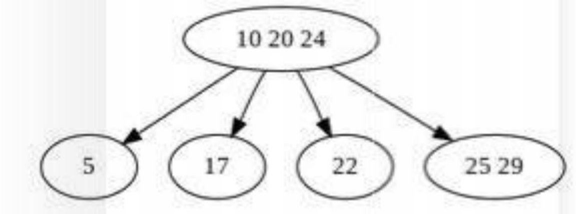
Properties

- Every node (leaf or internal) is a 2-node, 3-node or a 4-node, and holds one, two, or three data elements, respectively.
- All leaves are at the same depth (the bottom level).
- All data is kept in sorted order.

Example:



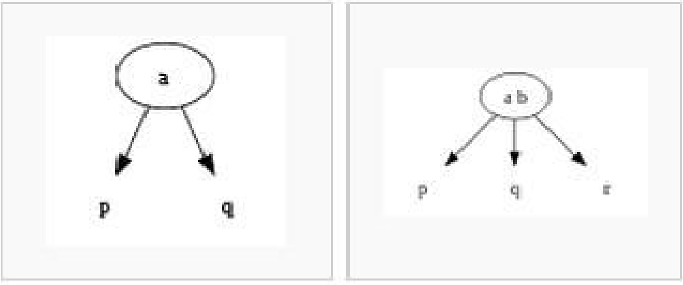
Insert 25



2-3-Tree:

A b-tree of order 3 is known as 2-3-tree.

A 2-3 tree is a tree (B-tree) in which each internal node (non leaf) has either 2 or 3 children and all leaves are at the same level.

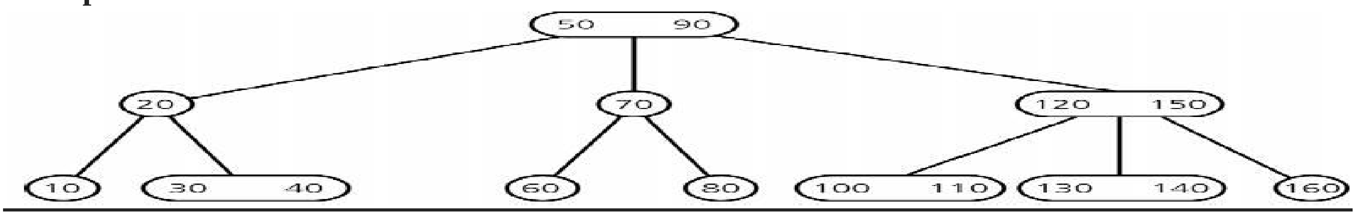


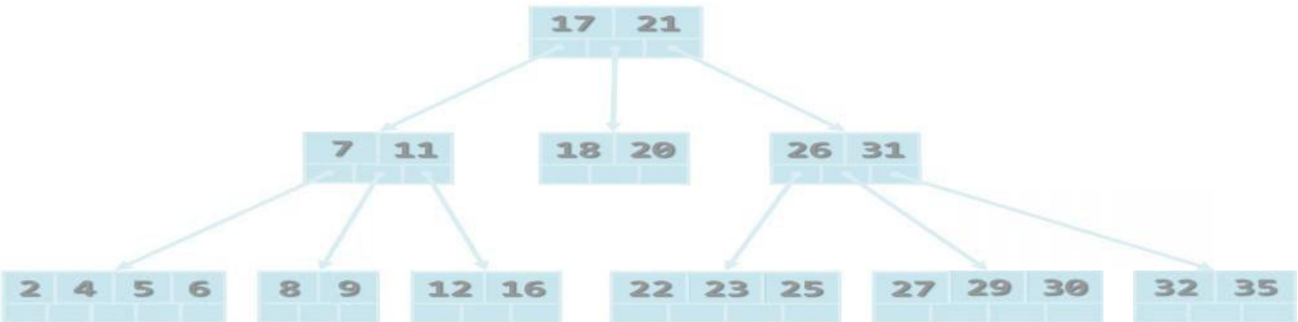
2 node 3 node

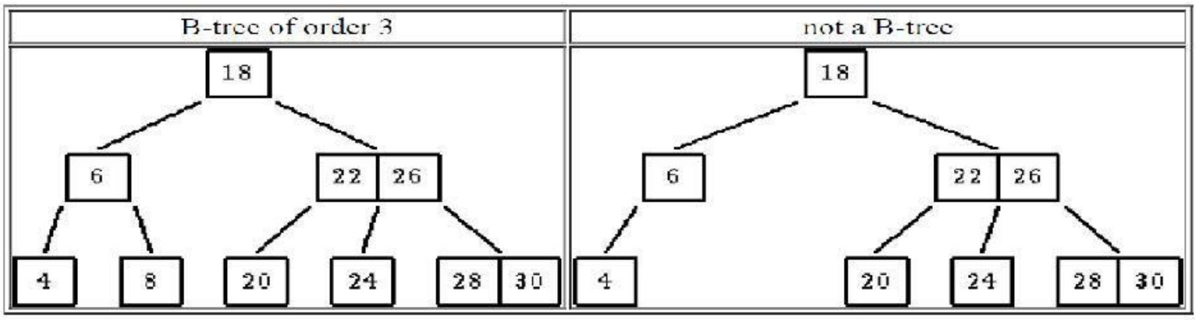
Properties of 2-3-tree:

- Every non-leaf is a 2-node or a 3-node. A 2-node contains one data item and has two children. A 3-node contains two data items and has 3 children.
- All leaves are at the same level (the bottom level)
- All data is kept in sorted order
- Every leaf node will contain 1 or 2 fields.

Example:







Operations on a 2-3 Tree:

The lookup operation (Search)

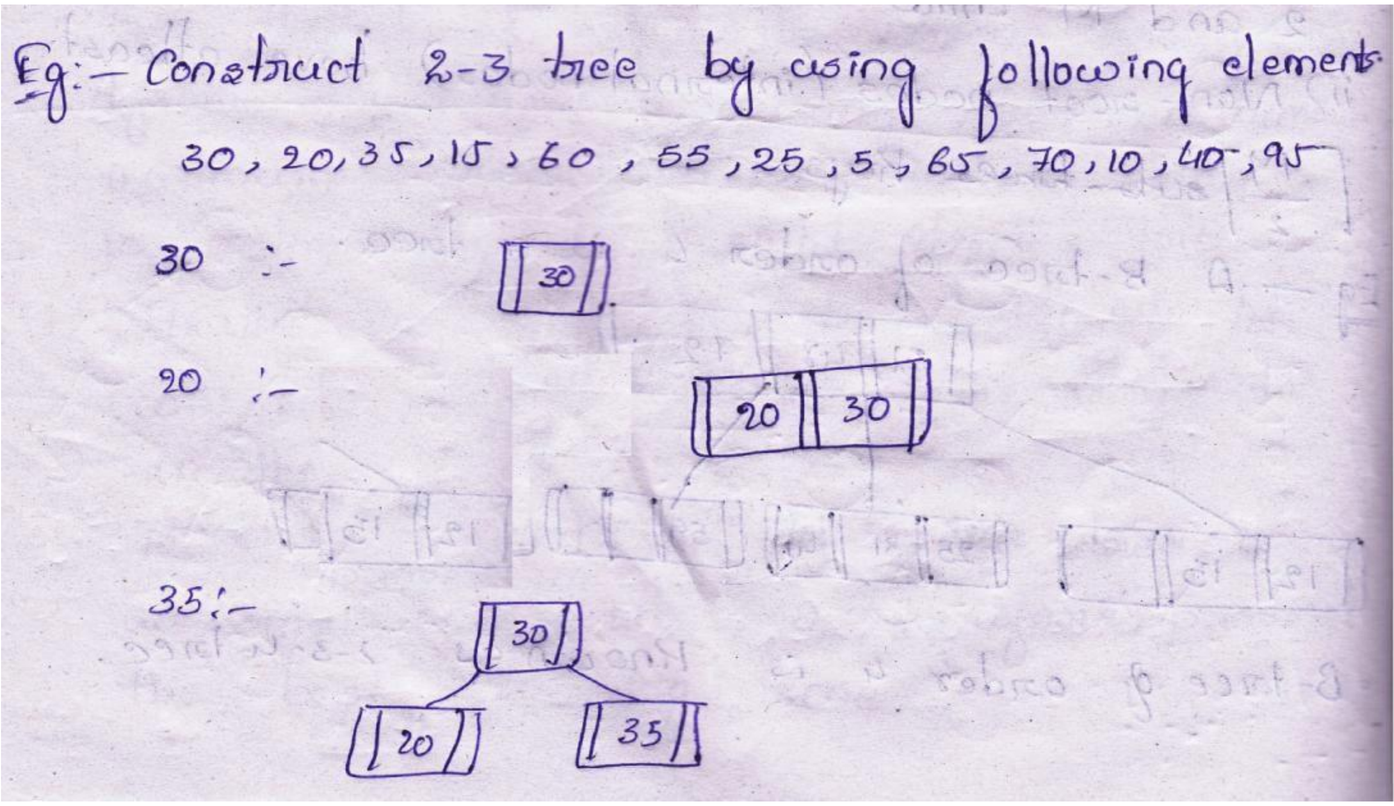
Recall that the lookup operation needs to determine whether key value k is in a 2-3 tree T. The lookup operation for a 2-3 tree is very similar to the lookup operation for a binary-search tree. There are 2 base cases:

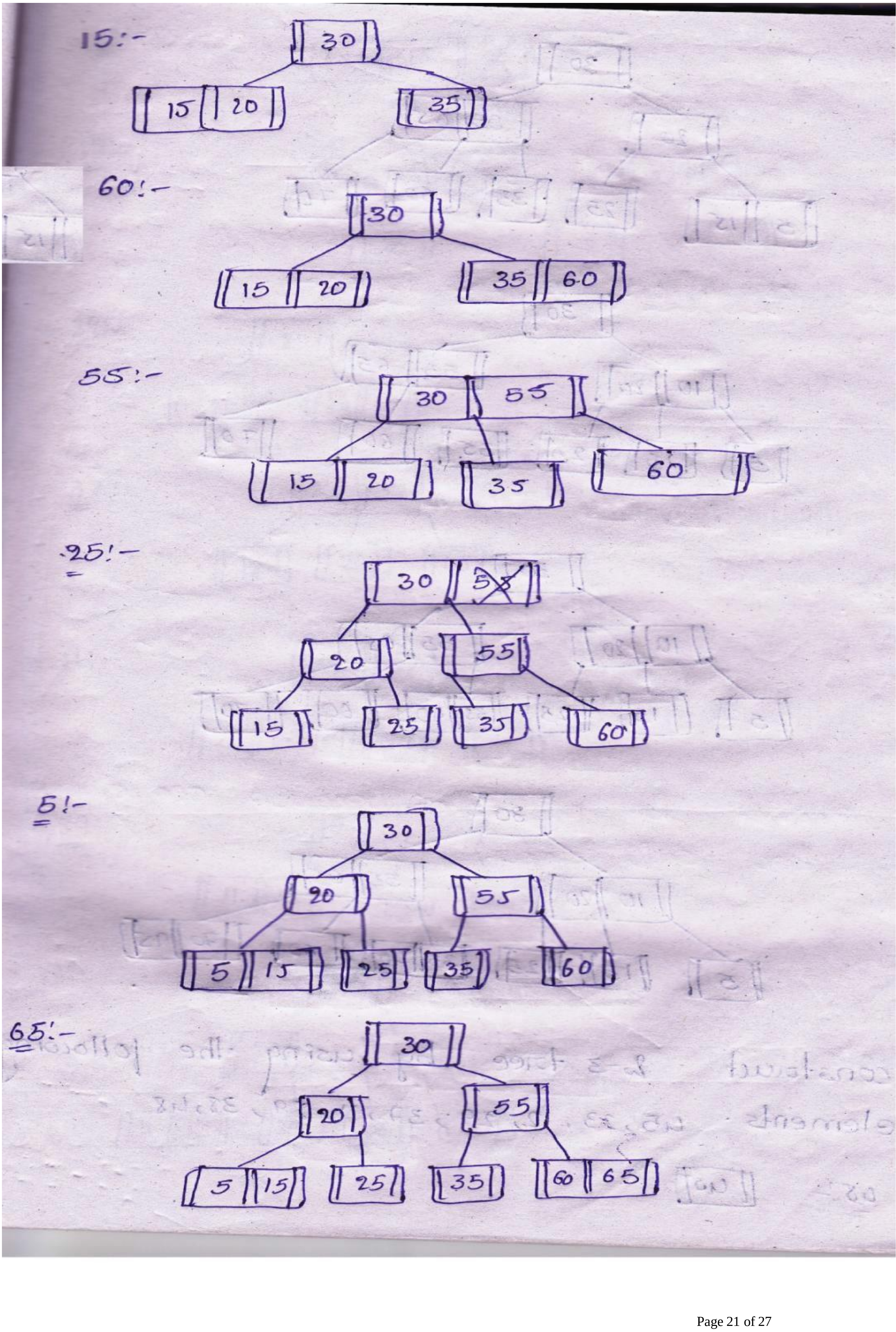
1. T is empty: return false
2. T is a leaf node: return true iff the key value in T is k

And there are 3 recursive cases:

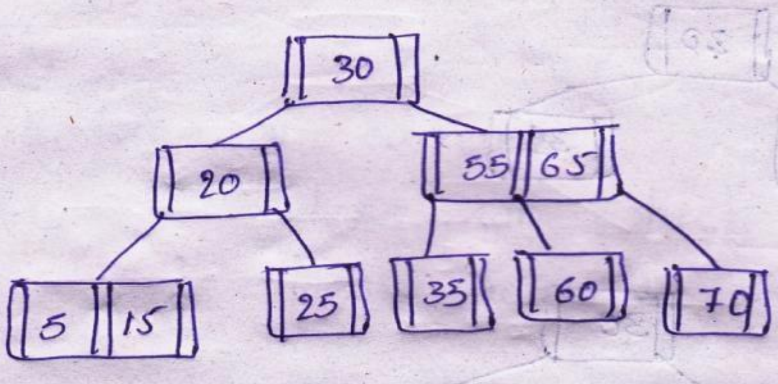
1. k <= T.leftMax: look up k in T's left subtree
2. T.leftMax < k <= T.middleMax: look up k in T's middle subtree
3. T.middleMax < k: look up k in T's right subtree

Constructing 2-3-tree:

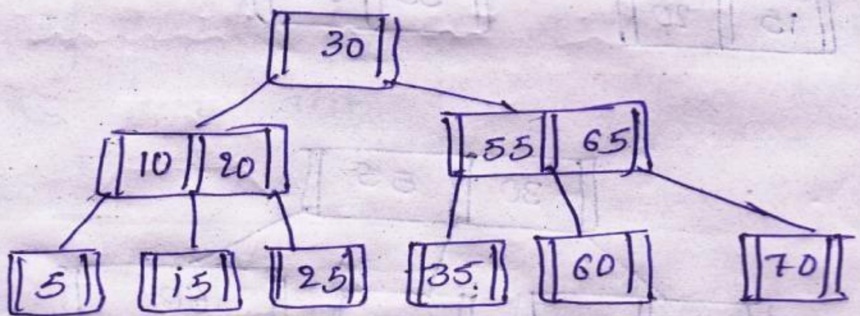



15:-
30
15 20
35
60:-
30
15 20
35 60
55:-
30 55
15 20
35
60
25:-
30 55
20
55
15
25
35
60
5:-
30
20
55
5 15
25
35
60
65:-
30
20
55
5 15
25
35
60 65

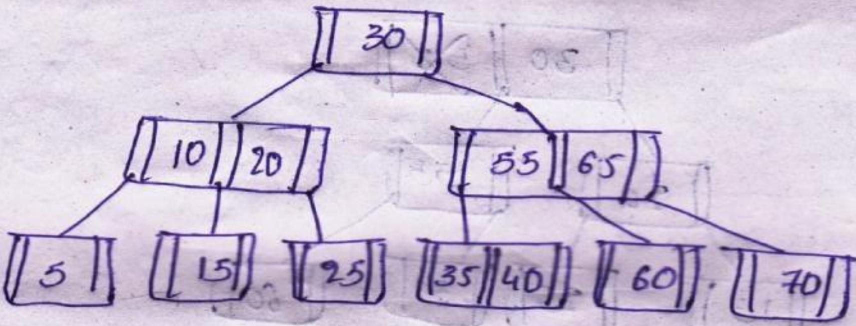
70:-



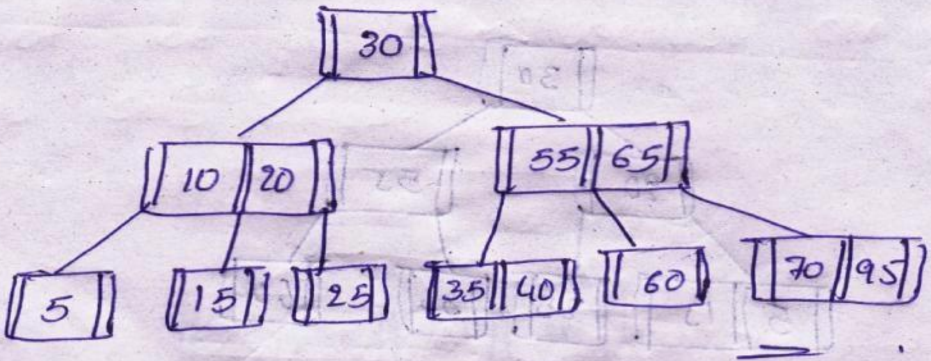
10:-



40:-

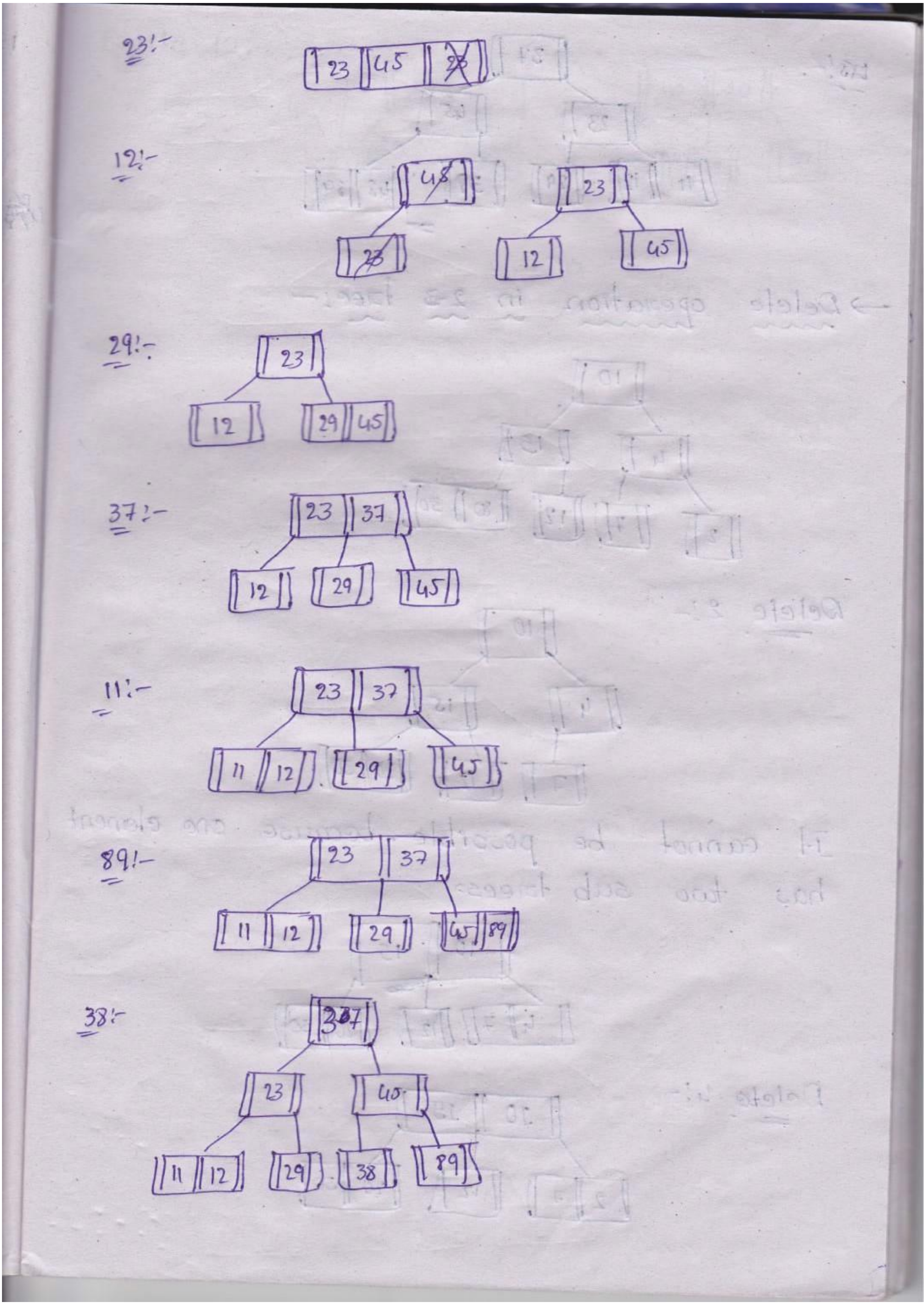


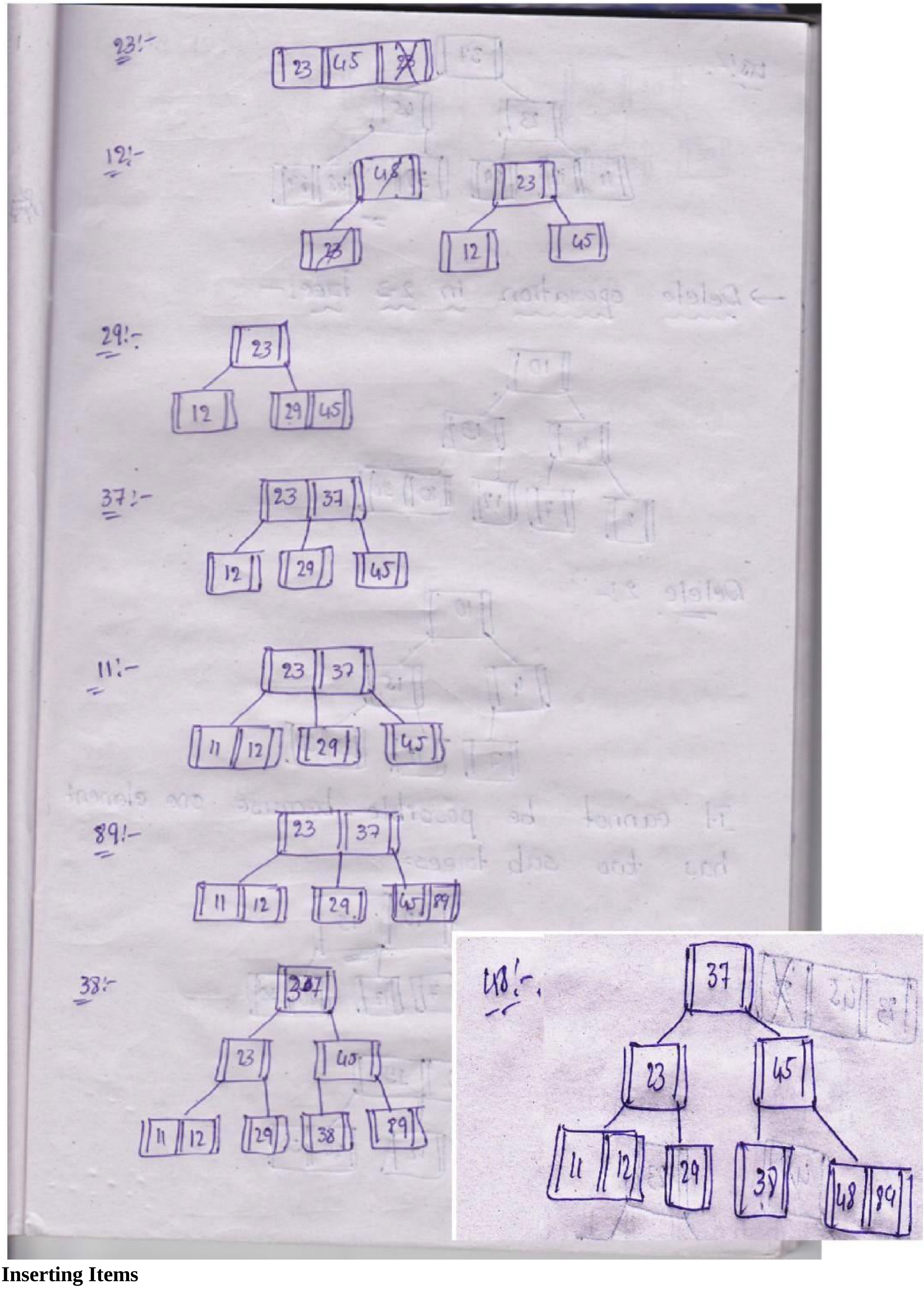
95:-



Eg:- construct 2-3 tree by using the following elements: 45, 23, 12, 29, 37, 11, 89, 38, 48

Sol:- 45:- [45]

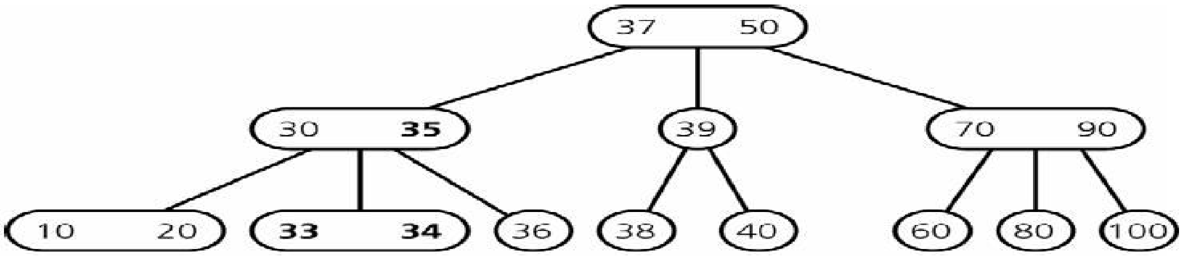

23:-
23 45
12:-
23
12
45
29:-
23
12
29 45
37:-
23 37
12
29
45
11:-
23 37
11 12
29
45
89:-
23 37
11 12
29
45 89
38:-
37
23
45
11 12
29
38
89



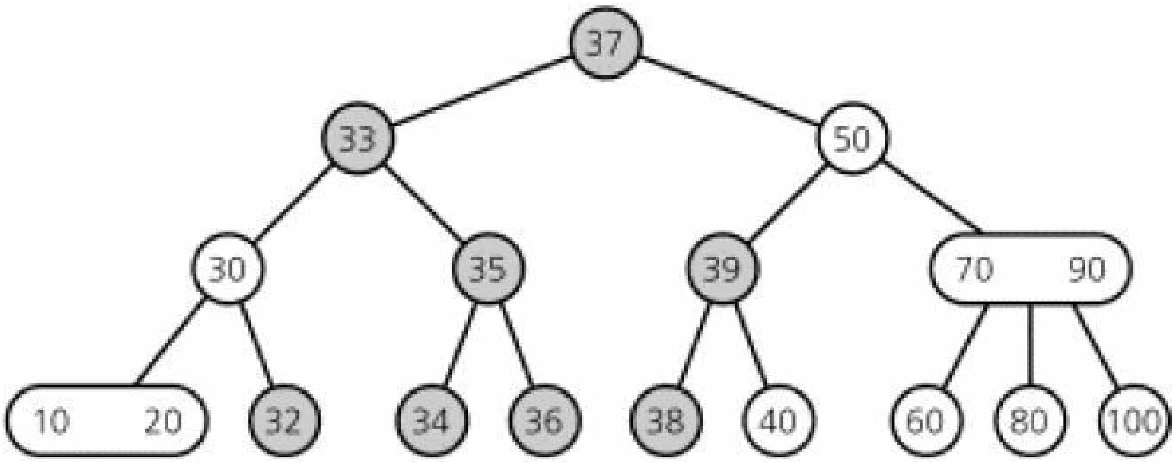
Inserting Items

The goal of the insert operation is to insert key k into tree T, maintaining T's 2-3 tree properties. Special cases are required for empty trees and for trees with just a single (leaf) node.

How do we insert32?



Final Result



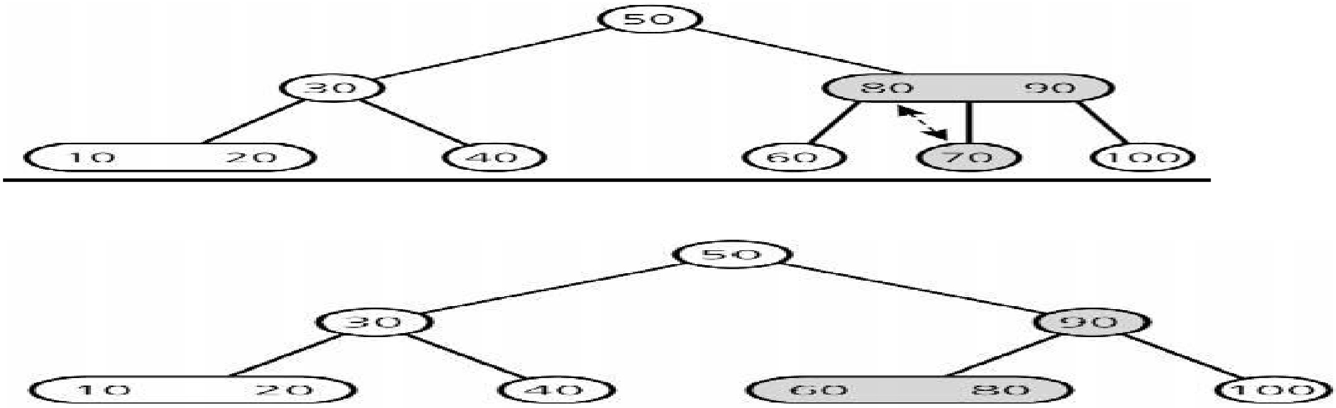
Deleting Items

After deleting an element in the tree (2-3-tree), the resulting tree must be 2-3-tree, means it must the resulting tree must satisfy all the properties of B-tree of order 3.

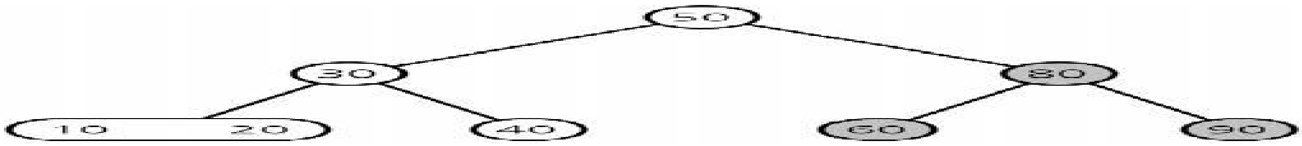
Deleting key k is similar to inserting: there is a special case when T is just a single (leaf) node containing k (T is made empty); otherwise, the parent of the node to be deleted is found, then the tree is fixed up if necessary so that it is still a 2-3 tree.

Consider following example for deleting nodes form 2-3-tree.

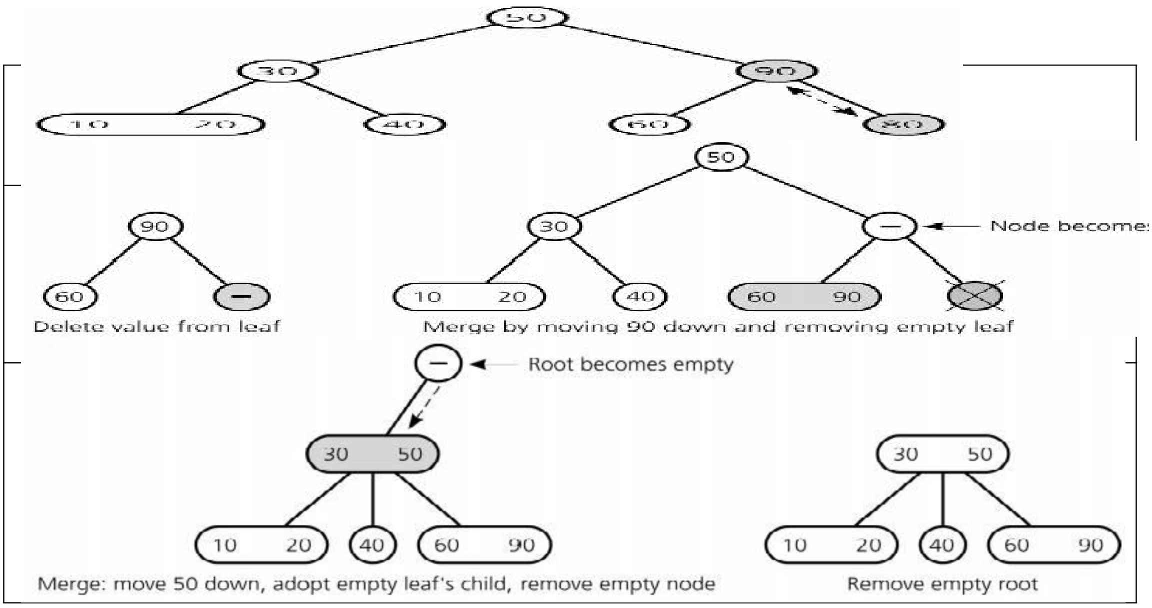
Deleting70:



<u>Delete 100</u>



Delete 80:



2-3 Trees

- These are _not_ binary search trees ....
- Because they are not necessarily _binary_
- They maintain all _leaves_ at _same depth_
    - But number of children can vary
    - 2-3 tree: 2 or 3 children
    - 2-3-4 tree: 2, 3, or 4 children
    - B-tree: B/2 to B children (roughly)

2-3 Trees

- 2-3 tree named for # of possible children of each node
- Each node designated as either 2-node or 3-node
- A 2-node is the same as a binary search tree node

- A 3-node contains *two data* fields, first < second,
- and references to three children:
  - First holds values < first data field
  - Second holds values between the two data fields
  - Third holds values > second data field
- All of the leaves are at the (same) lowest level

Searching a 2-3 Tree

1. if r is null, return null (not in tree)
2. if r is a 2-node
3. if item equals data1, return data1
4. if item < data1, search left subtree
5. else search right subtree
6. else // r is a 3-node
7. if item < data1, search left subtree
8. if item = data1, return data1
9. if item < data2, search middle subtree
10. if item = data2, return data 2
11. else search right subtree

Inserting into a 2-3 Tree (3)

FIGURE 11.35
Inserting into a Tree with All 2-Nodes

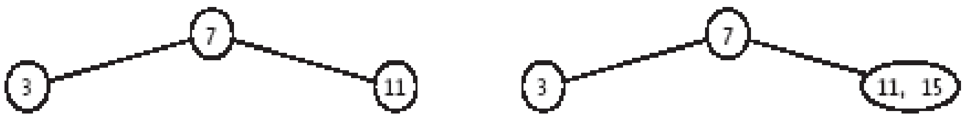


FIGURE 11.36
A Virtual Insertion

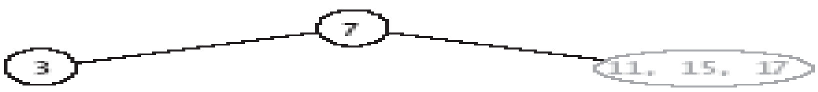


FIGURE 11.37
Result of Propagating 15 to 2-Node Parent

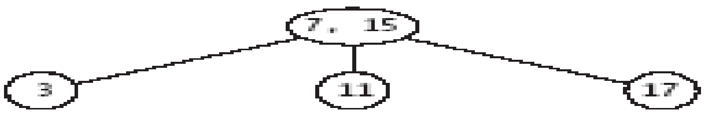


FIGURE 11.38
Inserting 5, 10, and 20

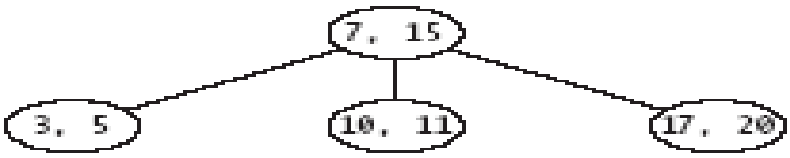


FIGURE 11.39
Virtually Inserting 13

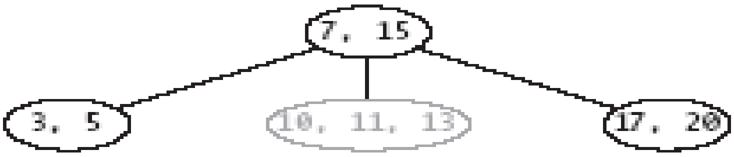


FIGURE 11.40
Virtually Inserting 11

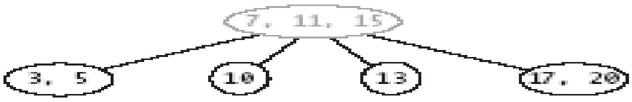
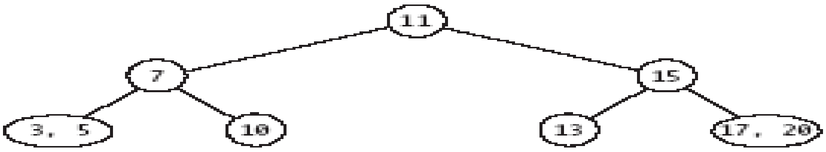


FIGURE 11.41
Result of Making 11 the New Root



- Inserting into 3-node with 3-node parent:
  - “Overload” parent, and repeat process higher up:

FIGURE 11.38
Inserting 5, 10, and 20

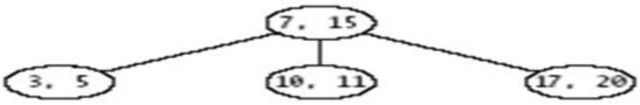


FIGURE 11.39
Virtually Inserting 13

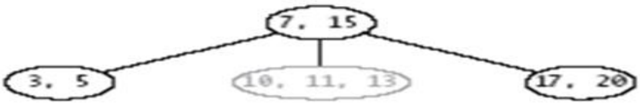


FIGURE 11.40
Virtually Inserting 11

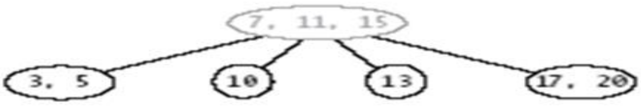
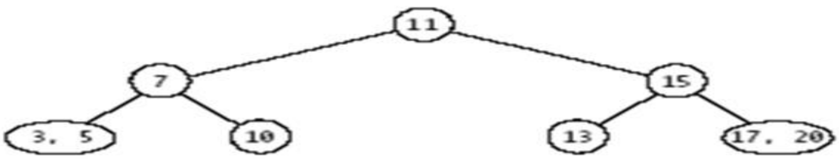


FIGURE 11.41
Result of Making 11 the New Root



Insert Algorithm for 2-3 Tree

1. if r is null, return new 2-node with item as data
2. if item matches r.data1 or r.data2, return **false**
3. if r is a leaf
4. if r is a 2-node, expand to 3-node and return it
5. split into two 2-nodes and pass them back up
6. else
7. recursively insert into appropriate child tree
8. if new parent passed back up
9. if will be tree root, create and use new 2-node

10. else recursively insert parent in r

11. return **true**

2-3 Tree Performance

- If height is h, number of nodes in range 2^h-1 to 3^h-1
- height in terms of # nodes n in range $\log_2 n$ to $\log_3 n$
- This is O(log n), since log base affects by constant factor
- So all operations are O(log n)

Removal from a 2-3 Tree

- Removing from a 2-3 tree is the reverse of insertion
- If the item in a leaf, simply delete it
- If not in a leaf
    - Swap it with its inorder predecessor in a leaf
    - Then delete it from the leaf node

Redistribute nodes between siblings and parent

FIGURE 11.42
Removing 13 from a 2-3 Tree

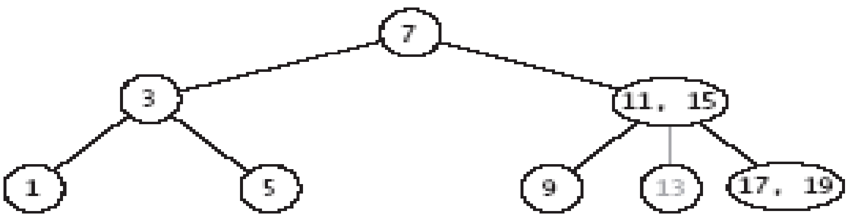


FIGURE 11.43
2-3 Tree After Redistribution of Nodes Resulting from Removal

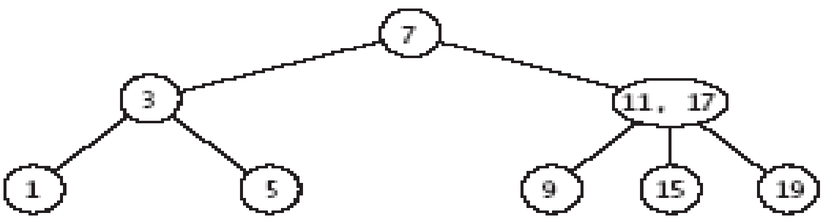


FIGURE 11.44
Removing 11 from the 2-3 Tree (Step 1)

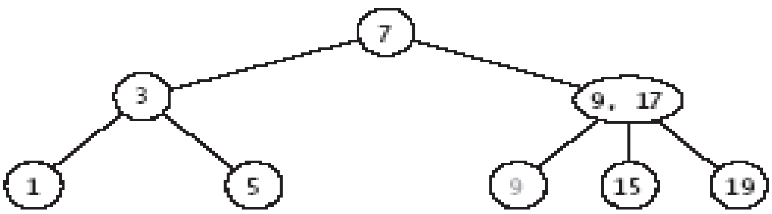


FIGURE 11.45
2-3 Tree After Removing 11

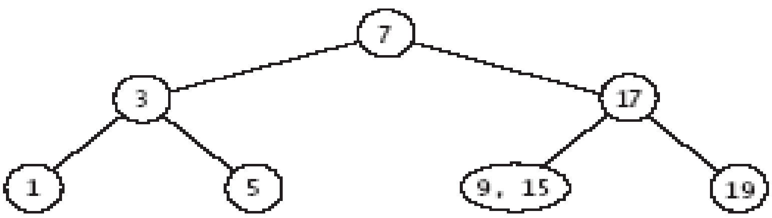


FIGURE 11.46
After Removing 1
(Intermediate Step)

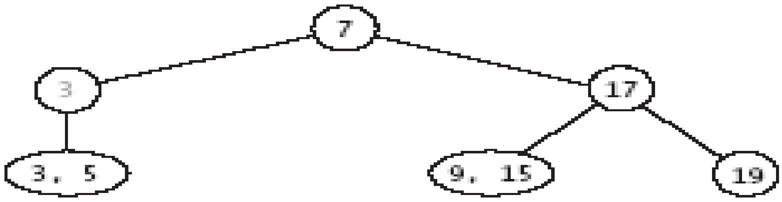
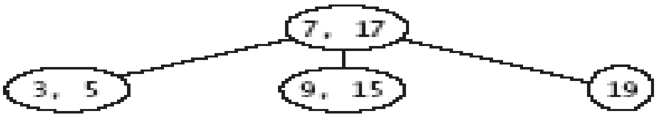


FIGURE 11.47
After Removing 1
(Final Form)



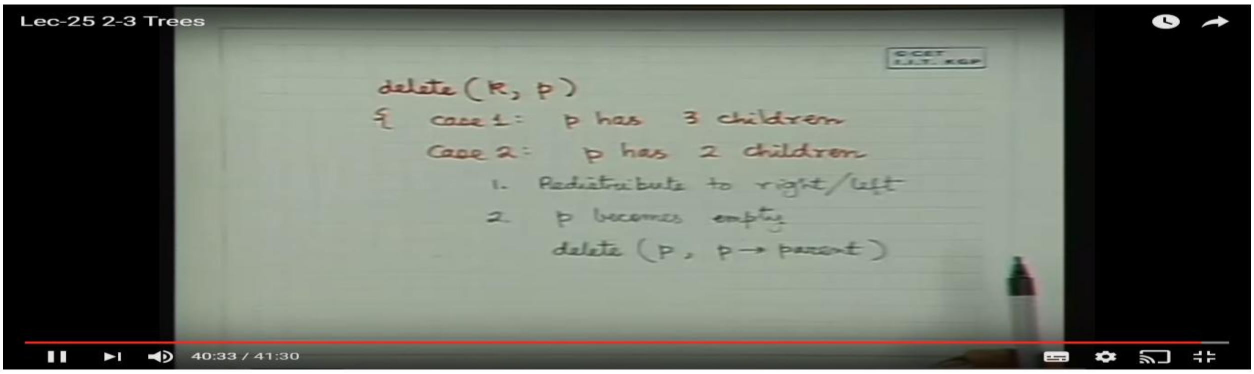
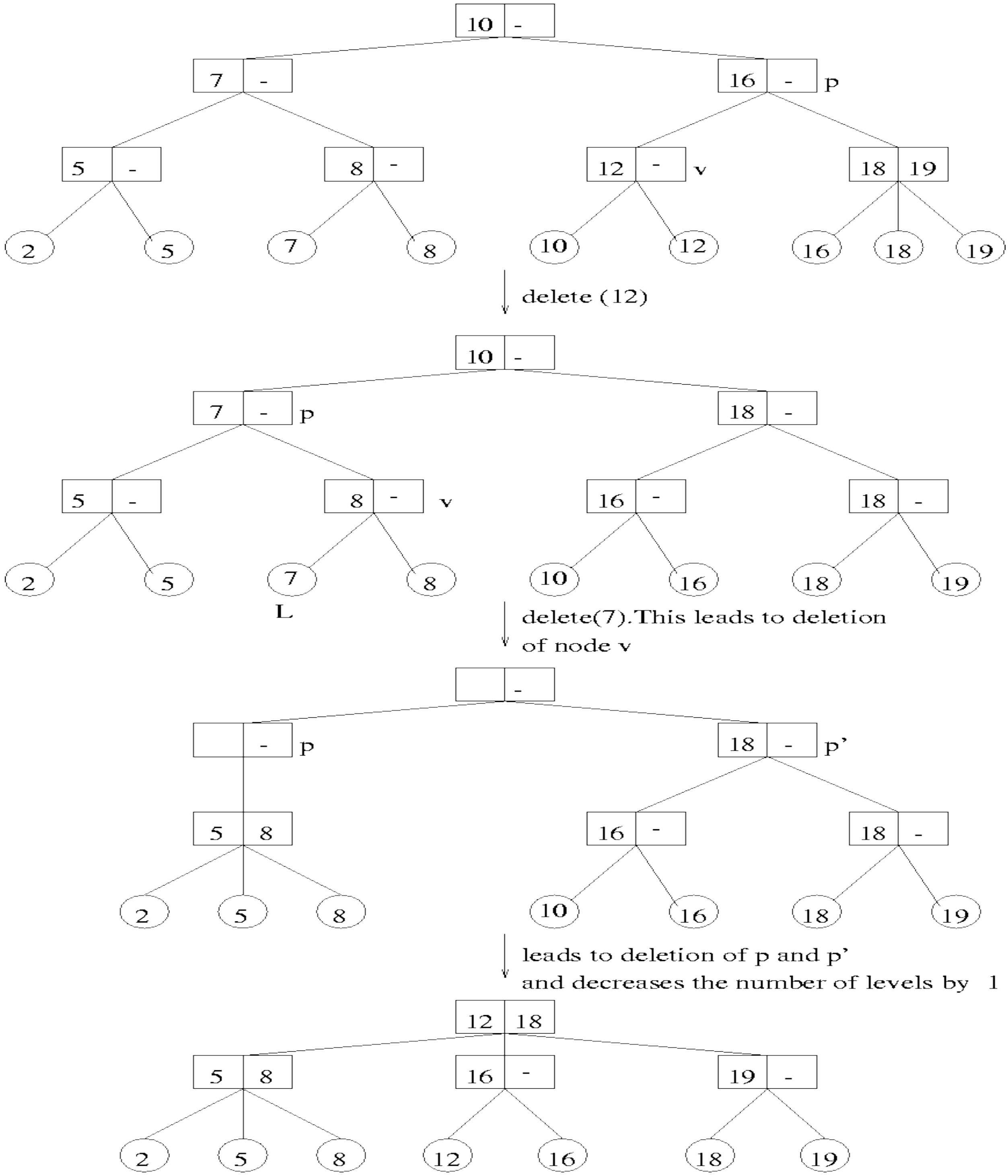
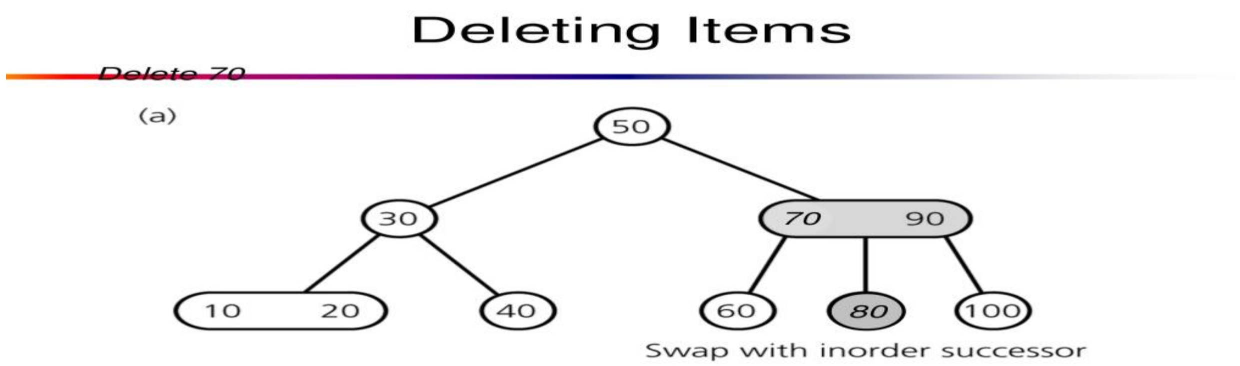


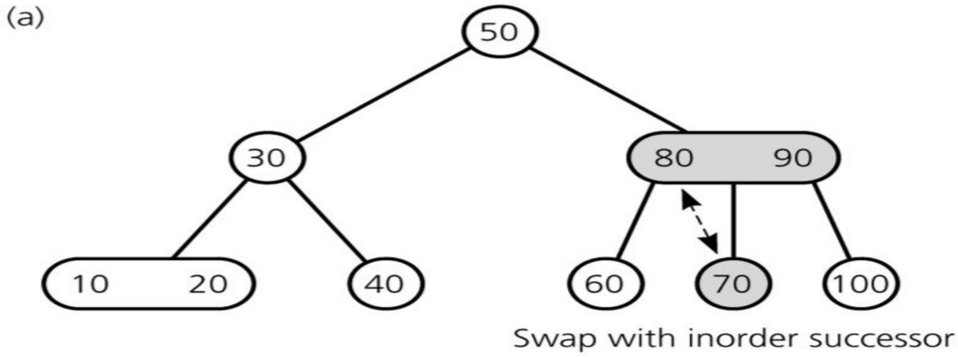
Figure 5.25: Deletion in 2-3 trees: An Example


delete (12)
delete(7).This leads to deletion of node v
leads to deletion of p and p'
and decreases the number of levels by 1


Deleting Items
Delete 70
(a)
Swap with inorder successor

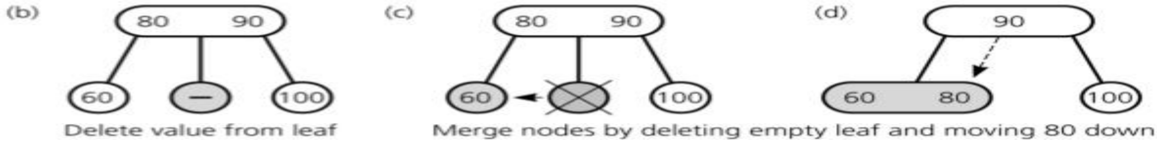
Deleting Items

Deleting 70: swap 70 with inorder successor (80)



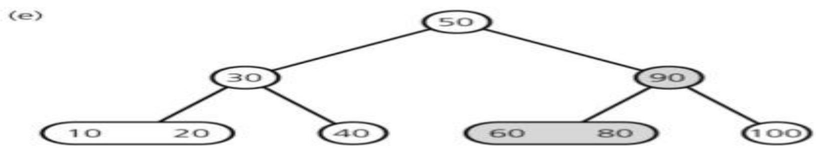
Deleting Items

Deleting 70: ... get rid of 70



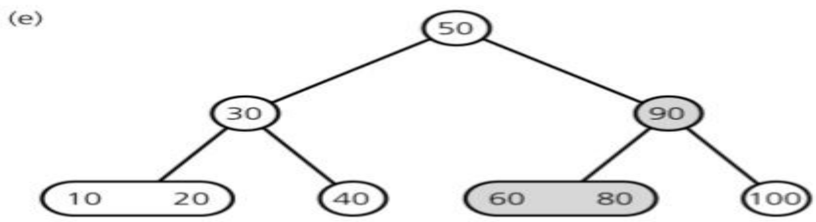
Deleting Items

Result



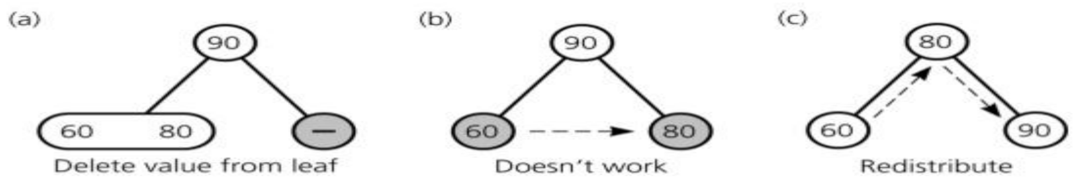
Deleting Items

Delete 100



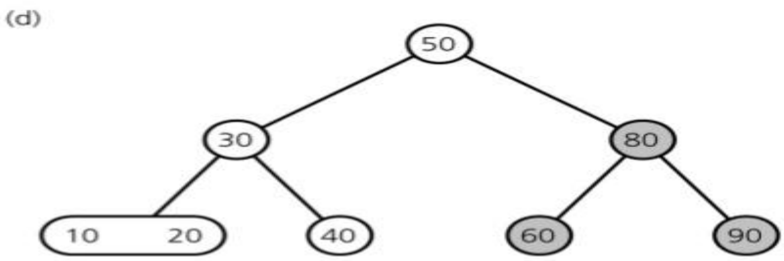
Deleting Items

Deleting 100



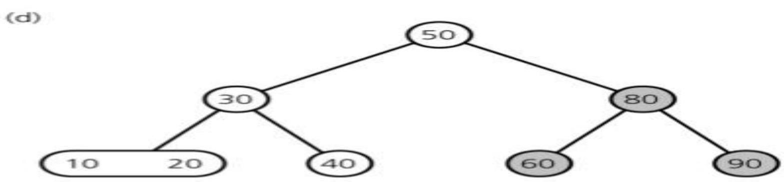
Deleting Items

Result



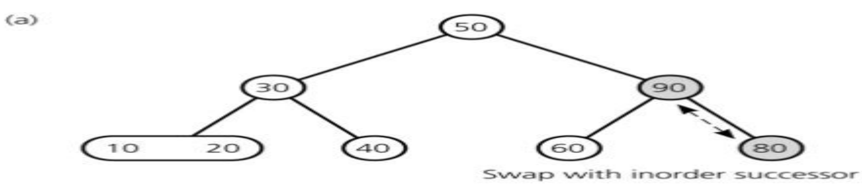
Deleting Items

Delete 80



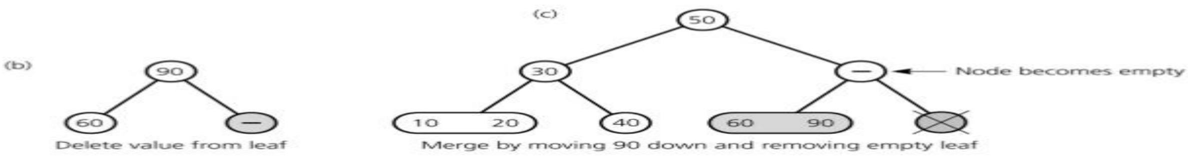
Deleting Items

Deleting 80 ...



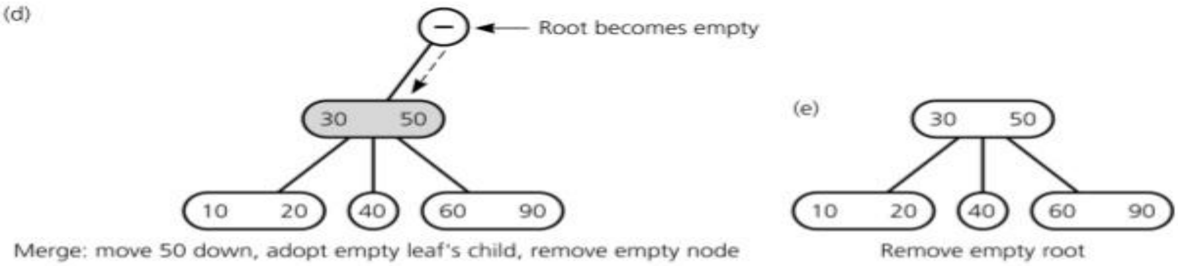
Deleting Items

Deleting 80 ...



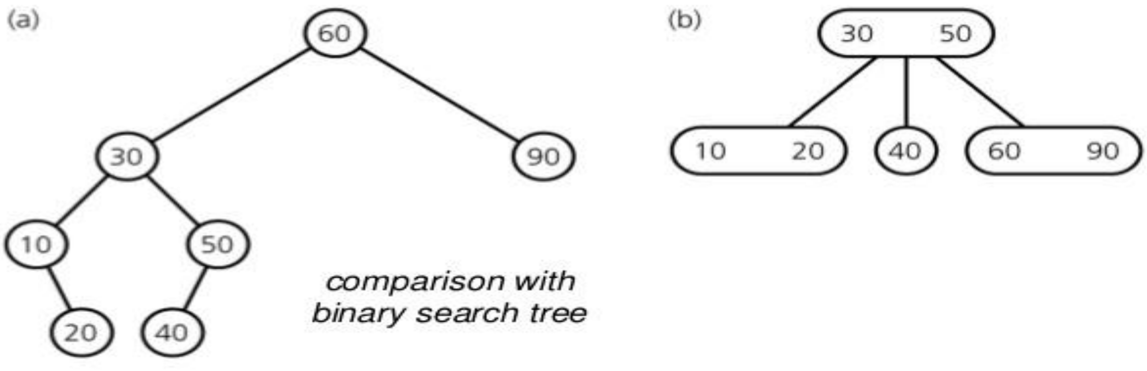
Deleting Items

Deleting 80 ...



Deleting Items

Final Result



Deletion Algorithm I

Deleting item I:

1. *Locate node n, which contains item I*
2. *If node n is not a leaf → swap I with inorder successor*

→ *deletion always begins at a leaf*

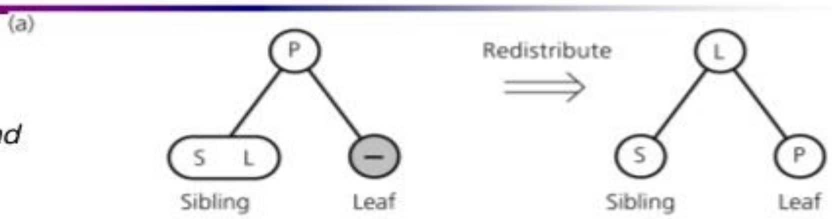
3. *If leaf node n contains another item, just delete item I*
   else
   try to redistribute nodes from siblings (see next slide)
   if not possible, merge node (see next slide)

Deletion Algorithm II

Redistribution

A sibling has 2 items:

→ *redistribute item between siblings and parent*

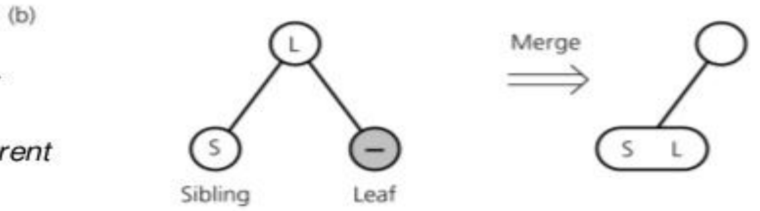


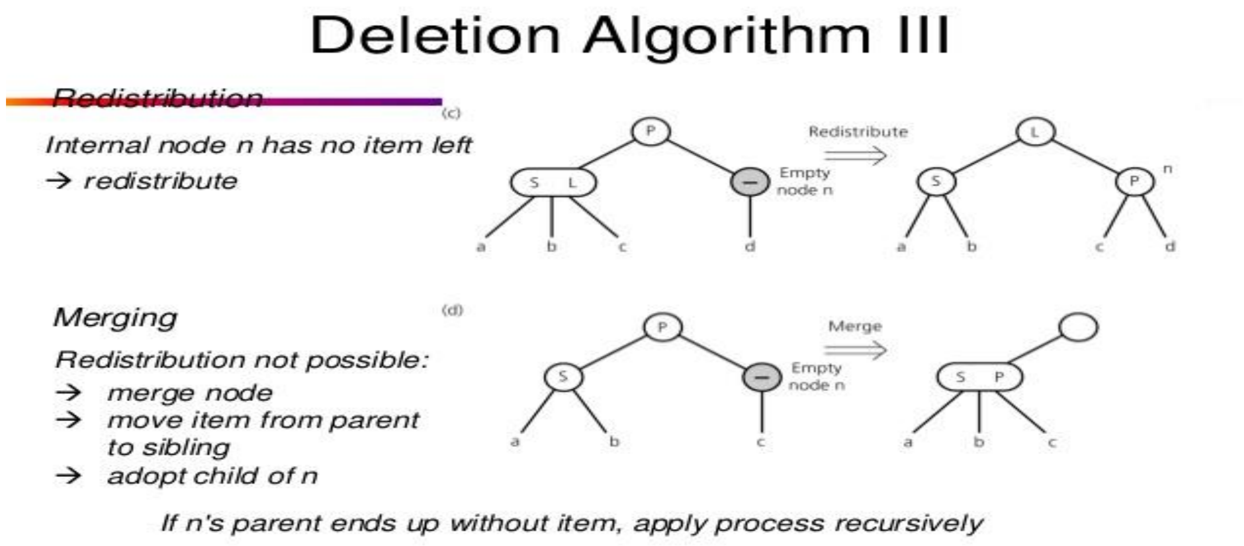
Merging

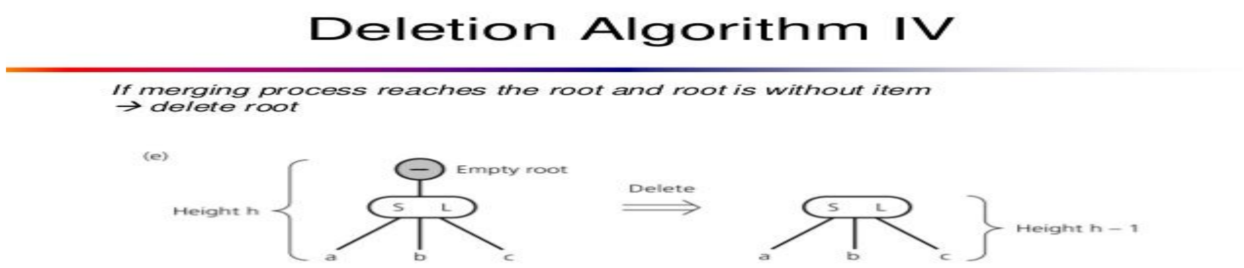
No sibling has 2 items:

→ *merge node*

→ *move item from parent to sibling*



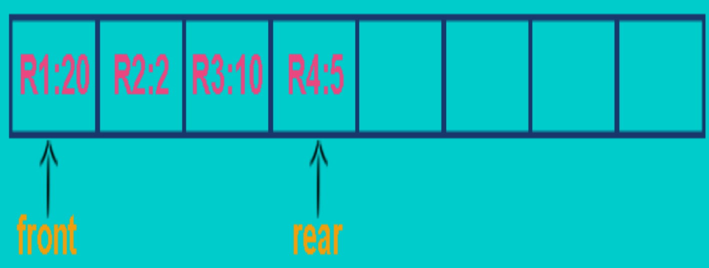




Priority Queue

In normal queue data structure, insertion is performed at the end of the queue and deletion is performed based on the FIFO principle. This queue implementation may not be suitable for all situations.

Consider a networking application where server has to respond for requests from multiple clients using queue data structure. Assume four requests arrived to the queue in the order of R1 requires 20 units of time, R2 requires 2 units of time, R3 requires 10 units of time and R4 requires 5 units of time. Queue is as follows...



Now, check waiting time for each request to be complete.

1. **R1 : 20 units of time**
2. **R2 : 22 units of time (R2 must wait till R1 complete - 20 units and R2 itself requeres 2 units. Total 22 units)**
3. **R3 : 32 units of time (R3 must wait till R2 complete - 22 units and R3 itself requeres 10 units. Total 32 units)**
4. **R4 : 37 units of time (R4 must wait till R3 complete - 35 units and R4 itself requeres 5 units. Total 37 units)**

Here, average waiting time for all requests (R1, R2, R3 and R4) is (20+22+32+37)/4 ≈ 27 units of time.

That means, if we use a normal queue data structure to serve these requests the average waiting time for each request is 27 units of time.

Now, consider another way of serving these requests. If we serve according to their required amount of time. That means, first we serve R2 which has minimum time required (2) then serve R4 which has second minimum time required (5) then serve R3 which has third minimum time required (10) and finnaly R1 which has maximum time required (20).

Now, check waiting time for each request to be complete.

1. **R2 : 2 units of time**
2. **R4 : 7 units of time (R4 must wait till R2 complete 2 units and R4 itself requeres 5 units. Total 7 units)**
3. **R3 : 17 units of time (R3 must wait till R4 complete 7 units and R3 itself requeres 10 units. Total 17 units)**
4. **R1 : 37 units of time (R1 must wait till R3 complete 17 units and R1 itself requeres 20 units. Total 37 units)**

Here, average waiting time for all requests (R1, R2, R3 and R4) is (2+7+17+37)/4 ≈ 15 units of time.

From above two situations, it is very clear that, by using second method server can complete all four requests with very less time compared to the first method. This is what exactly done by the priority queue.

Priority queue is a variant of queue data structure in which insertion is performed in the order of arrival and deletion is performed based on the priority.

There are two types of priority queues they are as follows...

1. **Max Priority Queue**
2. **Min Priority Queue**

In max priority queue, elements are inserted in the order in which they arrive the queue and always maximum value is removed first from the queue. For example assume that we insert in order 8, 3, 2, 5 and they are removed in the order 8, 5, 3, 2.

The following are the operations performed in a Max priority queue...

1. **isEmpty() - Check whether queue is Empty.**
2. **insert() - Inserts a new value into the queue.**
3. **findMax() - Find maximum value in the queue.**
4. **remove() - Delete maximum value from the queue.**

Max Priority Queue Representations

There are 6 representations of max priority queue.

1. **Using an Unordered Array (Dynamic Array)**
2. **Using an Unordered Array (Dynamic Array) with the index of the maximum value**
3. **Using an Array (Dynamic Array) in Decreasing Order**
4. **Using an Array (Dynamic Array) in Increasing Order**
5. **Using Linked List in Increasing Order**
6. **Using Unordered Linked List with reference to node with the maximum value**

#1. Using an Unordered Array (Dynamic Array)

In this representation elements are inserted according to their arrival order and maximum element is deleted first from max priority queue.

For example, assume that elements are inserted in the order of 8, 2, 3 and 5. And they are removed in the order 8, 5, 3 and 2.

Now, let us analyse each operation according to this representation...

isEmpty() - If '**front == -1**' queue is Empty. This operation requires O(1) time complexity that means constant time.

insert() - New element is added at the end of the queue. This operation requires O(1) time complexity that means constant time.

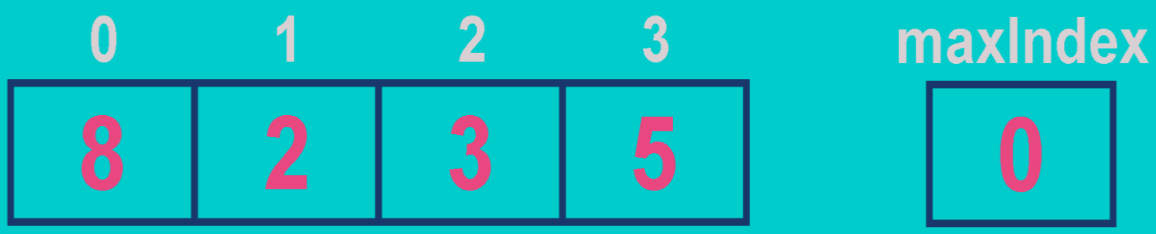
findMax() - To find maximum element in the queue, we need to compare with all the elements in the queue. This operation requires O(n) time complexity.

remove() - To remove an element from the queue first we need to perform **findMax()** which requires **O(n)** and removal of particular element requires constant time **O(1)**. This operation requires O(n) time complexity.

#2. Using an Unordered Array (Dynamic Array) with the index of the maximum value

In this representation elements are inserted according to their arrival order and maximum element is deleted first from max priority queue.

For example, assume that elements are inserted in the order of 8, 2, 3 and 5. And they are removed in the order 8, 5, 3 and 2.



Now, let us analyse each operation according to this representation...

isEmpty() - If '**front == -1**' queue is Empty. This operation requires O(1) time complexity that means constant time.

insert() - New element is added at the end of the queue with **O(1)** and for each insertion we need to update maxIndex with **O(1)**. This operation requires O(1) time complexity that means constant time.

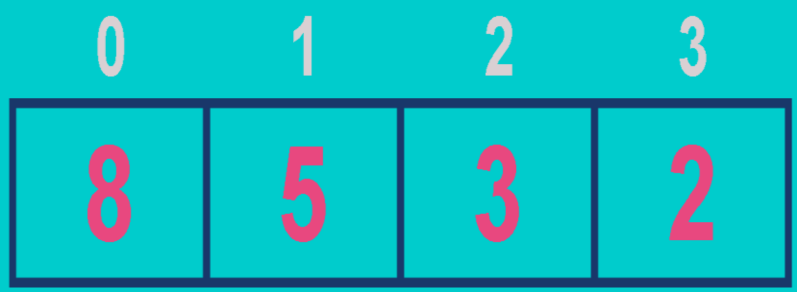
findMax() - To find maximum element in the queue is very simple as maxIndex has maximum element index. This operation requires O(1) time complexity.

remove() - To remove an element from the queue first we need to perform **findMax()** which requires **O(1)** , removal of particular element requires constant time **O(1)** and update maxIndex value which requires **O(n)**. This operation requires O(n) time complexity.

#3. Using an Array (Dynamic Array) in Decreasing Order

In this representation elements are inserted according to their value in decreasing order and maximum element is deleted first from max priority queue.

For example, assume that elements are inserted in the order of 8, 5, 3 and 2. And they are removed in the order 8, 5, 3 and 2.



Now, let us analyse each operation according to this representation...

isEmpty() - If '**front == -1**' queue is Empty. This operation requires O(1) time complexity that means constant time.

insert() - New element is added at a particular position in the decreasing order into the queue with **O(n)**, because we need to shift existing elements inorder to insert new element in decreasing order. This operation requires O(n) time complexity.

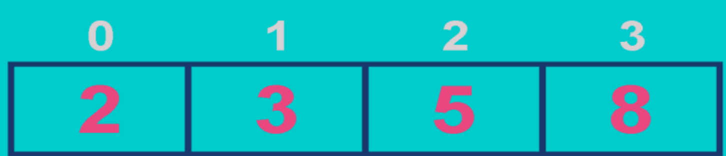
findMax() - To find maximum element in the queue is very simple as maximum element is at the beginning of the queue. This operation requires O(1) time complexity.

remove() - To remove an element from the queue first we need to perform **findMax()** which requires **O(1)**, removal of particular element requires constant time **O(1)** and rearrange remaining elements which requires **O(n)**. This operation requires O(n) time complexity.

#4. Using an Array (Dynamic Array) in Increasing Order

In this representation elements are inserted according to their value in increasing order and maximum element is deleted first from max priority queue.

For example, assume that elements are inserted in the order of 2, 3, 5 and 8. And they are removed in the order 8, 5, 3 and 2.



Now, let us analyse each operation according to this representation...

isEmpty() - If '**front == -1**' queue is Empty. This operation requires **O(1)** time complexity that means constant time.

insert() - New element is added at a particular position in the increasing order into the queue with **O(n)**, because we need to shift existing elements inorder to insert new element in increasing order. This operation requires **O(n)** time complexity.

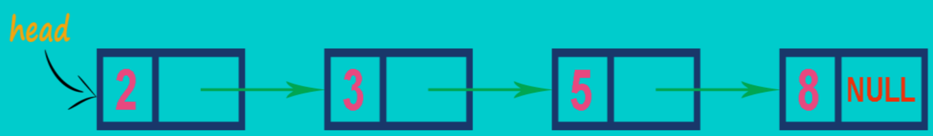
findMax() - To find maximum element in the queue is very simple as maximum element is at the end of the queue. This operation requires **O(1)** time complexity.

remove() - To remove an element from the queue first we need to perform **findMax()** which requires **O(1)**, removal of particular element requires constant time **O(1)** and rearrange remaining elements which requires **O(n)**. This operation requires **O(n)** time complexity.

#5. Using Linked List in Increasing Order

In this representation, we use a single linked list to represent max priority queue. In this representation elements are inserted according to their value in increasing order and node with maximum value is deleted first from max priority queue.

For example, assume that elements are inserted in the order of 2, 3, 5 and 8. And they are removed in the order 8, 5, 3 and 2.



Now, let us analyse each operation according to this representation...

isEmpty() - If '**head == NULL**' queue is Empty. This operation requires **O(1)** time complexity that means constant time.

insert() - New element is added at a particular position in the increasing order into the queue with **O(n)**, because we need to the position where new element has to be inserted. This operation requires **O(n)** time complexity.

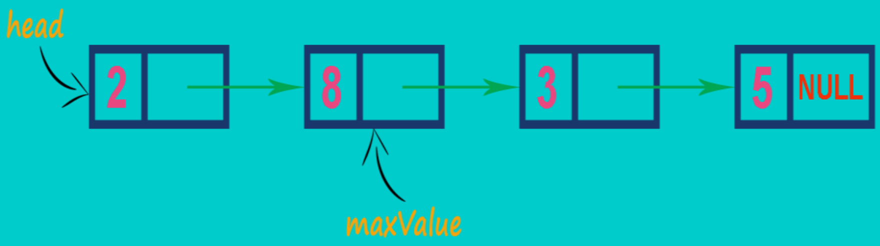
findMax() - To find maximum element in the queue is very simple as maximum element is at the end of the queue. This operation requires **O(1)** time complexity.

remove() - To remove an element from the queue is simply removing the last node in the queue which requires **O(1)**. This operation requires **O(1)** time complexity.

#6. Using Unordered Linked List with reference to node with the maximum value

In this representation, we use a single linked list to represent max priority queue. Always we maitain a reference (maxValue) to the node with maximum value. In this representation elements are inserted according to their arrival and node with maximum value is deleted first from max priority queue.

For example, assume that elements are inserted in the order of 2, 8, 3 and 5. And they are removed in the order 8, 5, 3 and 2.



Now, let us analyse each operation according to this representation...

isEmpty() - If '**head == NULL**' queue is Empty. This operation requires **O(1)** time complexity that means constant time.

insert() - New element is added at end the queue with **O(1)** and update maxValue reference with **O(1)**. This operation requires **O(1)** time complexity.

findMax() - To find maximum element in the queue is very simple as maxValue is referenced to the node with maximum value in the queue. This operation requires **O(1)** time complexity.

remove() - To remove an element from the queue is deleting the node which referenced by maxValue which requires **O(1)** and update maxValue reference to new node with maximum value in the queue which requires **O(n) time complexity**. This operation requires **O(n)** time complexity.

Min Priority Queue is similar to max priority queue except removing maximum element first, we remove minimum element first in min priority queue.

The following operations are performed in Min Priority Queue...

1. **isEmpty()** - Check whether queue is Empty.
2. **insert()** - Inserts a new value into the queue.
3. **findMin()** - Find minimum value in the queue.
4. **remove()** - Delete minimum value from the queue.

Min priority queue is also has same representations as Max priority queue with minimum value removal.

Heap Data Structure

Heap data structure is a specialized binary tree based data structure. Heap is a binary tree with special characteristics. In a heap data structure, nodes are arranged based on thier value. A heap data structure, some time called as Binary Heap.

There are two types of heap data structures and they are as follows...

1. **Max Heap**
2. **Min Heap**

Every heap data structure has the following properties...

Property #1 (Ordering): Nodes must be arranged in a order according to values based on Max heap or Min heap.

Property #2 (Structural): All levels in a heap must full, except last level and nodes must be filled from left to right strictly.

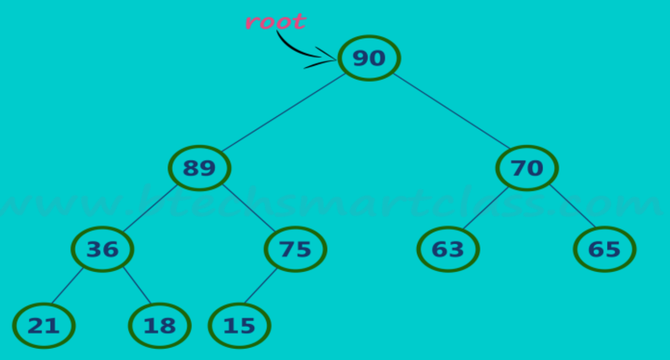
Max Heap

Max heap data structure is a specialized full binary tree data structure except last leaf node can be alone. In a max heap nodes are arranged based on node value.

Max heap is defined as follows...

Max heap is a specialized full binary tree in which every parent node contains greater or equal value than its child nodes. And last leaf node can be alone.

Example



Above tree is satisfying both Ordering property and Structural property according to the Max Heap data structure.

Operations on Max Heap

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

1. **Finding Maximum**
2. **Insertion**
3. **Deletion**

Finding Maximum Value Operation in Max Heap

Finding the node which has maximum value in a max heap is very simple. In max heap, the root node has the maximum value than all other nodes in the max heap. So, directly we can display root node value as maximum value in max heap.

Insertion Operation in Max Heap

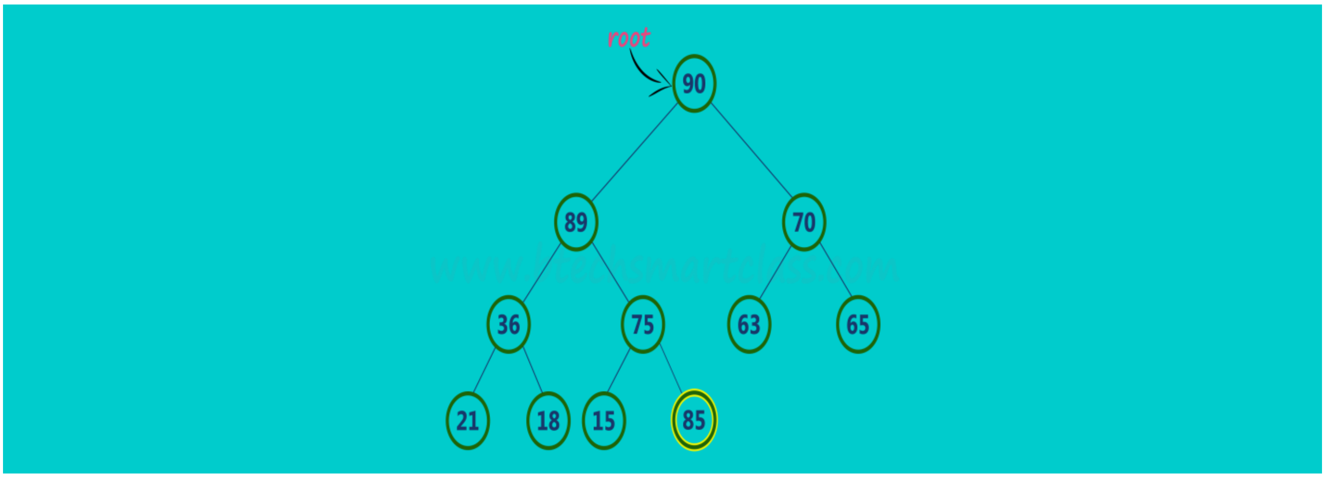
Insertion Operation in max heap is performed as follows...

- **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 nede (or) newNode reached to root.

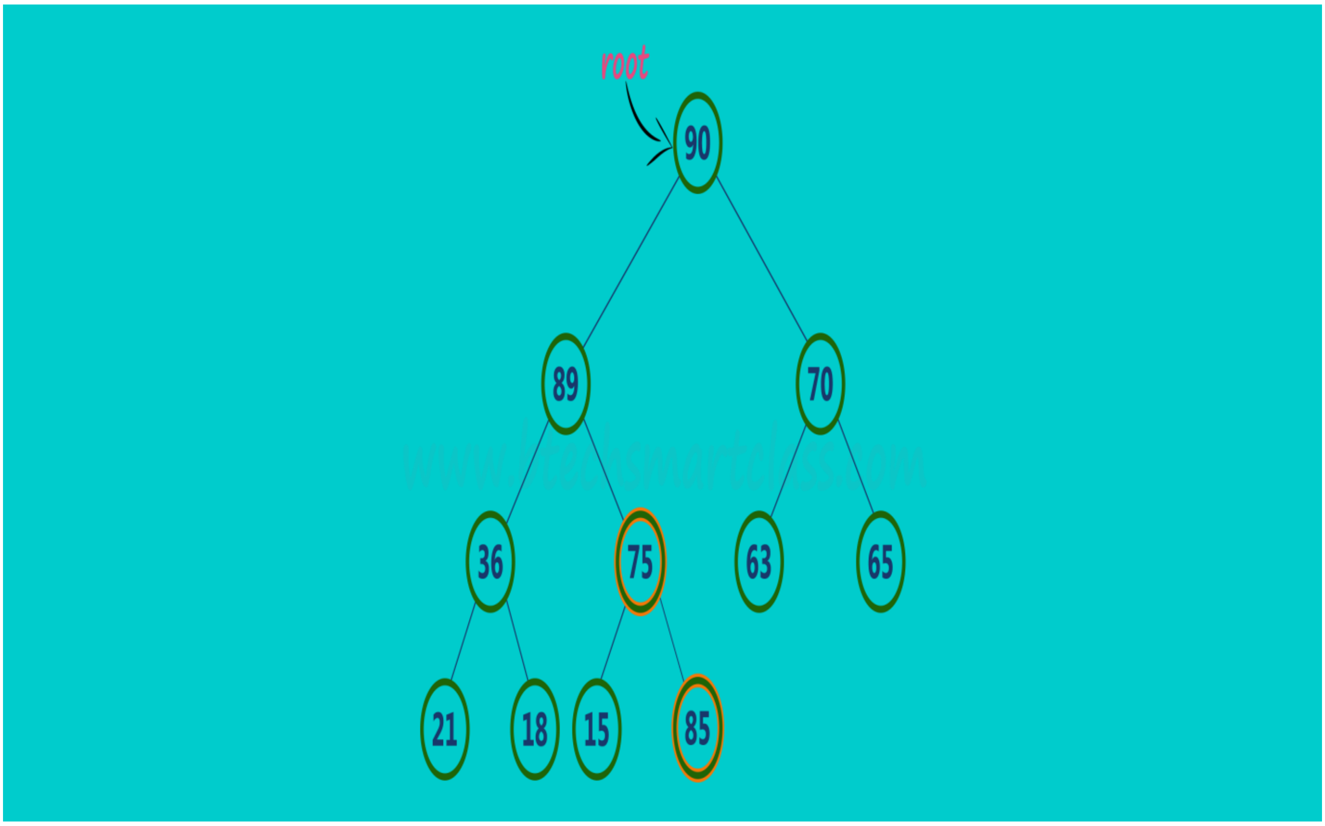
Example

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

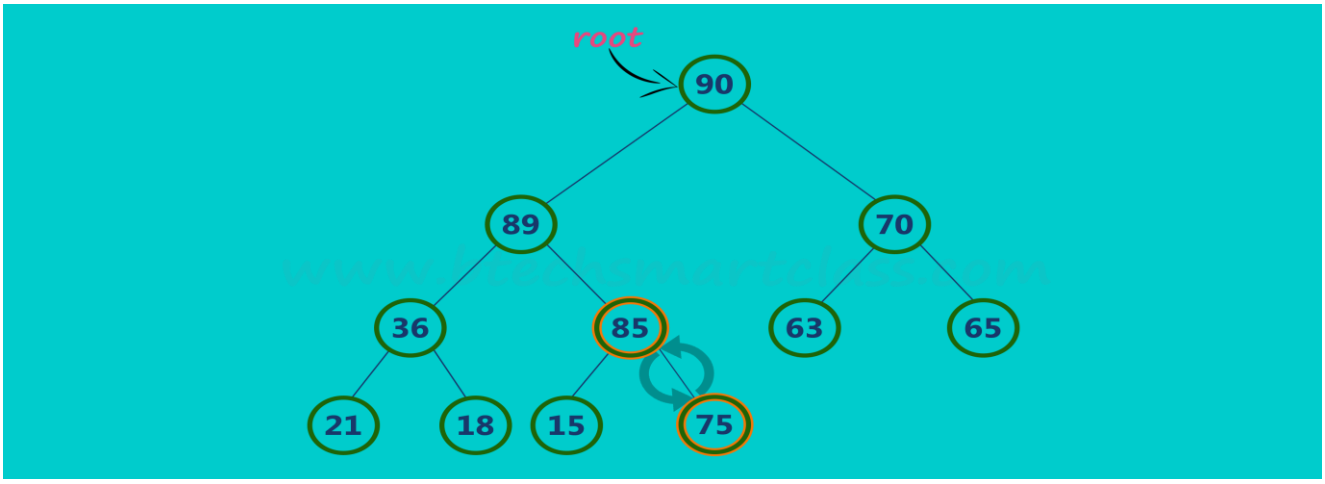
- **Step 1:** Insert the **newNode** with value 85 as **last leaf** from left to right. That means newNode is added as a right child of node with value 75. After adding max heap is as follows...



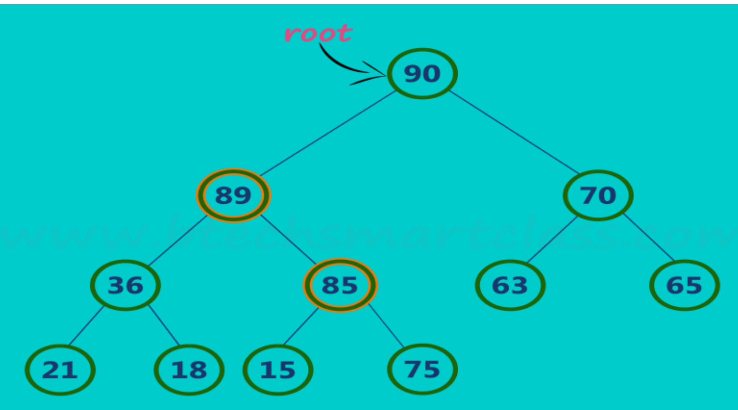
- **Step 2:** Compare **newNode value (85)** with its **Parent node value (75)**. That means **85 > 75**



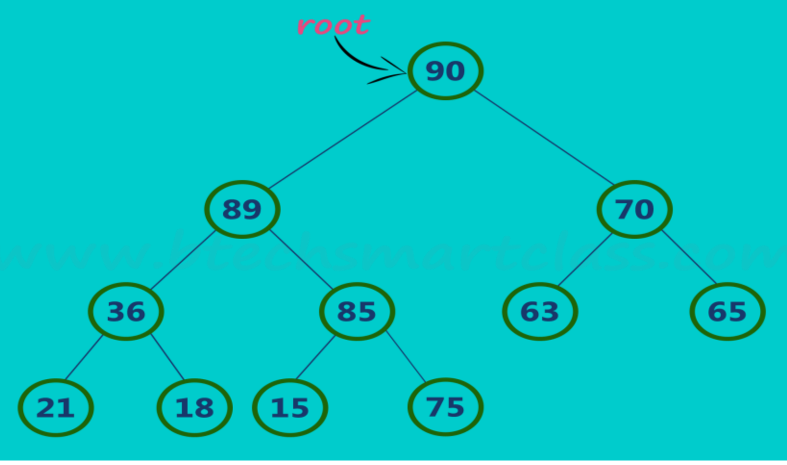
- **Step 3:** Here **new Node value (85) is greater** than its **parent value (75)**, then **swap** both of them. After wrapping, max heap is as follows...



- **Step 4:** Now, again compare newNode value (85) with its parent nede value (89).



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



Deletion Operation in Max Heap

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

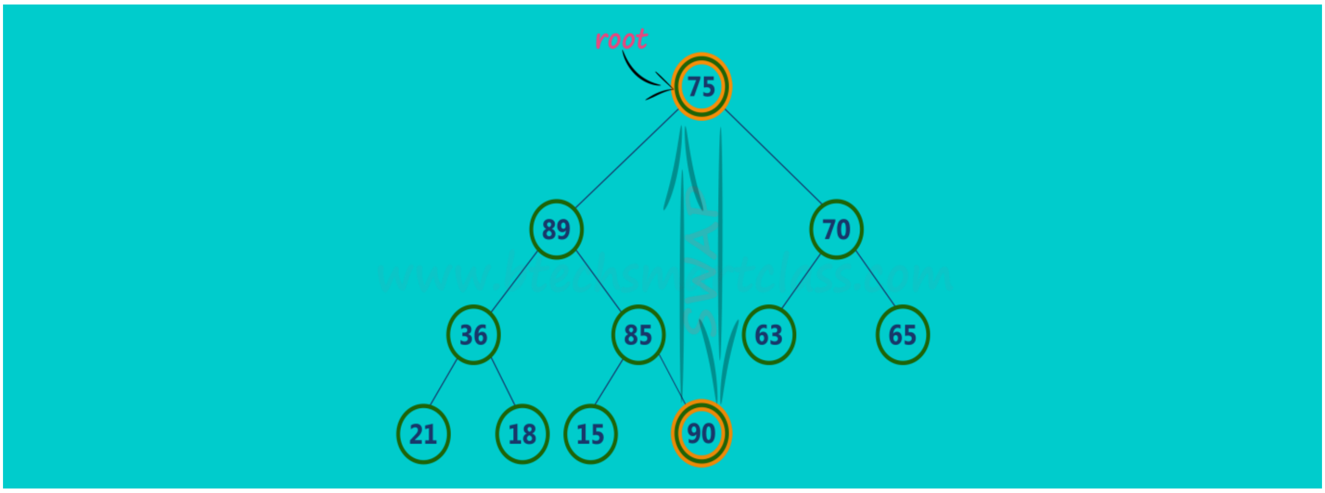
Deleting root node from a max heap is title difficult as it disturbing the max heap properties. We use the following steps to delete root node from a max heap...

- **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 **rith child**. otherwise **stop the process**.
- **Step 8:** Repeat the same until root node is fixed at its exact position.

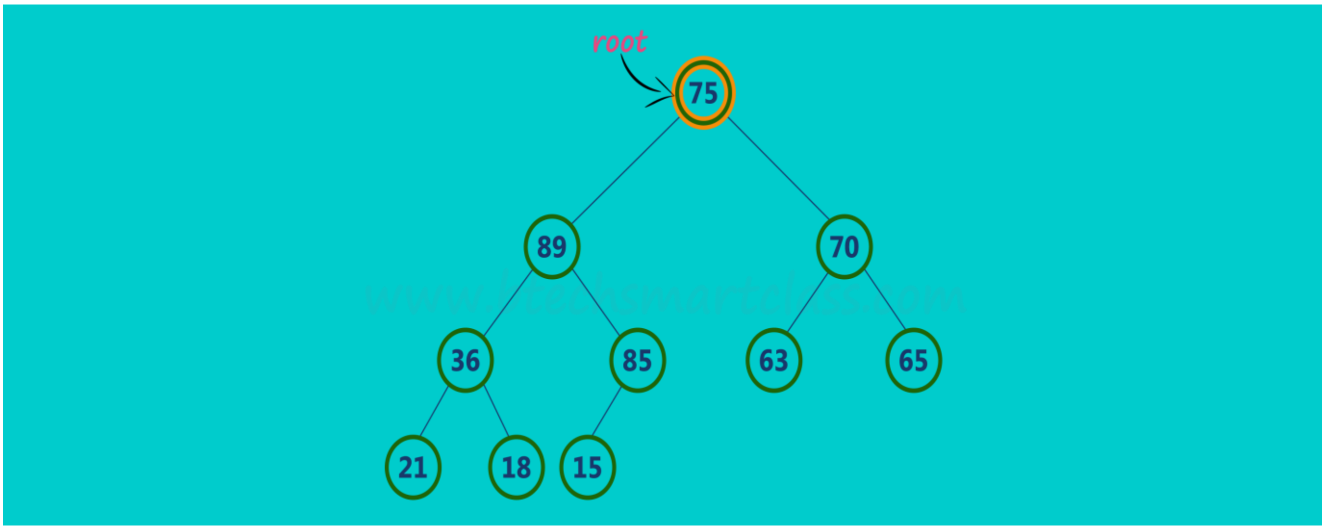
Example

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

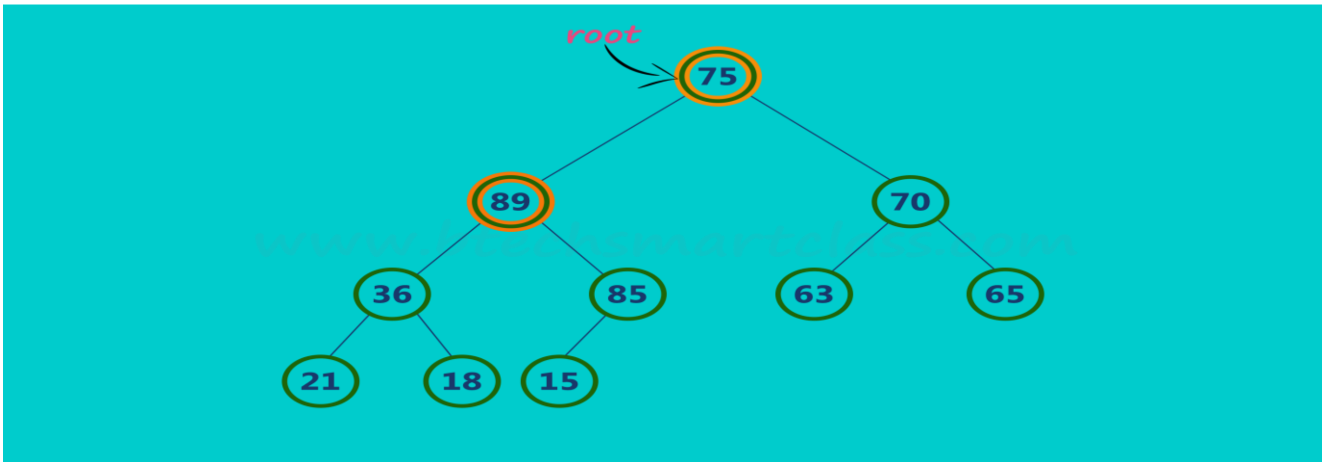
- **Step 1:** **Swap** the **root node (90)** with **last node 75** in max heap After swapping max heap is as follows...



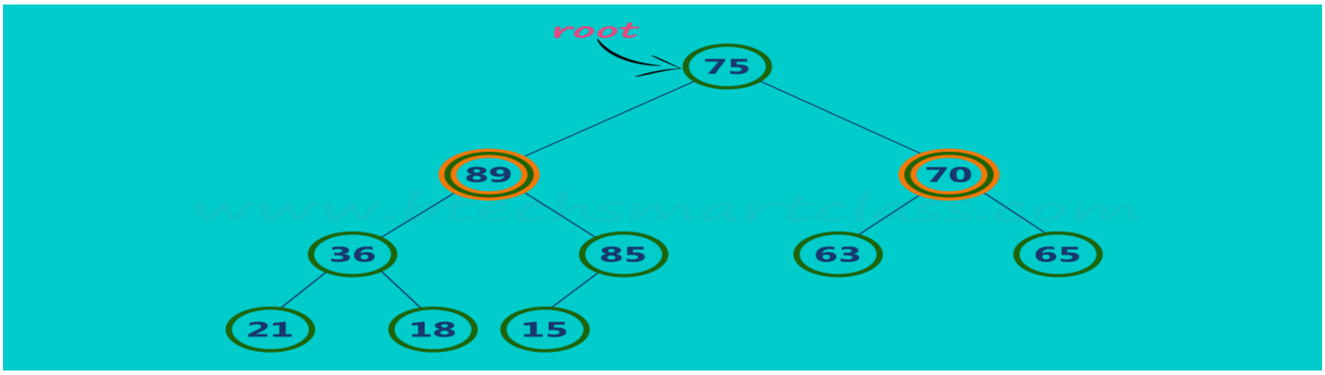
- **Step 2: Delete** last node. Here node with value 90. After deleting node with value 90 from heap, max heap is as follows...



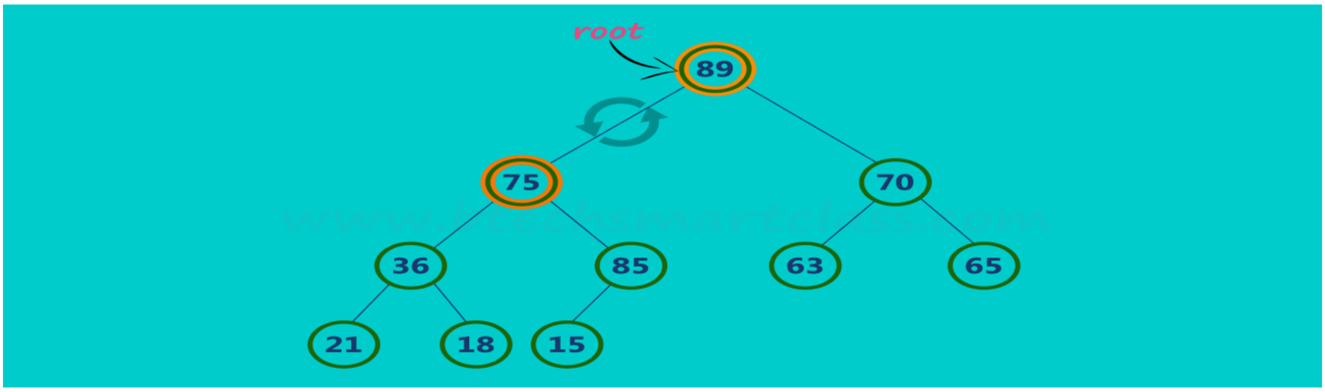
- **Step 3:** Compare **root node (75)** with its **left child (89)**.



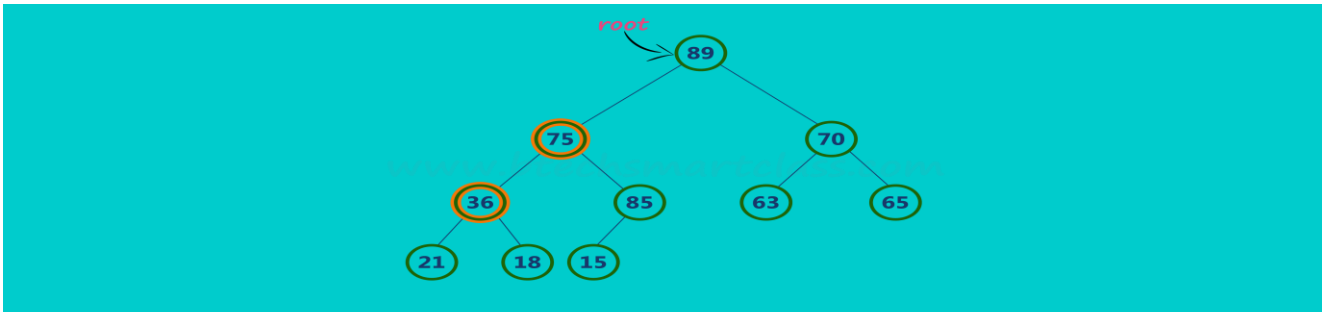
Here, **root value (75) is smaller** than its left child value (89). So, compare left child (89) with its right sibling (70).



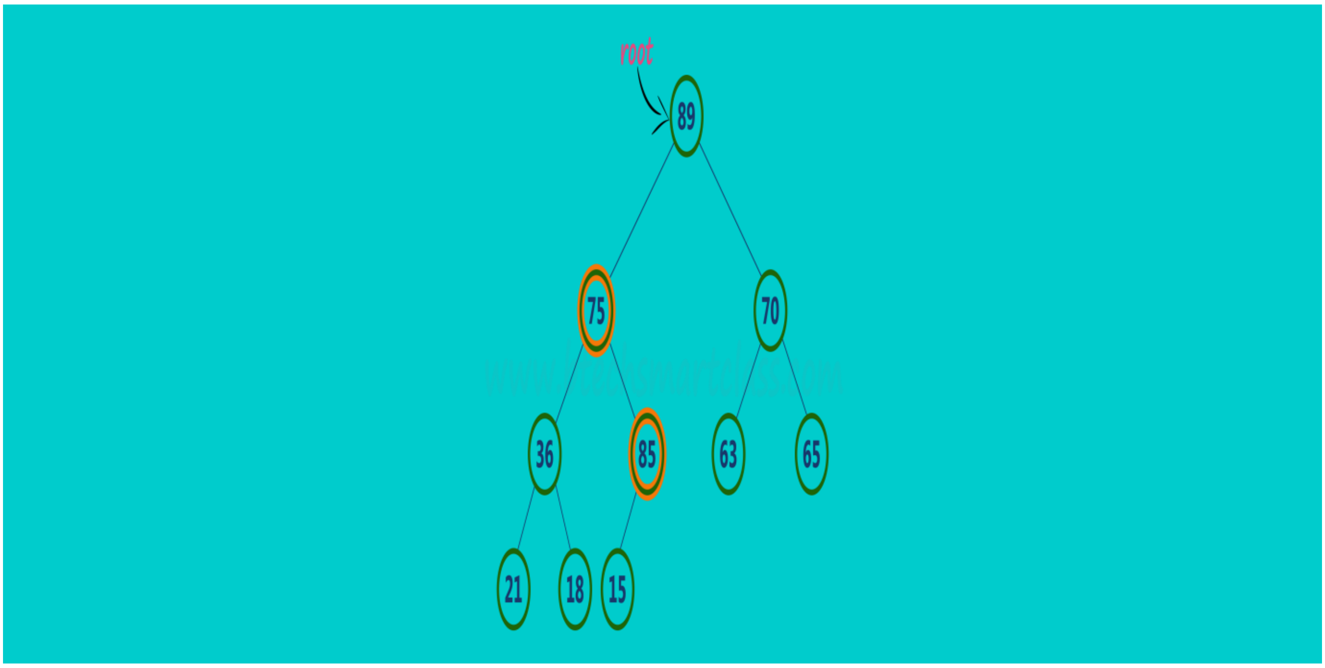
- **Step 4:** Here, **left child value (89) is larger** than its **right sibling (70)**, So, **swap root (75)** with **left child (89)**.



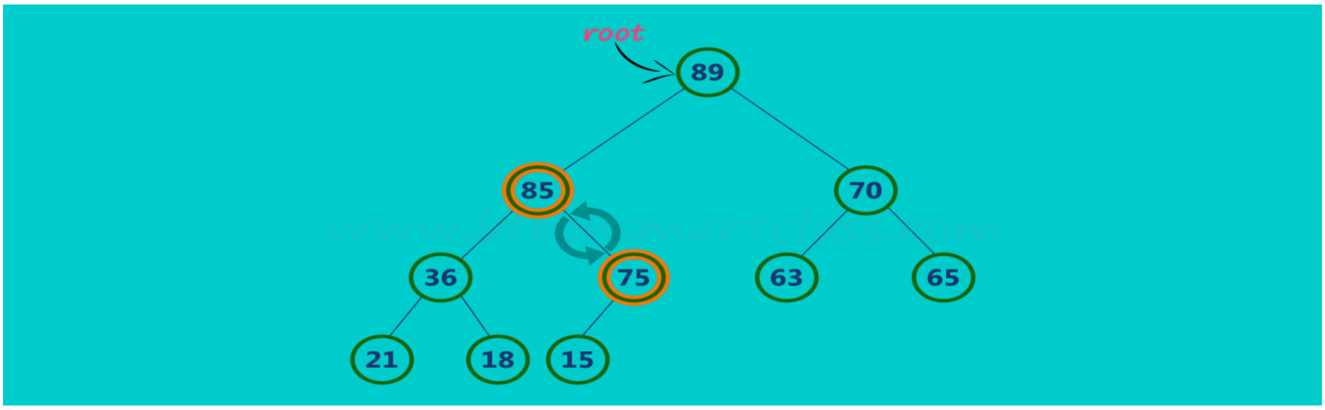
- **Step 5:** Now, again compare **75** with its **left child (36)**.



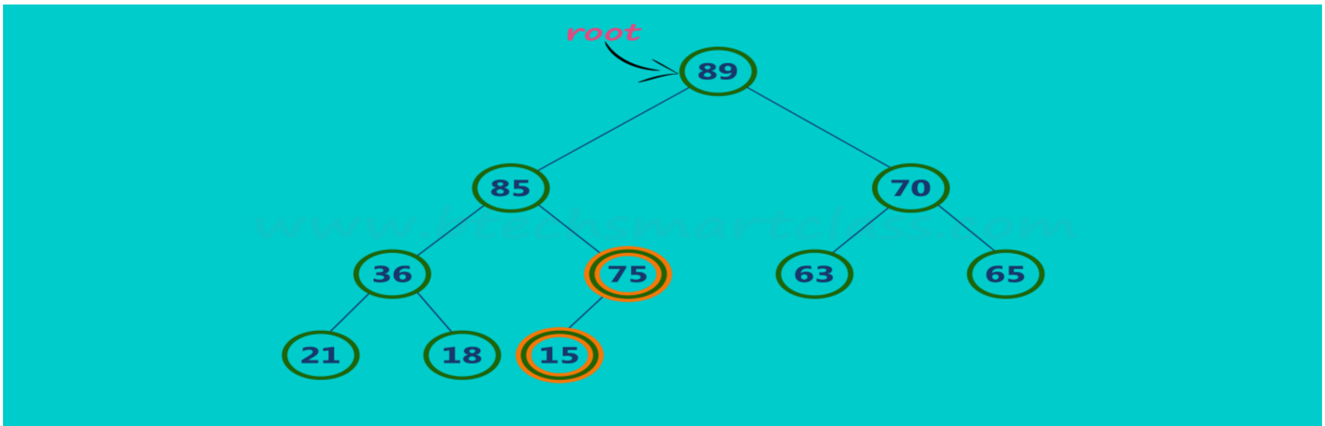
Here, node with value **75** is larger than its left child. So, we compare node with value **75** is compared with its right child **85**.



- **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...

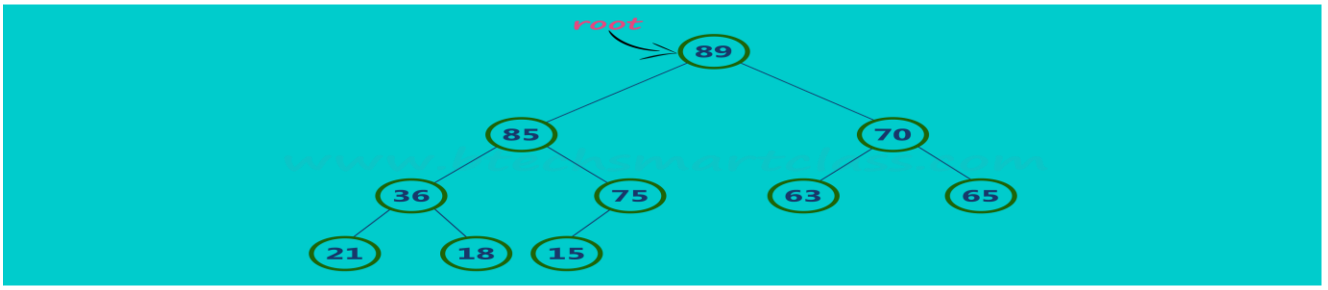


- **Step 7:** Now, compare node with value **75** with its left child (**15**).



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...


root
89
85
70
36
75
63
65
21
18
15