Unit 5

Trees and Heaps

	Syllabus	Guidelines	Suggested number of lectures
1	Introduction to tree as a data structure, binary trees, binary search trees, analysis of insert, delete, search operations, recursive and iterative traversals on binary search trees	Chapter 6, Section 6.1 - 6.6 (Threaded Trees not to be done) **Ref 1,** Chapter 5, Section 5.1, **Additional resource 4**	14
2	Height-balanced trees (AVL), B trees, analysis of insert, delete, search operations on 14 AVL and B trees	Chapter 6, Section 6.7.2, **Ref 1** Chapter 7, Section 7.1.1 **Ref 1**	
3	Introduction to heap as a data structure. analysis of insert, extract-min/max and delete-min/max operations, applications to priority queues	Chapter 8, Section 8.1.1, 8.1.3, 8.3.1-8.3.4 **Ref 2**	

References

1. Ref 1: . Drozdek, A., (2012), *Data Structures and algorithm in C++*. 3rd edition. Cengage Learning. Note: 4th edition is available. Ebook is 4th edition

2. Ref 2.: Goodrich, M., Tamassia, R., & Mount, D., (2011). *Data Structures and Algorithms Analysis in C+ +*. 2nd edition. Wiley.

3. Additional Resource 3: Sahni, S. (2011). Data Structures, Algorithms and applications in C++. 2ndEdition, Universities Press

4. Additional Resource 4: Tenenbaum, A. M., Augenstein, M. J., & Langsam Y., (2009), *Data Structures Using C and C++*. 2nd edition. PHI.

Note: Ref1, Additional resource etc. as per the LOCF syllabus for the paper.

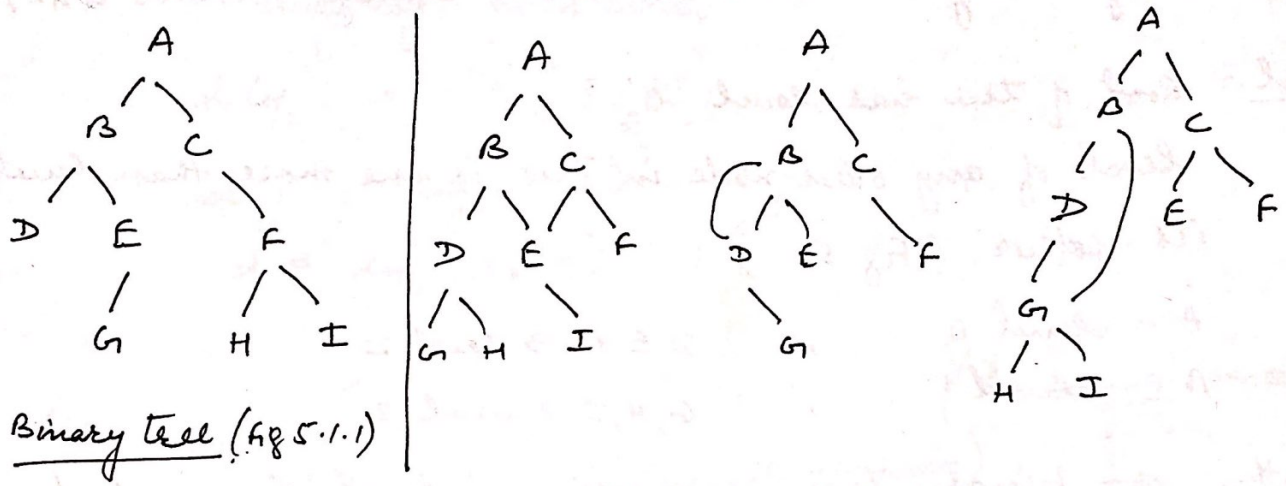
Ch 5.

5.1 Binary Trees

Def:- A Binary Tree is a finite set of elements that is either empty or is partitioned into three disjoint subsets.

first subset :- contains a single element called the 'root'.

other two subsets :- are themselves binary trees, called 'left' & 'right' subtrees of the original tree.



Binary tree (Fig 5.1.1)

father :- A is father of B.

leftson :- B is left son of A.

rightson :- C is right son of A.

leaf :- node having no sons. as D, G, H, I.

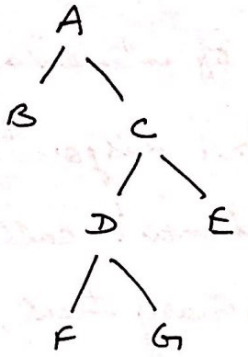
ancestor :- A is ancestor of B, C, D, E, G, F, H, I.
B is ancestor of D, E, G.

descendant :- D is descendant of B, A.

left descendant :- D is left descendant of B.
D is left " of A.
E is right descendant of B.
E is left descendant of A.

Brothers :- D & E are brothers.
E & F are not brothers. [If nodes are left & right sons of the same father.]

Strictly binary tree:- If every non-leaf node in a binary tree has non-empty left & right subtrees.



n → leaves

2n-1 → nodes

n=4

nodes = 8-1 = 7

A strictly binary tree with 'n' leaves always contain 2n-1 nodes.

level:- Root of tree has level '0'.

level of any other node in tree is one more than level of its father. (Fig 5.1.1)

A - level 0 , D,E,f → level 2

B,C - level 1 , G,H,I → level 3.

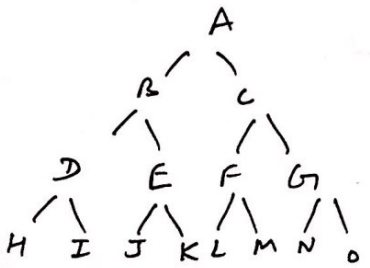
Depth:- of a binary tree is maximum level of any leaf.

eg:- Depth of tree 5.1.1. is '3'.

(Full BT by some authors)

Complete binary tree:- complete binary tree of depth 'd' is ~~a~~ strictly binary tree all of whose leaves are at level 'd'.

(1) (2)



complete-binary tree of depth 'd'.

d = 3.

If a binary tree contains 'm' nodes at level 'l', it contains atmost '2m' nodes at level 'l+1'.

Total no. of nodes 'tn' in complete binary tree of depth 'd' is:-

$$t_n = 2^0 + 2^1 + 2^2 + \ldots + 2^d = \sum_{j=0}^{d} 2^j \quad OR \quad t_n = 2^{d+1} - 1$$

All leaves are at level 'd' thus 2^d leaves. $n = 2^m - 1 \therefore d+1 = m$

$\therefore 2^d - 1$ non-leaf nodes.

If t_n total no. of nodes are given in a complete binary tree, then depth 'd' can be calculated as

$$t_n = 2^{d+1} - 1 \quad \text{(given)}.$$

$$t_n + 1 = 2^{d+1}$$

$$t_n + 1 = 2^d \cdot 2$$

$$(t_n + 1)/2 = 2^d.$$

Taking log both sides,

$$d \log_2 2 = \log_2 \left(\frac{t_n + 1}{2}\right)$$

$$d \cdot 1 = \log_2 (t_n + 1) - \log_2 2$$

$$d = \log_2 (t_n + 1) - 1$$

Also $t_n = 2^d + 2^d - 1$ (In case of complete binary tree).

(leaf nodes) (non-leaf nodes)

$$= 2 \cdot 2^d - 1 = 2 \cdot 2^3 - 1 = 15 = t_n$$

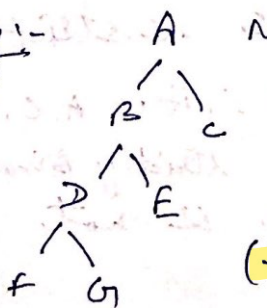
Almost Complete Binary Tree (or Complete BT by some authors)

A binary tree of depth 'd' is an almost complete binary tree if:

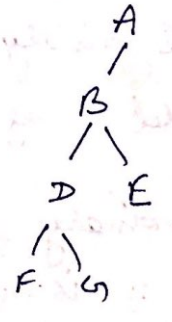
1) Any node 'nd' at level less than 'd-1' has two sons. OR every leaf is either at level 'd' or level 'd-1'.

2) For any node 'nd' in the tree with a right descendant at level 'd', nd must have a left son & every left descendant of nd is either a leaf at level d or has two sons.

eg:-

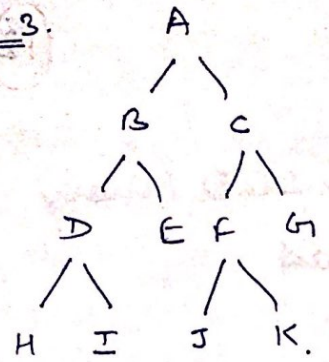


Not almost complete Binary tree since leaves are at level 1, 2, 3. Violating cond ①. (X)



level(A) < d-1 i.e. 0 < 2 but A has 1 son violating cond ① (X). Thus it is not almost complete B. Tree.

eg 3.

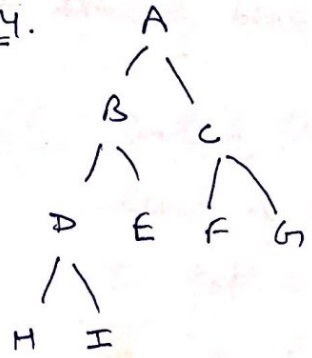


(✓) cond ① → satisfied since leaves are at level d or d-1 ie 3 or 2.

(✗) Cond ② → not satisfied, since A has a right descendant at level 3(J) but also has a left descendant that is a leaf at level 2 (E). (d-1).

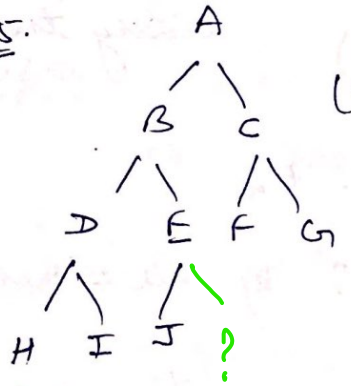
Thus, not a Almost complete binary tree.

eg-4.



(✓) It is an 'almost complete binary tree' as both cond. ① & ② are satisfied.

eg-5.



(✓) It is an 'almost complete binary tree' as both cond. are satisfied but it is not 'strictly binary tree'.

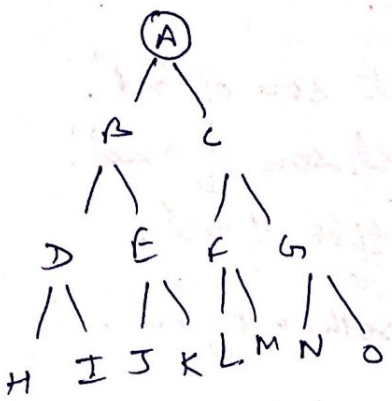
An 'almost complete strictly binary tree with 'n' leaves has '2n-1' nodes. (eg-4).

An 'almost complete binary tree with 'n' leaves that is not strictly binary has '2n' nodes. (eg-5).

An almost complete binary tree with 'n' nodes is strictly binary iff n is odd. (eg-4). [here n = 9] ∴ it is A.C.B.T. & strictly Binary Tree.

An almost Complete binary tree with 'n' nodes is not strictly binary if n is even (eg-5) (n = 10)

An almost complete binary tree of depth 'd' is intermediate b/w the complete binary of depth $d-1$ that contains $2^d - 1$ nodes & the complete binary tree of depth 'd' that contains $2^{d+1} - 1$ nodes



depth = $\boxed{d = 3.}$ then $n = 2^{d+1} - 1 = 2^4 - 1 = 15$ nodes

If $\boxed{d-1}$ then $2^d - 1 = 2^3 - 1 = 7$ nodes.

Thus no. of nodes of almost complete Binary tree lies b/w 7 to 15.

If 't_n' is the total no. of nodes in an almost complete binary tree, its depth is the largest integer less than or equal to $\log_2 t_n$.

(consider above tree)

$d : \log(t_n + 1) - 1$

$t_n \rightarrow$ nodes

eg:- If $t_n = 8$,

then depth = $\log_2 8 = 3$. $\therefore \boxed{\text{depth} = 3}$

$d <= \log t_n$

If $t_n = 15$,

then depth = $\log_2 15 = 3.9....$

largest integer less than 3.9 is 3.

$\therefore \boxed{\text{depth} = 3.}$

Operations on Binary Trees

If 'p' is a pointer to a node 'nd' of a binary tree:-

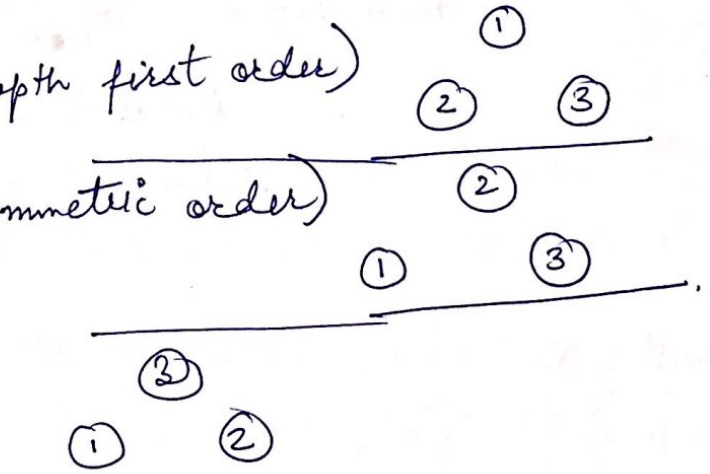
1) Info(p) :- returns the contents of 'nd'.
2) left(p) :- returns pointers to the left son of 'nd'.
3) right(p) :- return pointers to the right son of 'nd'.
4) father(p) :- return pointer to the father of 'nd'.
5) brother(p) :- return pointer to the brother of 'nd'.

These functions return 'null' pointer if 'nd' has no left son, right son, father or brother.

6) isleft(p) :- returns the value true if 'nd' is a left child otherwise false.
7) isright(p) :- returns the value true if 'nd' is a right child otherwise false.

Applications of Binary Trees

1) finding Duplicates (An app. of BST's).
2) traversing a Binary Tree.
   - (a) Preorder (depth first order)
   - (b) Inorder (symmetric order)
   - (c) Postorder

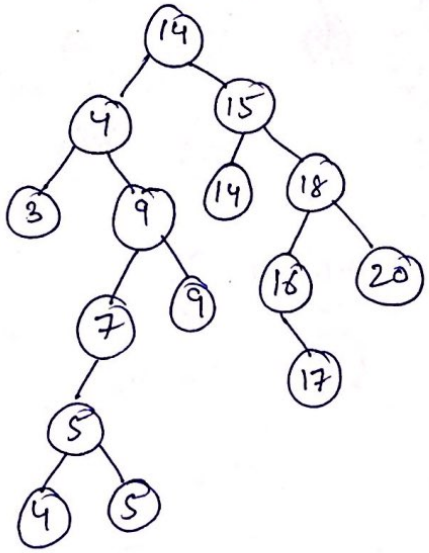


3) Representing an expression.

I/P string :-

14 15 4 9 7 18 3 5 16 4 20 17

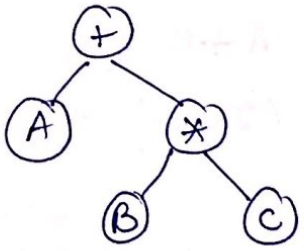
9 14 5



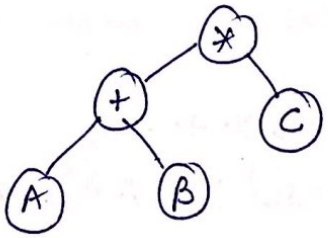
BST has the property that all elements in the left subtree of a node n are less than the contents of n. And all elements in the right subtree of n are greater than or equal to the contents of n.

(III). Representing an Expression :-

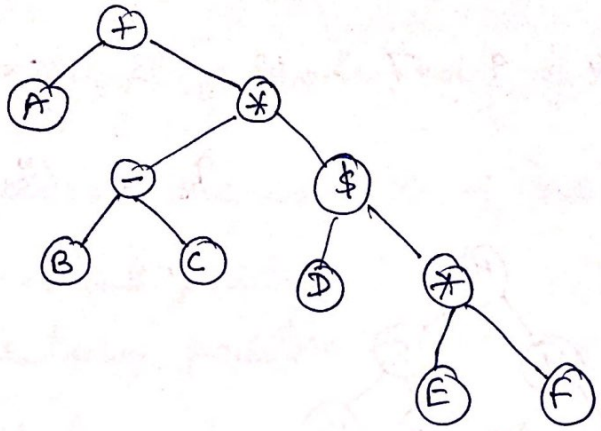
eg:- (a) A + B * C



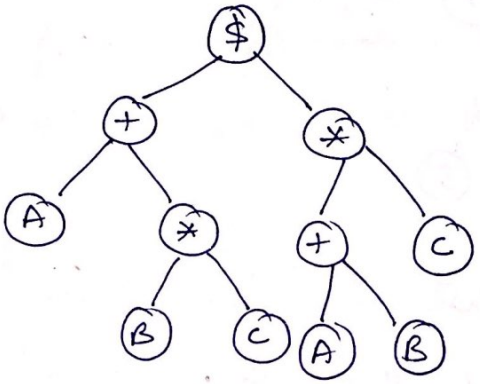
(b) (A + B) * C



eg ① $A + (B - C) * D \$ (E * F)$



eg - ② $(A + B * C) \$ ((A + B) * C)$

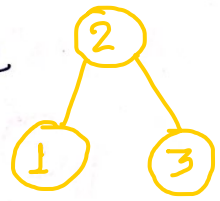


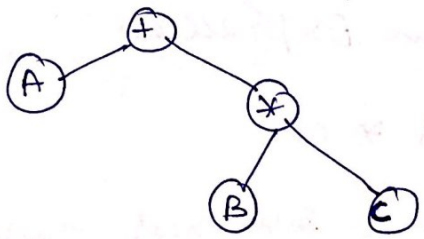
Preorder traversal gives prefix form of the expression.

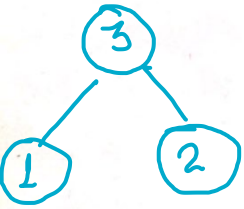
postorder traversal produces postfix form of the expression.

eg ③. $A + B * C$

(Infix).

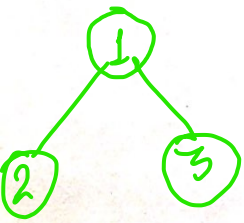






Postfix :- $ABC * +$.

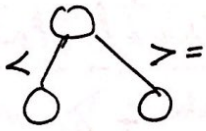
Postorder traversal :- $ABC * +$.



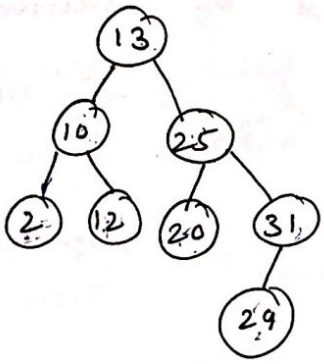
Prefix :- $+ A * BC$.

Prefix traversal :- $+ A * BC$.

Binary Search Tree



string :- 13, 10, 25, 2, 12, 20, 31, 29



Inorder - 2 10 12 13 20 25 29 31

Preorder - 13 10 2 12 25 20 31 29

Postorder - 2 12 10 20 29 31 25 13

BST also called ordered Binary Trees.

Tree Traversals :-

1) BFS

2) DFS - ~~Pre~~ Preorder (1), Inorder (2), Postorder (2).

(2) (3) (1) (3) (1) (2)

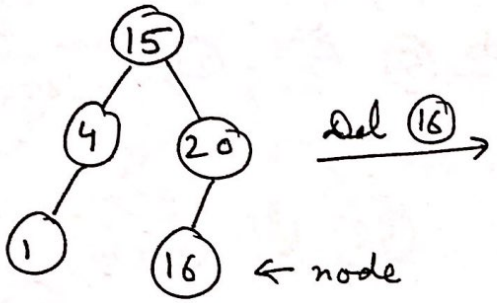
worst case in above eg. is when search is for 26, 27, 28, 29, or 30 because each search requires four tests.

Complexity of Searching :- is measured by no. of comparisons performed during the searching process. This no. depends on the no. of nodes encountered on the unique path leading from the root to the node being searched for. ∴ The complexity is the length of the path leading to this node plus 1.

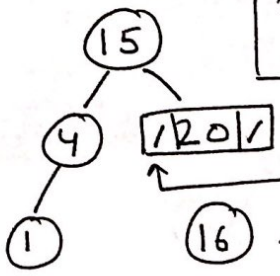
Deletion in BST

Three cases :-

case 1. When node to be deleted has no child. (It is a leaf).



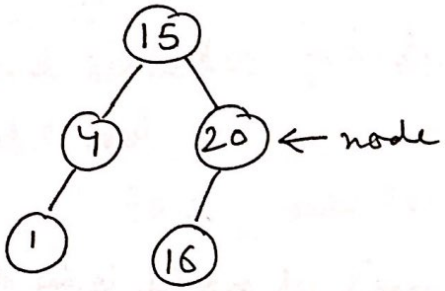
Del (16) →



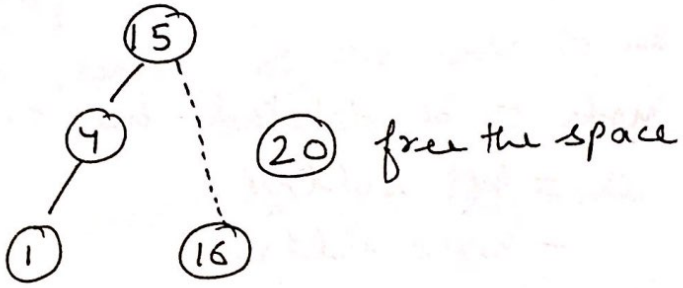
node to be deleted - 16
Actual node deleted - 16

case 2. When node to be deleted has one child (either left child or right child).

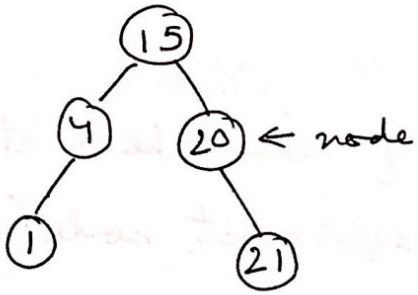
#



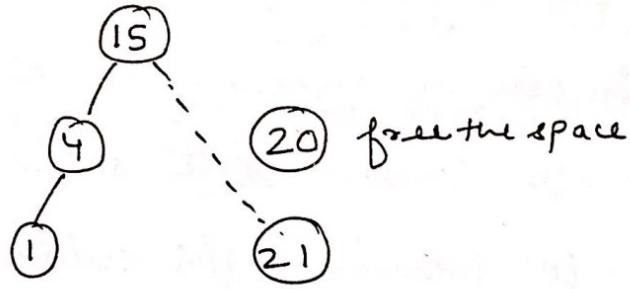
del (20) →



#



del (20) →



node to be deleted - 20
actual node deleted - 20

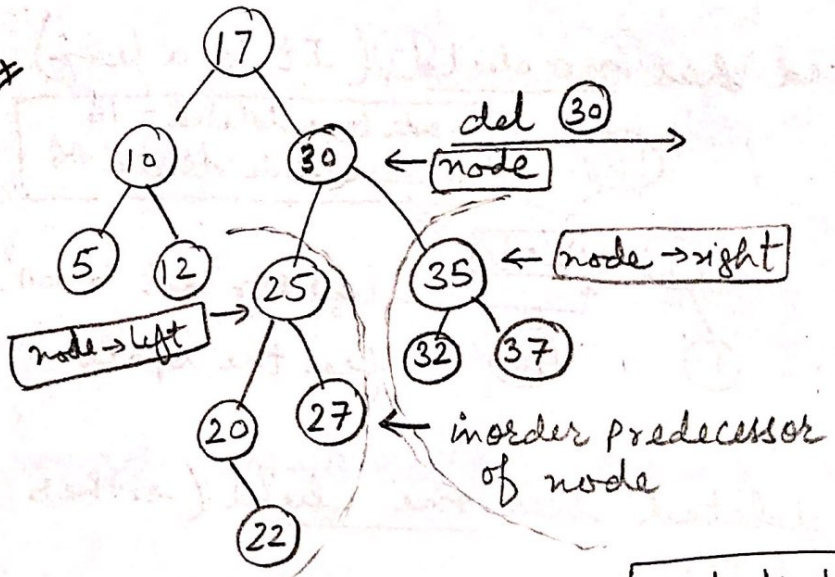
case 3 When node to be deleted has two children.

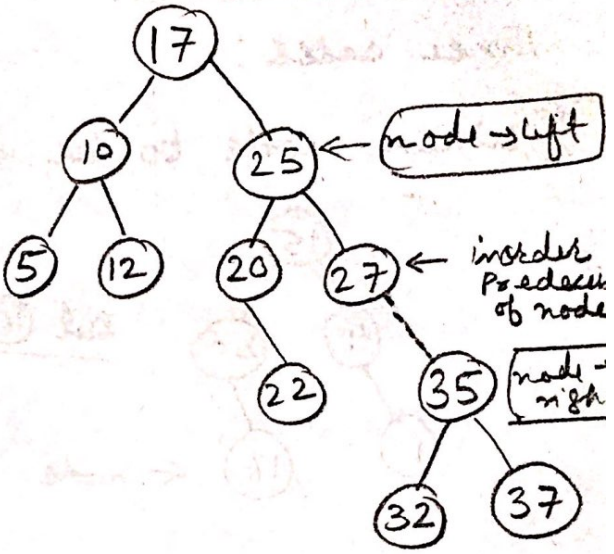
Two methods :-

a). Deletion by Merging.

b). Deletion by copying.

a). Deletion by Merging





node to be deleted - 30
Actual node deleted - 30

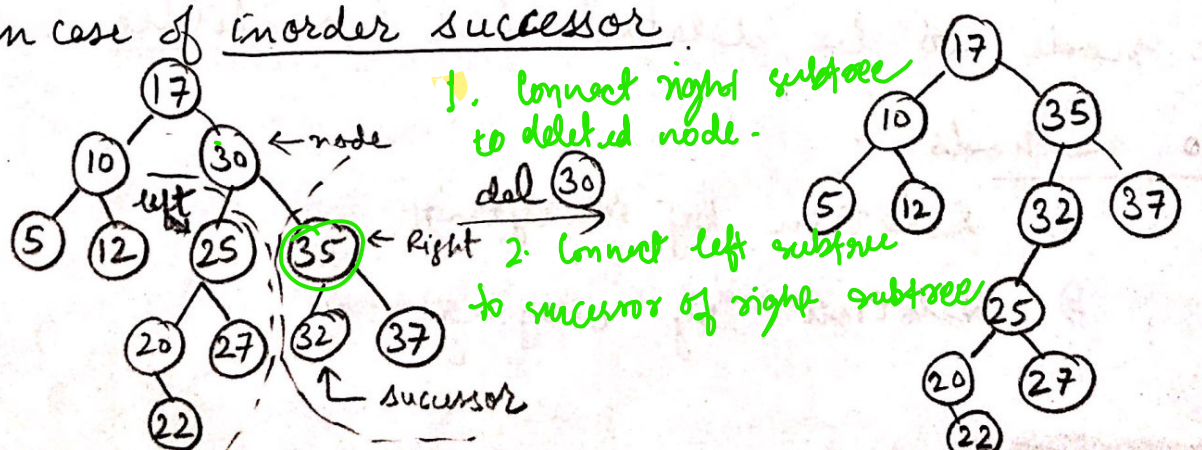
node to be deleted has two subtrees :-
- Left subtree
- Right subtree

Find inorder predecessor (or successor) of node to be deleted.

In case of inorder predecessor
(i) Connect left subtree in place of node to be deleted.
(ii) Connect right subtree to the rightmost node (inorder predecessor) of left subtree.

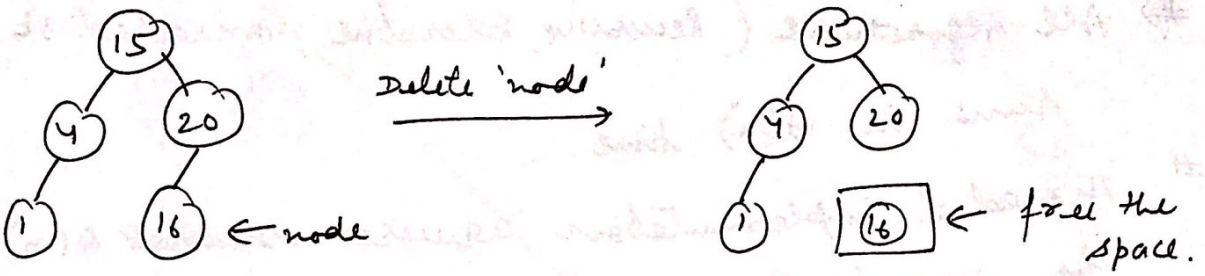
Free the space of node to be deleted.

In case of inorder successor.

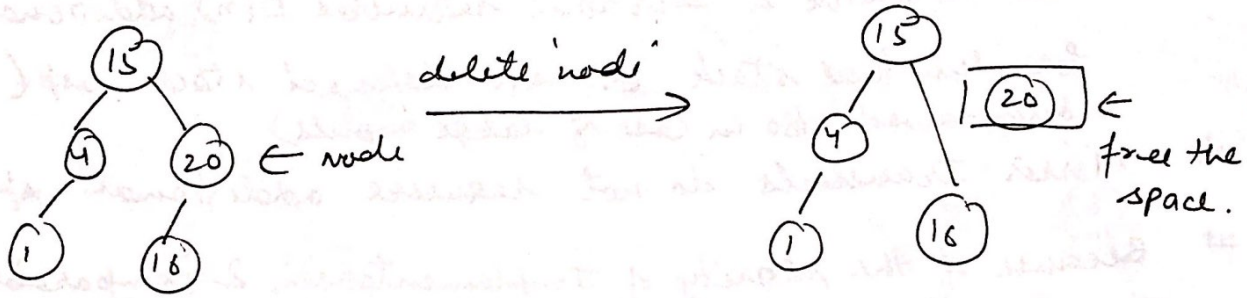


Deletion in BST

Case-I node to be deleted has no childs.



Case II :- Node to be delete has only one child (either left or right).



Case III Node to be deleted has two childs.

Two solutions :-

1. Deletion by merging.
2. Deletion by copying.

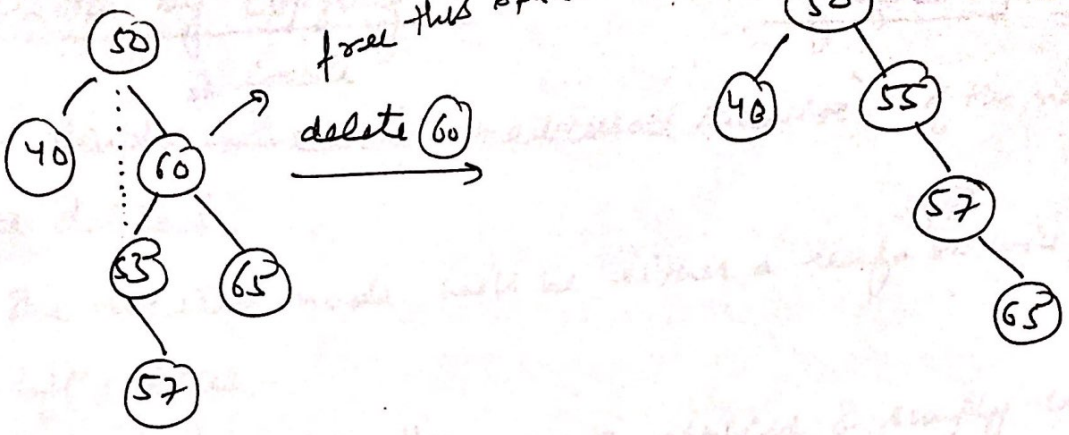
(1) Deletion by Merging.

(a) Go To left subtree & then find rightmost node. (or inorder predecessor).

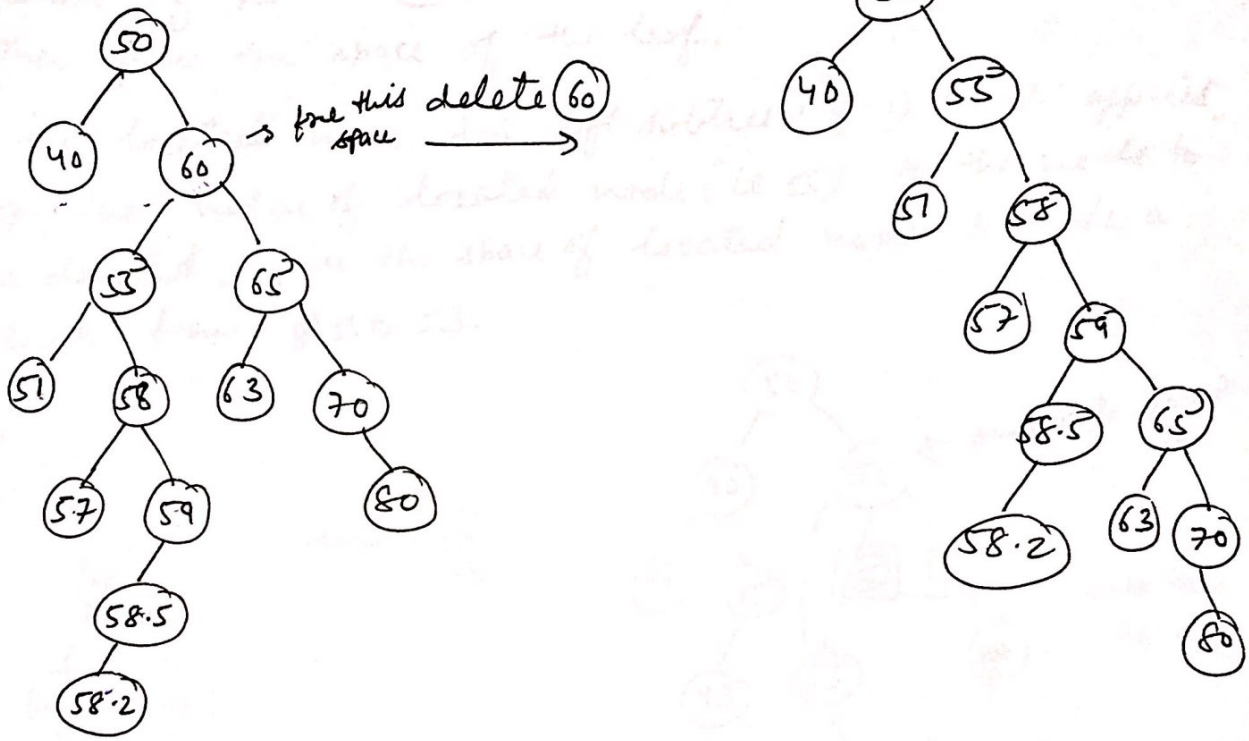
(b) make leftsubtree of the deleted node as the parent. (in place of deleted node).

(c) merge right subtree of deleted node to the right of left subtree.

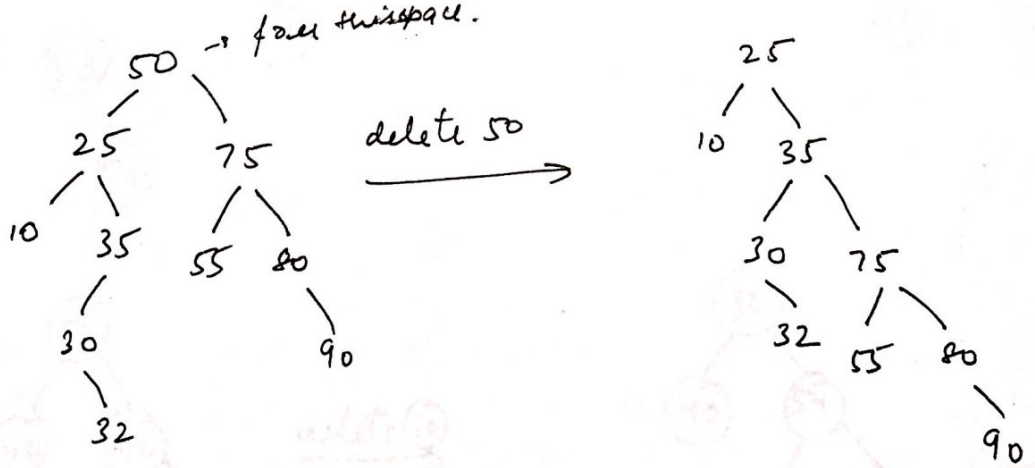
eg 1.


50
40
60
55
65
57
free this space.
delete 60
50
40
55
57
65

eg 2.


50
40
60
55
65
51
58
63
70
57
59
80
58.5
58.2
free this space
delete 60
50
40
55
51
58
57
59
58.5
65
58.2
63
70
80

eg 3.

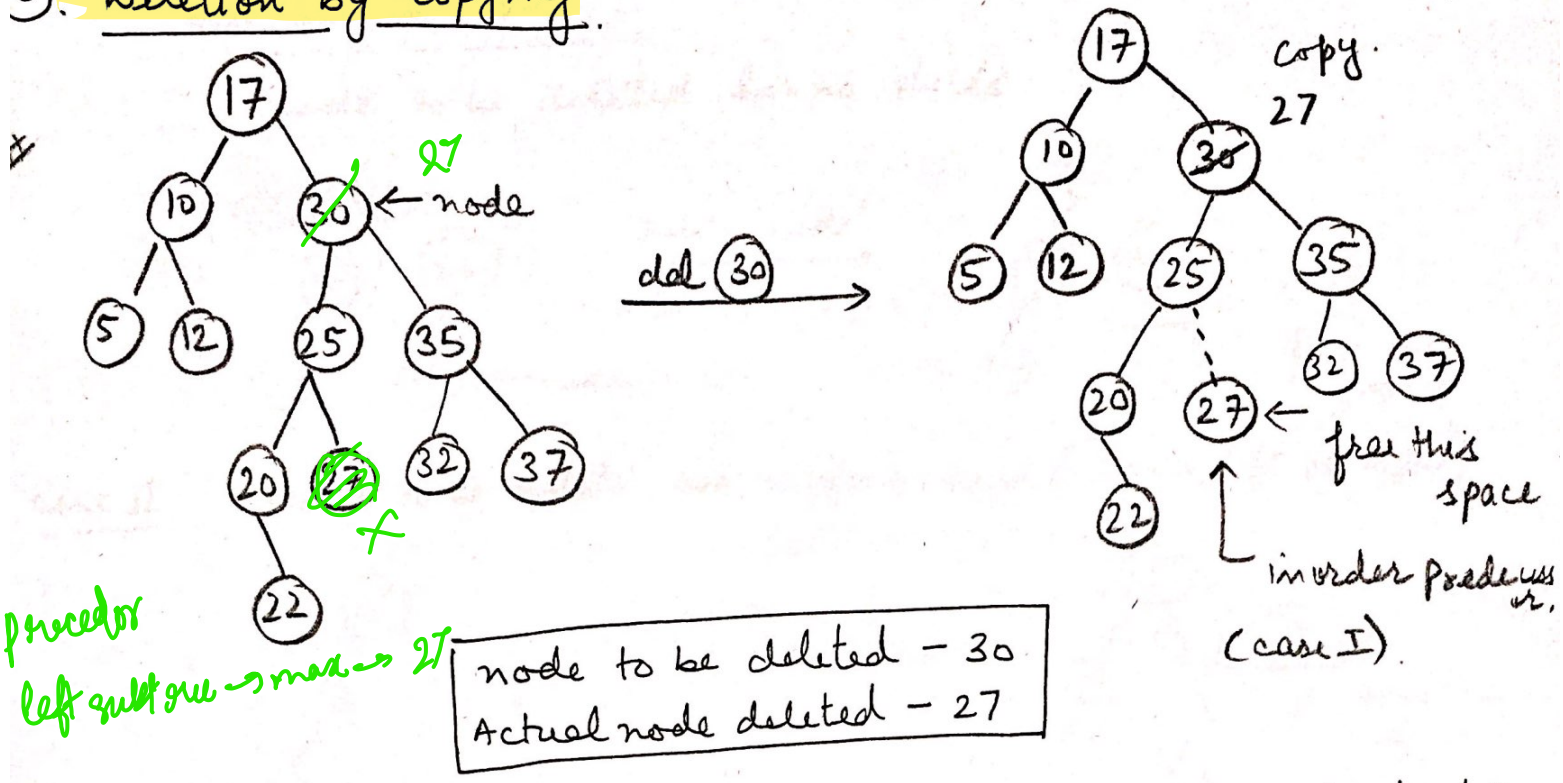

50
25
75
10
35
55
80
30
90
32
free this space.
delete 50
25
10
35
30
75
32
55
80
90

```
void bst::delete_by_merging(node* temp,int el)
{
  node* prev=NULL;
  while(temp!=NULL)
  {
        if(temp->info==el)
              break;
        prev=temp;
        if(temp->info<el)
             temp=temp->right;
        else
             temp=temp->left;
   }
   if(temp!=NULL &&temp->info==el)
       {
             if(temp==root)
                   delmerge(root);
             else if(prev->left==temp)
                   delmerge(prev->left);
             else
                   delmerge(prev->right);
       }
       else if(root!=0)
             cout<<"\nKEY "<<el<<" IS NOT IN TREE";
       else
             cout<<"\nTREE IS EMPTY";
 }

 void bst::delmerge(node*& temp1)
 {
       node* tmp= temp1;
       if(temp1!=NULL)
       {
             if(temp1->right==NULL)
               temp1=temp1->left;
             else if(temp1->left==NULL)
               temp1=temp1->right;
             else
             {
               tmp=temp1->left;
               while(tmp->right!=NULL)
                     tmp=tmp->right;
               tmp->right=temp1->right;
               tmp=temp1;
               temp1=temp1->left;
             }
             delete tmp;
       }
 }
```

Predecessor

b). **Deletion by Copying.**

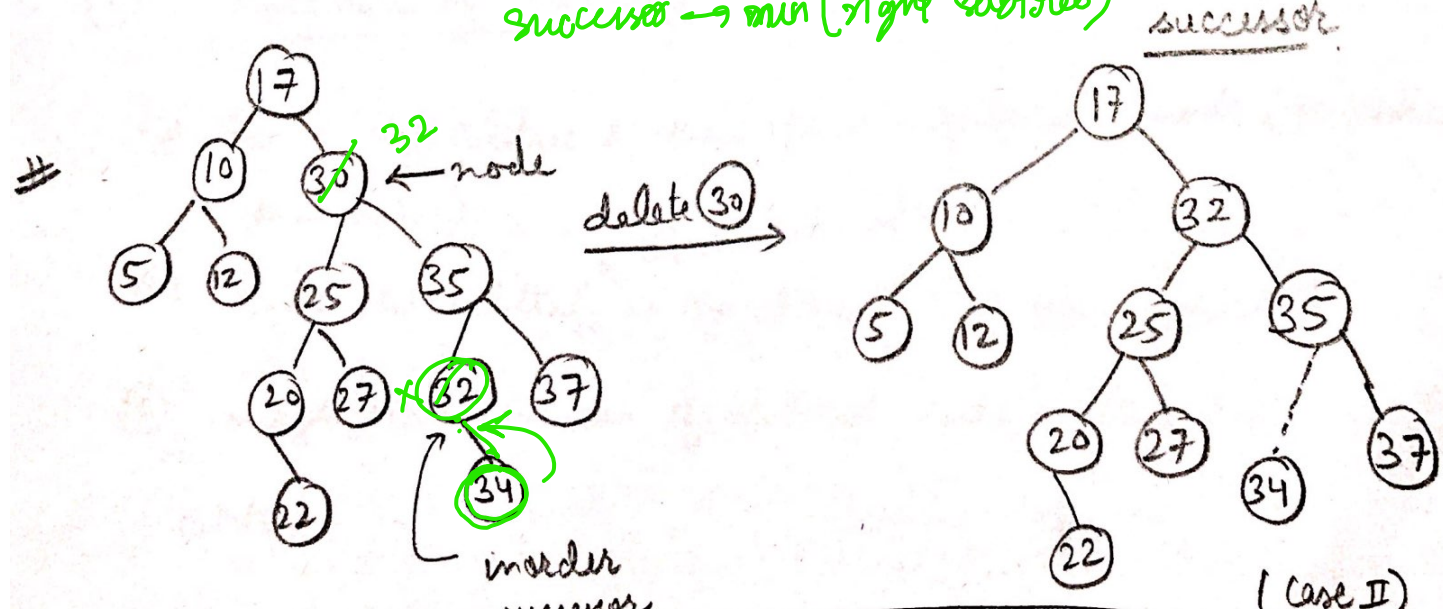


predecessor
left subtree → max → 27

node to be deleted - 30
Actual node deleted - 27

1) Find inorder predecessor (or successor) of the node to be deleted.
2) Copy this predecessor (or successor) in place of node that is to be deleted.
3) Now delete (or free the space) of copied node value. (This again brings us to the first or second case).

successor → min (right subtree) → 32



node to be deleted - 30
Actual node deleted - 32

Deletion by copying

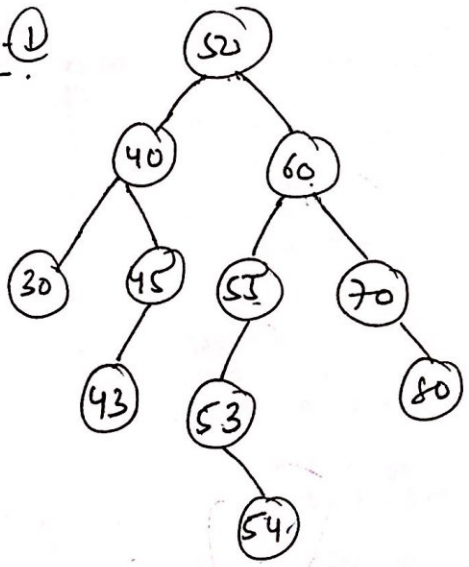
(2). (1) find immediate predecessor (successor) of the node to be deleted.
or locate

(2). The located node will be either a leaf (eg ②) or will have left subtree.

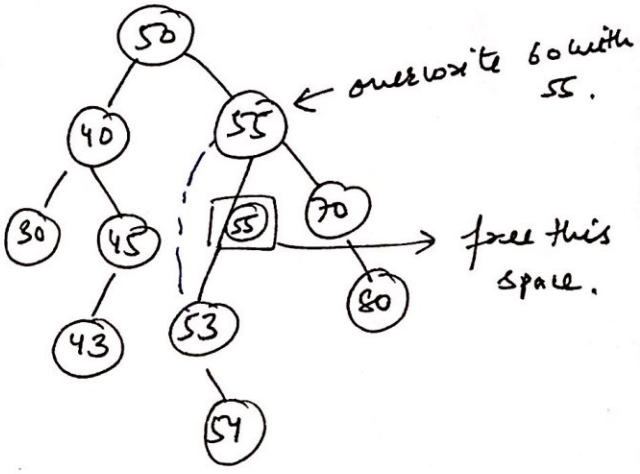
(3). If it is a leaf, then case I applies & simply copy the value of node i.e (54) to the node to be deleted. Then free the space of the leaf.

(4) If the located node has left subtree (eg-1) case II applies, copy the value of located node (i.e 55) to the node to be deleted, free the space of located node & make a link from 55 to 53.

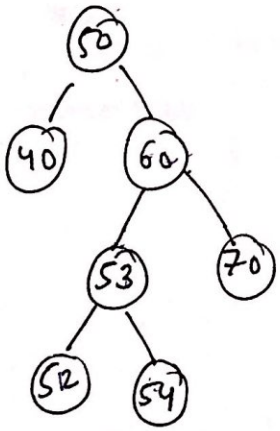
eg: (1)



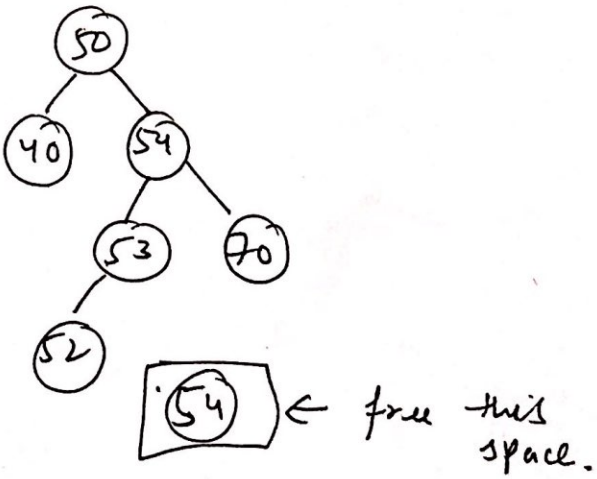
delete (60).



eg: (2)



delete (60)



```
void bst::delete_by_copying(node* temp,int el)
{
  node* prev=NULL;
  while(temp!=NULL)
  {
        if(temp->info==el)
              break;
        prev=temp;
        if(temp->info<el)
             temp=temp->right;
        else
             temp=temp->left;
  }
  if(temp!=NULL &&temp->info==el)
      {
             if(temp==root)
                   delcopy(root);
             else if(prev->left==temp)
                   delcopy(prev->left);
             else
                   delcopy(prev->right);
      }
      else if(root!=0)
            cout<<"\nKEY "<<el<<" IS NOT IN TREE";
      else
            cout<<"\nTREE IS EMPTY";
}

void bst::delcopy(node*& temp1)
{
      node* prev,*tmp= temp1;
      if(temp1->right==NULL)
            temp1=temp1->left;
      else if(temp1->left==NULL)
            temp1=temp1->right;
      else
      {
        tmp=temp1->left;
        prev=temp1;
        while(tmp->right!=NULL)
        {
              prev=tmp;
              tmp=tmp->right;
        }
        temp1->info=tmp->info;
        if(prev==temp1)
              prev->left=tmp->left;
        else
              prev->right=tmp->left;
      }
      delete tmp;
}
```

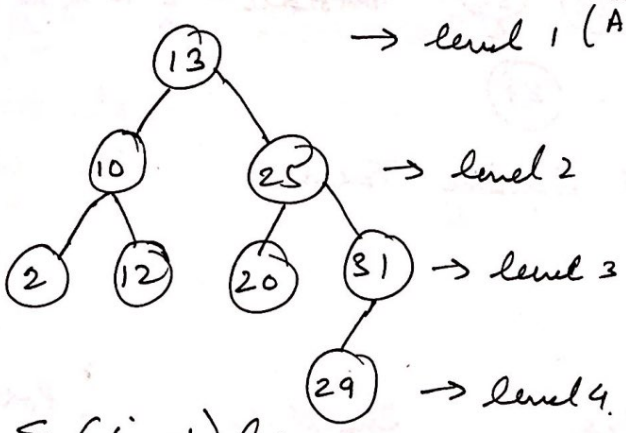
An important point to notice is that, In deletion by copying method, predecessor value is copied to the 'node to be deleted' value & space that is freed is of Predecessor. (ie why it is called del. by copying).

But 'In deletion by merging' space freed is of the node to be deleted. Nothing is copied or no key value is overwritten of any node. only new links are created.

Not in course w.e.f 2019-20 (CCBCS)

Internal Path length (IPL)

IPL is the sum of all path lengths of all nodes, which is calculated by summing $\sum (i-1) l_i$ over all levels i, where l_i is the no. of nodes on level i



$IPL = \sum (i-1) l_i$

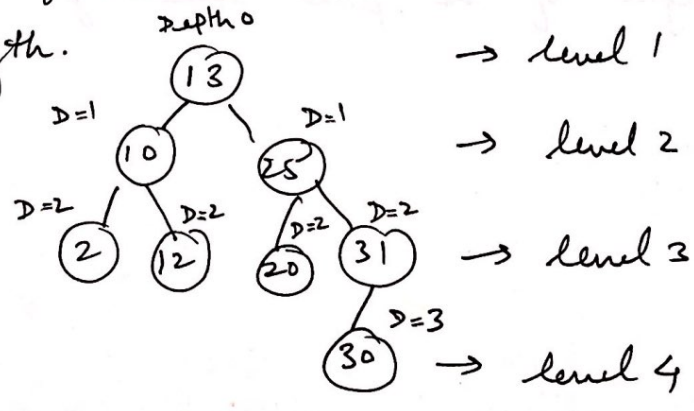
$IPL = (1-1)1 + (2-1)2 + (3-1)4 + (4-1)1$

$= 0 + 2 + 8 + 3 = 13$

OR.

NODE NO	Path length
13	0
10	1
25	1
2	2
12	2
20	2
31	2
29	3
	13

The depth of a node in the tree is determined by the path length.

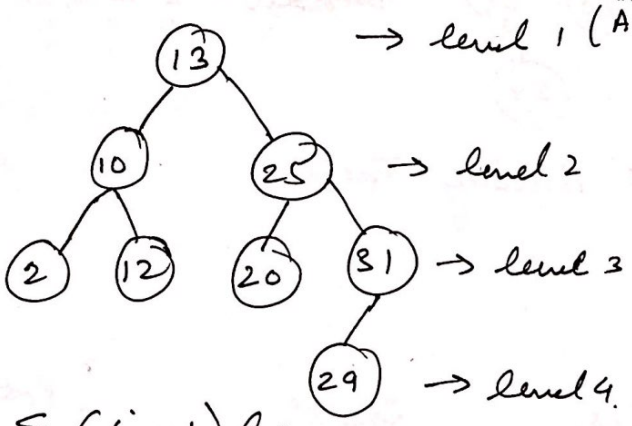


∴ height = 4. (max. level).

Not in course w.e.f 2019-20 (CCBCS)

Internal Path length (IPL)

IPL is the sum of all path lengths of all nodes, which is calculated by summing $\sum (i-1) l_i$ over all levels i, where l_i is the no. of nodes on level i



$IPL = \sum (i-1) l_i$

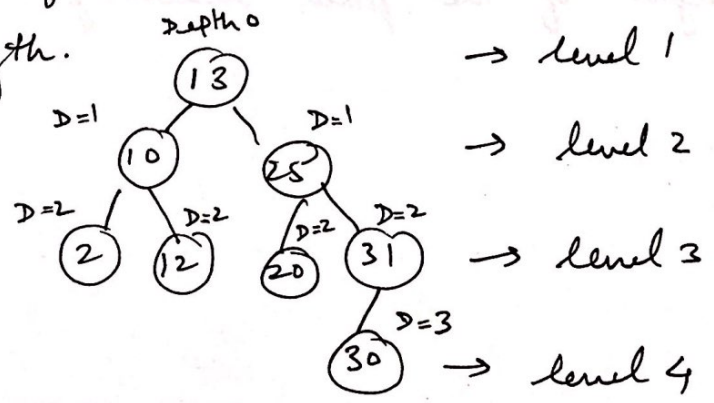
$IPL = (1-1)1 + (2-1)2 + (3-1)4 + (4-1)1$

$= 0 + 2 + 8 + 3 = 13$

OR.

NODE NO	Path length
13	0
10	1
25	1
2	2
12	2
20	2
31	2
29	3
	13

The depth of a node in the tree is determined by the path length.



∴ height = 4. (max. level).

Average path length (Average depth) is given by the formula $\frac{IPL}{n}$. (Here n, is the total no. of nodes in tree)

(I) worst case :- In worst case, tree turns into a linked list & search takes 'n' units of time.

i.e $path_{worst} = \frac{1}{n} \sum_{i=1}^{n} (i-1) = \frac{n-1}{2} = O(n)$.

$n \rightarrow$ height or no. of nodes

[$i \rightarrow$ level no.]

[$l_i \rightarrow$ no. of nodes on level 'i'] [l_i is always equal to 1].

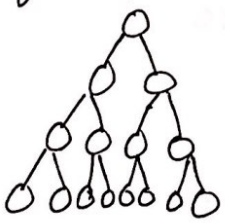
$n \rightarrow$ height of levels or no. of nodes in worst case.

'n' is total no. of nodes in tree.

(II) Best case :- Best case occurs when all leaves in the tree of height 'h' are in atmost two levels & only nodes in the next to last level can have one child.

few results :-

1). IPL of Complete Binary tree ($h=4$). $= \sum_{i=1}^{h-1} i \cdot 2^i$



$IPL = \sum (i-1) l_i$

$= (1-1)1 + (2-1)2 + (3-1)4 + (4-1)8$

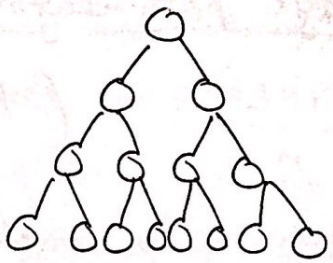
$= 0 + 2 + 8 + 24 = 34.$

OR

$\sum_{i=1}^{h-1} i \cdot 2^i = 1.2 + 2.2^2 + 3.2^3$ [$h=4$].

$= 2 + 8 + 24 = 34$

(2) $\sum_{i=1}^{h-1} 2^i = 2^h - 2.$



$LH.S = \sum_{i=1}^{3} 2^i = 2^1 + 2^2 + 2^3 = 14.$

R.H.S. $2^h - 2 = 2^4 - 2 = 14.$

(3) IPL of complete Binary tree :-

$(h-2)2^h + 2$ eg :- $(4-2)2^4 + 2 = \underline{34}.$

(4) (a) No. of nodes in complete Binary tree = $2^h - 1$

$$\therefore Path_{best} = \frac{IPL}{n} = \frac{(h-2)2^h + 2}{2^h - 1} \approx h - 2$$

(n ↓ no. of nodes)

Thus, one-half of the nodes are in the leaf level with path length $h-1$.

Also, in complete Binary tree,

height, $h = \log_2(n+1)$, so

$$path_{best} = \log_2(n+1) - 2$$

Average path length in a perfectly balanced tree is

$$\lceil \log_2(n+1) \rceil - 2 = O(\log_2 n)$$

```
//PROGRAM TO IMPLEMENT BINARY SEARCH TREE.....

#include<iostream.h>
#include<conio.h>
#include<process.h>

const int size=20;    //size of stack and queue...
int count_leaf=0;
int count_nonleaf=0; //No. of leaf and non-leaf nodes initialized...


class node
{
  public:
       int info;
       node* left;
       node* right;
}*root=NULL;        //made global due to use in function main()


class bst
{
      node *p,*q,*p1;
      public:

 /*    bst()              //in comments because made global due to  use of
       {                 //root in function main().....
            root=NULL;
       }  */

       void insert();
       void recursive_inorder(node*);
       void recursive_preorder(node*);
       void recursive_postorder(node*);
       void iterative_inorder(node*);
       void iterative_postorder(node*);
       void iterative_preorder(node*);
       void bfs_traversal(node*);
       void mirror_image(node*);
       void count_leaf_nonleaf_totalnodes_usingpreorder(node*);
       int height_of_tree();
       void treeheight(node*,int&);
       void search_tree(node*,int);
       void delete_by_copying(node*,int);
       void delcopy(node*&);
       void delete_by_merging(node*,int);
       void delmerge(node*&);
};
```

I/P — 13, 10, 25, 2, 12, 20, 31, 29

Inorder :- 2, 10, 12, 13, 20, 25, 29, 31

Preorder :- 13, 10, 2, 12, 25, 20, 31, 29

Postorder :- 2, 12, 10, 20, 29, 31, 25, 13

```
void main()
{
  clrscr();
  int n,h,el;
  bst b1;              //Binary search tree object...
  char ch='y';

  do
  {
  cout<<"\n1.INSERT:";
  cout<<"\n2.RECURSIVE INORDER:";
  cout<<"\n3.RECURSIVE PREORDER:";
  cout<<"\n4.RECURSIVE POSTORDER:";
  cout<<"\n5.ITERATIVE INORDER:";
  cout<<"\n6.ITERATIVE PREORDER:";
  cout<<"\n7.ITERATIVE POSTORDER:";
  cout<<"\n8.BREADTH FIRST TRAVERSAL:";
  cout<<"\n9.MIRROR IMAGE(BREADTH FIRST TRAVERSAL):";
  cout<<"\n10.DELETION BY COPYING METHOD:";
  cout<<"\n11.DELETION BY MERGING METHOD:";
  cout<<"\n12.NO. OF LEAF,NONLEAF AND TOTAL NO. OF NODES:";
  cout<<"\n13.HEIGHT OF TREE IS:";
  cout<<"\n14.SEARCHING A NODE IN TREE:";
  cout<<"\n\n ENTER UR CHOICE:";
  cin>>n;

  switch(n)
  {
       case 1:  b1.insert();
                break;

       case 2:  cout<<"\nRECURSIVE INORDER IS:\n";
                b1.recursive_inorder(root);
                break;

       case 3:  cout<<"\nRECURSIVE PREORDER IS:\n";
                b1.recursive_preorder(root);
                break;

       case 4:  cout<<"\nRECURSIVE POSTORDER IS:\n";
                b1.recursive_postorder(root);
                break;

       case 5:  cout<<"\nITERATIVE INORDER IS:\n";
                b1.iterative_inorder(root);
                break;

       case 6:  cout<<"\nITERATIVE PREORDER IS:\n";
                b1.iterative_preorder(root);
                break;

       case 7:  cout<<"\nITERATIVE POSTORDER IS:\n";
                b1.iterative_postorder(root);
                break;
```

```
        case 8:  cout<<"\nBREADTH FIRST SEARCH TRAVERSAL IS:\n";
                 b1.bfs_traversal(root);
                 break;

        case 9:  cout<<"\nBFS TRAVERSAL OF MIRROR IMAGE  IS:\n";
                 b1.mirror_image(root);
                 break;

        case 10: cout<<"\nENTER THE NODE VALUE YOU WANT TO DELETE: ";
                 cin>>el;
                 b1.delete_by_copying(root,el);
                 break;

        case 11: cout<<"\nENTER THE NODE VALUE YOU WANT TO DELETE: ";
                 cin>>el;
                 b1.delete_by_merging(root,el);
                 break;

        case 12: b1.count_leaf_nonleaf_totalnodes_usingpreorder(root);
                 cout<<"\nTOTAL NO. OF LEAF NODES ARE: "<<count_leaf;
                cout<<"\nTOTAL NO. OF NON-LEAF NODES ARE: "<<count_nonleaf;
                cout<<"\nTOTAL NO. OF NODES ARE:";
                cout<<(count_leaf+count_nonleaf);
                count_leaf=count_nonleaf=0;//cleared to 0 for next selection
                break;

        case 13: h=b1.height_of_tree();
                   cout<<"\nHEIGHT OF TREE IS: "<<h; //height starting from 1
                 break;

        case 14: cout<<"\nENTER THE ELEMENT U WANT TO SEARCH: ";
                 cin>>el;
                 b1.search_tree(root,el);
                 break;

        default: cout<<"\nENTER VALID CHOICE";
       }

  cout<<"\nWANT TO CONTINUE: ";
  cin>>ch;
 }while(ch=='y' || ch=='Y');

     getch();
}
```

```
class stack
{
      node* a[size];
      public:
        int top;
        stack()
        {
             top=-1;
        }
        void push(node*);
        node* pop();
};

void stack::push(node* ele)
{
  if(top!=size-1)
       {
             top++;
             a[top]=ele;
       }
  else
       cout<<"\nstack is full\n";
}


node* stack::pop()
{
  if(top!=-1)
  {
       return a[top--];
  }
  else
        {
        cout<<"\nstack is empty\n";
        return NULL;
        }
}
```

```
class queue
{
          node* a[size];
          int front,rear;
     public:
          queue()
          {
               front=rear=-1;
          }

          void enqueue(node*);
          node* dequeue();
          int isfull()
          {
            return((front==0 && rear==size-1) ||front==rear+1);
          }
          int isempty()
          {
               return(front==-1);
          }
};

void queue::enqueue(node* temp1)
{
     if(isfull())
       cout<<"\nqueue is full";
     else if(rear==-1)
               front=rear=0;
     else if(rear==size-1)
          rear=0;
     else
          rear++;
          a[rear]=temp1;
}

node* queue::dequeue()
{
     node* r;
     if(isempty())
       {
          cout<<"\nqueue is empty";
          return NULL;
       }
     else
       {
          r=a[front];
          if(front==rear)
               front=rear=-1;
          else if(front==size-1)
                front=0;
          else front++;
       }
       return r;
}
```

```
void bst::insert()
{
      int i;
      cout<<"\nenter node value and 0 to quit:";
      cin>>i;
      while(i!=0)
      {
            if(root==NULL)
             {
                   root=new node;
                   root->left=0;root->right=0;
                   root->info=i;
             }
             else
             {
                  p=root;
                  q=new node;
                  q->left=0;q->right=0;
                  q->info=i;

                  while(p!=0)          //find position to insert new node
                  {
                    if(i < p->info)
                    {
                         p1=p;
                         p=p->left;
                    }
                    else
                    {
                        if(i>= p->info)
                        {
                          p1=p;
                          p=p->right;
                        }
                    }
                  }

                  if(i<p1->info)       //new node inserted
                         p1->left=q;
                  else
                         p1->right=q;
             }
              cout<<"\nenter node value or 0 to quit:";
              cin>>i;
      }
  }
```

```
void bst::recursive_inorder(node* temp)
{
     if(temp!=0)
     {
          recursive_inorder(temp->left);
          cout<<temp->info<<" ";
          recursive_inorder(temp->right);
     }
}

void bst::recursive_preorder(node* temp)
{
     if(temp!=0)
     {
          cout<<temp->info<<" ";
          recursive_preorder(temp->left);
          recursive_preorder(temp->right);
     }
}

void bst::recursive_postorder(node* temp)
{
     if(temp!=0)
     {
          recursive_postorder(temp->left);
          recursive_postorder(temp->right);
          cout<<temp->info<<" ";
     }
}

void bst::iterative_preorder(node* temp)
{
     stack s1;
     if(temp!=NULL)
     {
          s1.push(temp);
          while(s1.top!=-1)
          {
               temp=s1.pop();
               cout<<" "<<temp->info;
               if(temp->right!=NULL)
                    s1.push(temp->right);
               if(temp->left!=NULL)
                    s1.push(temp->left);
          }
     }
}
```

```
void bst::iterative_inorder(node* temp)
{
      stack s1;

      while(temp!=NULL)
      {
            while(temp!=NULL)
            {
              if(temp->right!=NULL)
                        s1.push(temp->right);
              s1.push(temp);
              temp=temp->left;
            }
            temp=s1.pop();
            while(s1.top!=-1 && temp->right==NULL)
            {
                  cout<<" "<<temp->info;
                  temp=s1.pop();
            }
            cout<<" "<<temp->info;
            if(s1.top!=-1)
                   temp=s1.pop();
            else
                   temp=NULL;
      }
}

void bst::iterative_postorder(node* temp)
{
      stack s1;
      q=temp;
      while(temp!=NULL)
      {
        for( ;temp->left!=NULL;temp=temp->left)
                s1.push(temp);
        while(temp!=NULL && (temp->right==NULL || temp->right==q))
        {
              cout<<" "<<temp->info;
              q=temp;
              if(s1.top==-1)
                    return;
              temp=s1.pop();
        }
        s1.push(temp);
        temp=temp->right;
      }
}
```

```
void bst::bfs_traversal(node* temp)
{
  queue q1;
  if(temp!=NULL)
  {
        q1.enqueue(temp);
        while(!q1.isempty())
              {
                    temp=q1.dequeue();
                    cout<<" "<<temp->info;
                    if(temp->left!=NULL)
                          q1.enqueue(temp->left);
                    if(temp->right!=NULL)
                          q1.enqueue(temp->right);
              }
  }
}

void bst::mirror_image(node* temp)
{
  queue q1;
  if(temp!=NULL)
  {
        q1.enqueue(temp);
        while(!q1.isempty())
              {
                    temp=q1.dequeue();
                    cout<<" "<<temp->info;
                    if(temp->right!=NULL)
                          q1.enqueue(temp->right);
                    if(temp->left!=NULL)
                          q1.enqueue(temp->left);
              }
  }
}

void bst::count_leaf_nonleaf_totalnodes_usingpreorder(node* temp)
{
          if(temp!=NULL)
          {
                if(temp->left != NULL || temp->right != NULL)
                      count_nonleaf++;
                if(temp->left == NULL && temp->right == NULL)
                      count_leaf++;
          count_leaf_nonleaf_totalnodes_usingpreorder(temp->left);
          count_leaf_nonleaf_totalnodes_usingpreorder(temp->right);
      }
}
```

```
int bst::height_of_tree()
{
  int height=0;
  treeheight(root,height);
  return (height);
}

void bst::treeheight(node* temp,int& ht) //'ht' is ampersand var,changes
made in 'height' also...
{
      int hleft=0;
      int hright=0;
      if(temp!=NULL)
       {
            treeheight(temp->left,hleft);
            treeheight(temp->right,hright);
            if(hleft>=hright)
                  ht=hleft+1;
            else
                  ht=hright+1;
            return;
       }
}

void bst::search_tree(node* temp,int el)
{
  int flag=0;             //set if element is found else remain cleared...
  while(temp!=NULL)
      {
        if(el==temp->info)
              {
                    flag=1;
                    break;
              }
        else if(el<temp->info)
                    temp=temp->left;
        else
                    temp=temp->right;

      }
      if(flag==1)
             cout<<"\nNODE FOUND";
      else
             cout<<"\n NODE NOT FOUND";
}
```

Morris Transversal

Traversal through tree transformation (Morris)

(1) Traversal through Recursion or Iteration → uses stack.

(2) Traversal without stack → Threaded trees.

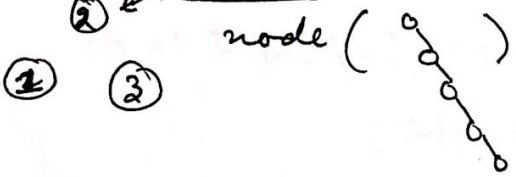
Also nodes are extended by one field to make distinction where the right pointer is for right child or successor. (two fields for left & right pointers).

(3) Traversal without stack or Threaded Trees — MORRIS_Inorder.

make some changes in the tree structure & assign new values to the nodes.

The tree temporarily loose its structure which is later restored.

Inorder traversal of a tree that has no left child in any node ( ) is reduced from ①②③ to ②③.



∴ transform the tree so that the node being processed has no left child.

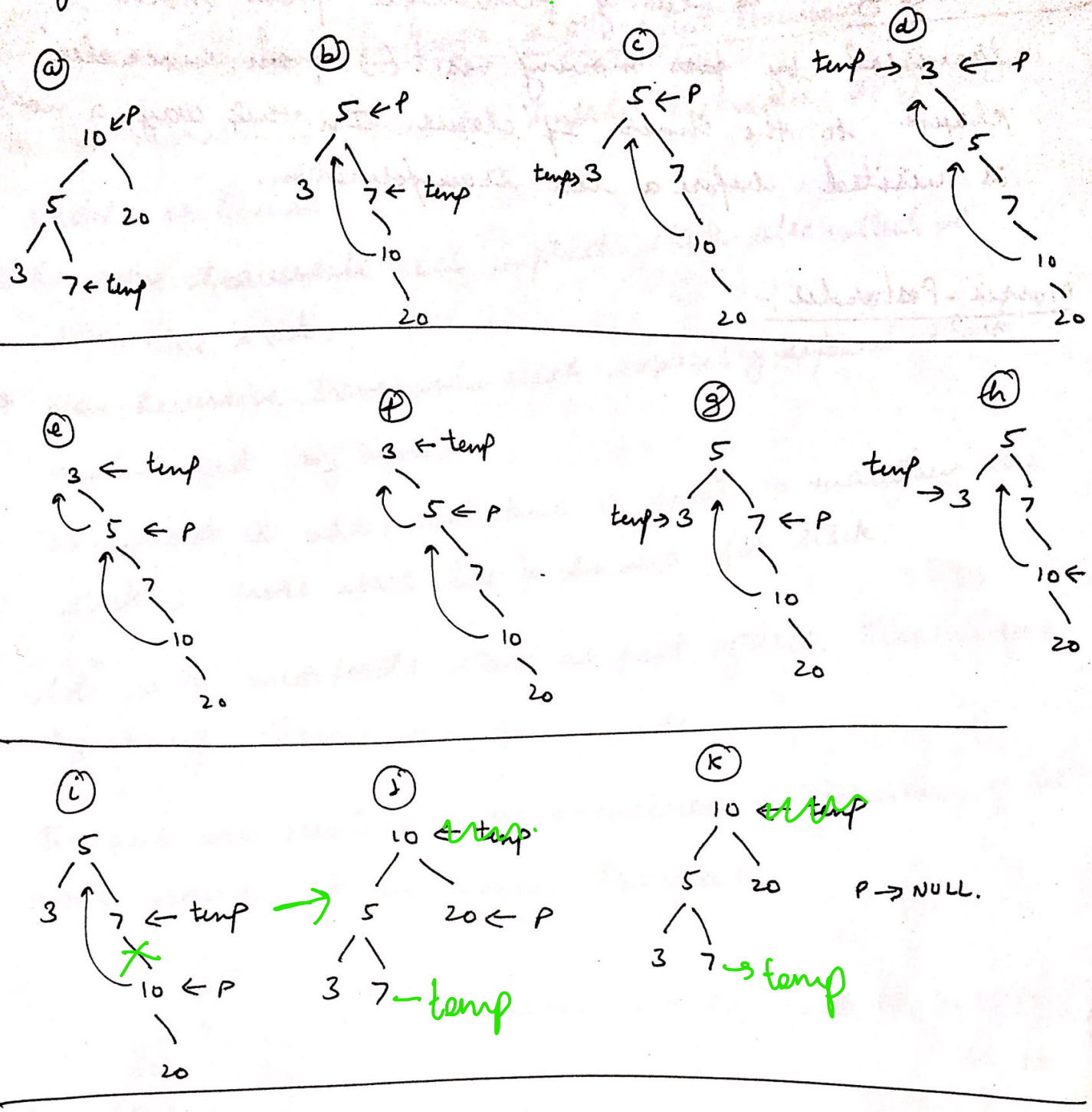
Algorithm:-

```
Morris_Inorder()
    while not finished (i.e. P != 0).
        if node has no left descendant
            visit it; (Print the value).
            go to the right;
        else
            make this node right child of the rightmost node
              in its left descendant;
            go to the left descendant;
```

The Algorithm successfully traverse the tree, but only once, because it destroys its original structure. ∴ some info has to be retained to allow the tree to restore its original form. This is achieved by retaining the left pointer of the node moved down its right subtree.

```
Morris_Inorder (node* p)          // p = root.
{ node * temp;
  while ( p != 0)
    {
      If (p -> left == 0)
        {
          visit (p);   // Print it
          p = p -> right;  // move p to right.
        }
      else
        { temp = p -> left;
          while( temp -> right != 0 && temp->right != p)
                temp = temp -> right;
          If ( temp -> right == 0)
            {
                temp -> right = p;
                p = p -> left;
            }
          else
            {
              visit(p);
              temp -> right = 0;
              p = p -> right;
            }
        }
    }
}
```

eg:- Morris-Inorder



Morris Preorder :- is easily obtainable from inorder traversal by ~~this~~ moving visit() from inner else clause to the inner If clause. In this way, a node is visited before a tree transformation.

6.4.3. Stackless Depth-first Traversal

The traversals (Preorder, Inorder, Postorder, BFS) uses stacks or Queues.

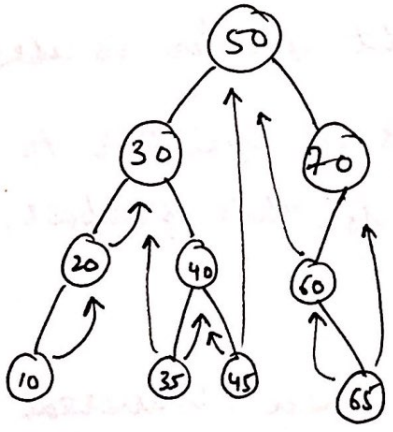
Recursive traversals uses implicit stack. also called as run-time stack.

Non-Recursive traversals uses explicitly defined stack maintained by user.

Drawback is additional time is spent to maintain the stack & more space has to be used for stack.

Soln is to incorporate stack as part of tree. This is done by using threads in a given node.

Threads are pointers to the predecessor & successor of the node according to an inorder traversal.



Inorder :- 10, 20, 30, 35, 40, 45, 50, 60, 65, 70

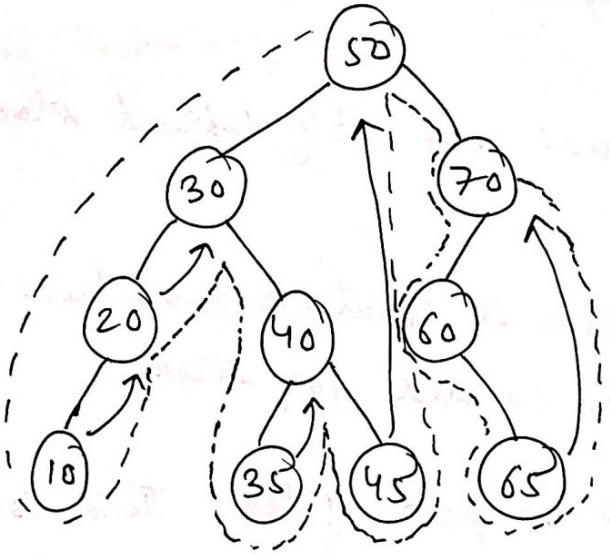
Trees whose nodes use threads are called threaded trees.

Pointers can be overloaded. eg:- left pointer is used to point to left child but if left child doesn't exist it is used to point to its predecessor. eg:- Node (60)

same is the case with Right pointer that can also point to successor

In above figure, pointer to both predecessors & successors are maintained.

we can also use only one thread. for eg:- pointers to successors only.



Inorder:-

10, 20, 30, 35, 40, 45, 50, 60, 65, 70.

P = 10, 20, 30, 35, 40, 45, 50, 60, 65, 70

only one variable 'P' is needed to traverse the tree. No stack is needed thus space is saved.

only one thing is important. That is node requires a data member indicating how the right pointer is used. Is it pointer to right child or is it pointer to successor. A Flag (0 or 1) is used for this purpose.

Preorder traversal using threads :-

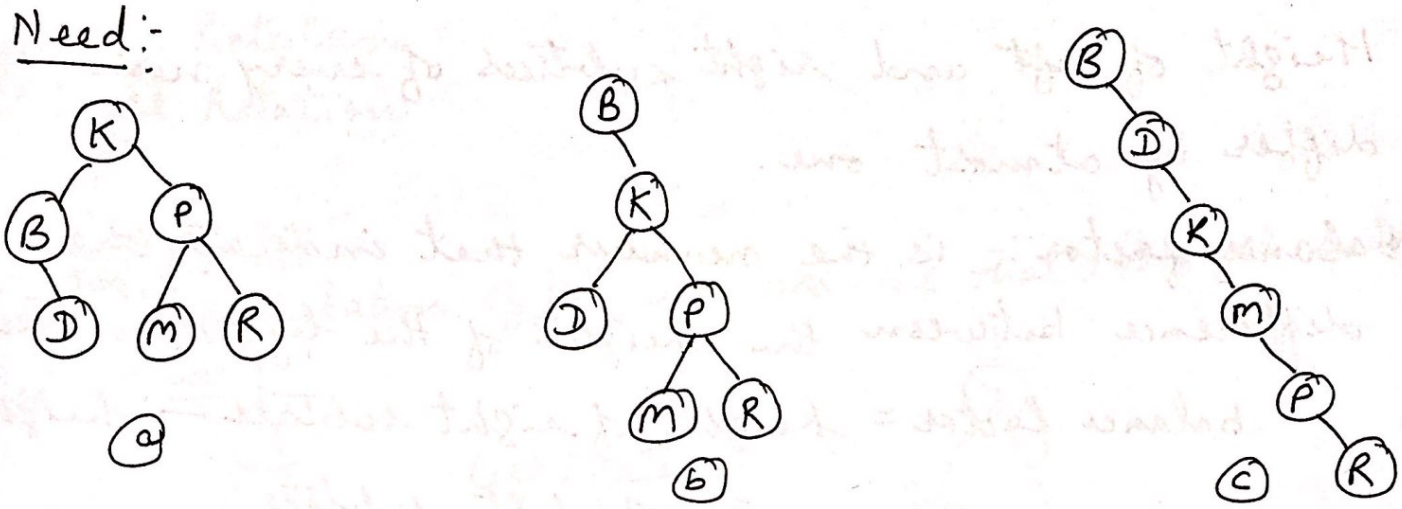
The current node is visited first & then traversal continues with its left descendent, if any, or right descendant, if any.

If the current node is a leaf, threads are used to go through the chain of its already visited inorder successors to restart traversal with the right descendant of the last successor.

6.7. Balancing a Tree.

Ref 1 - Drozdek.

Need:-



Comparisons in worst case:-

Tree (a) - 3
Tree (c) - 6

] However data is the same.

Def^{n}:- A binary tree is height balanced (or balanced) if the difference in height of both subtrees of any node in the tree is either zero or one (or -1).

Perfectly balanced :- A tree is perfectly balanced if it is balanced and all leaves are to be found on one level or two levels.

Consider Tree (b).

(2-1=1)

At node K ;- The diff. in height of subtree is 1. (acceptable)

At node B :- The diff. in height of subtree is 3 i.e (3-1)

This means entire tree is unbalanced.

6.7.2 AVL Trees (Adelson-Vel'skii and Landis)

also known as admissible tree.

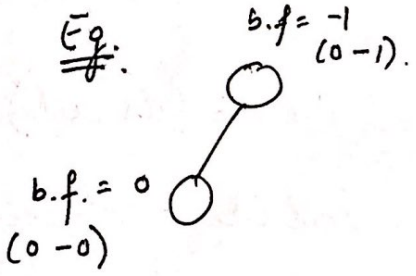
Height of left and right subtrees of every node differ by atmost one.

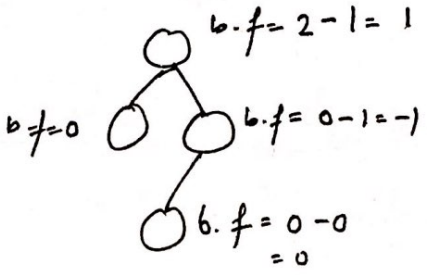
balance factor :- is the number that indicate the difference between the heights of the left & right subtrees.

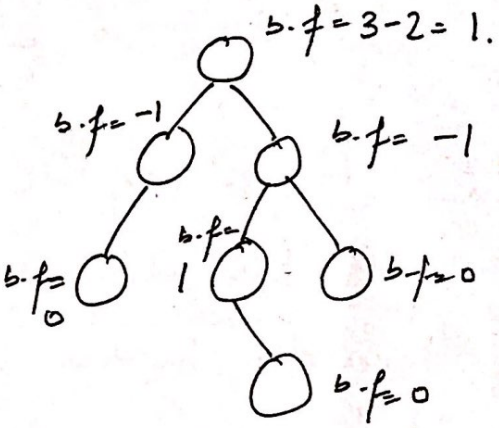
balance factor = height of right subtree − height of left subtree.

Thus the range is +1, 0, −1.

Eg:







Rotations (height adjustments)

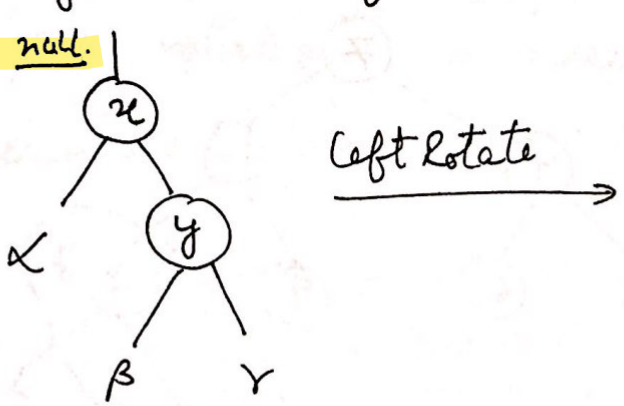
Two Types :-

(I) Left Rotation
(II) Right Rotation

Pe L or R child ke L or R sub tree me daala hai.

(I) Left Rotate

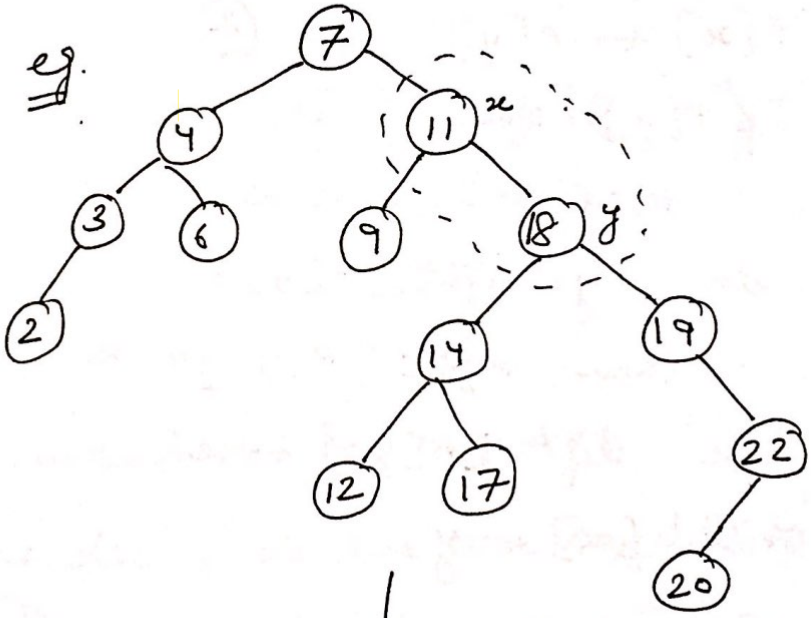
To do left Rotation of 'x', we assume that its right child 'y' is not null.



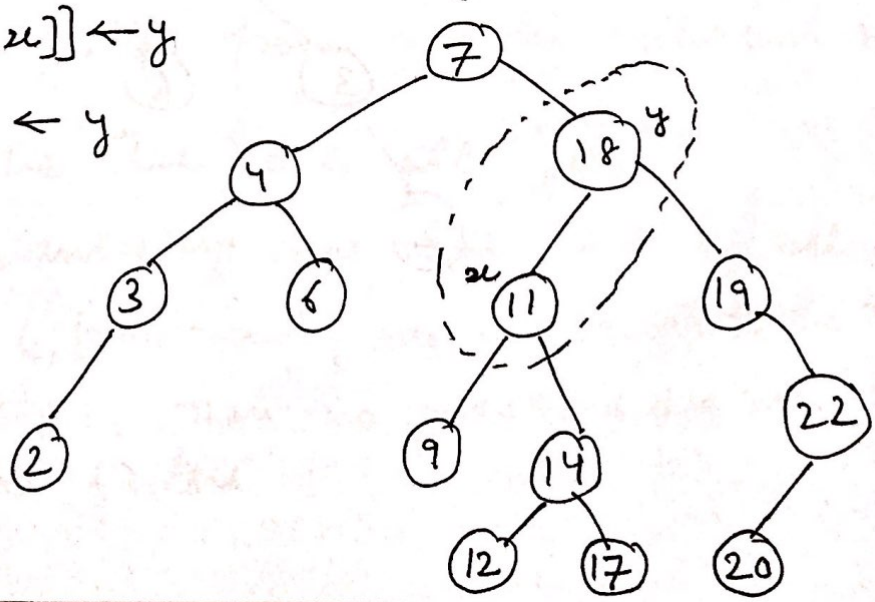
Left Rotate (T, x)

1. y ← right [x]
2. right [x] ← left [y]
3. P [left [y]] ← x
4. P [y] ← P [x]
5. if P [x] == null
6. then root [T] ← y
7. else if x == left [P [x]]
8. then left [P [x]] ← y
9. else right [P [x]] ← y
10. left [y] ← x
11. P [x] ← y.

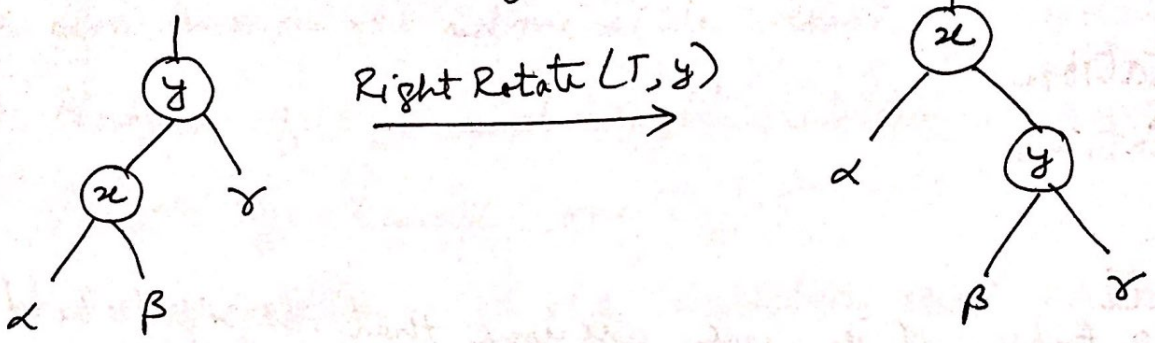
eg.



Left Rotate (T, x)



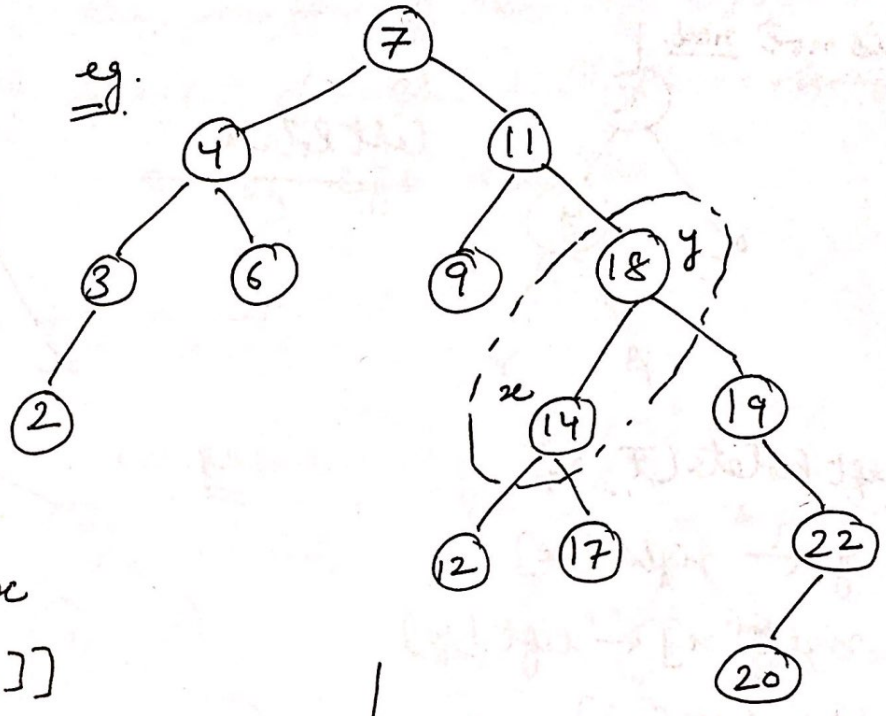
(II) **Right Rotate** : To do right rotation on 'y', we assume that its left child 'x' must not be null.



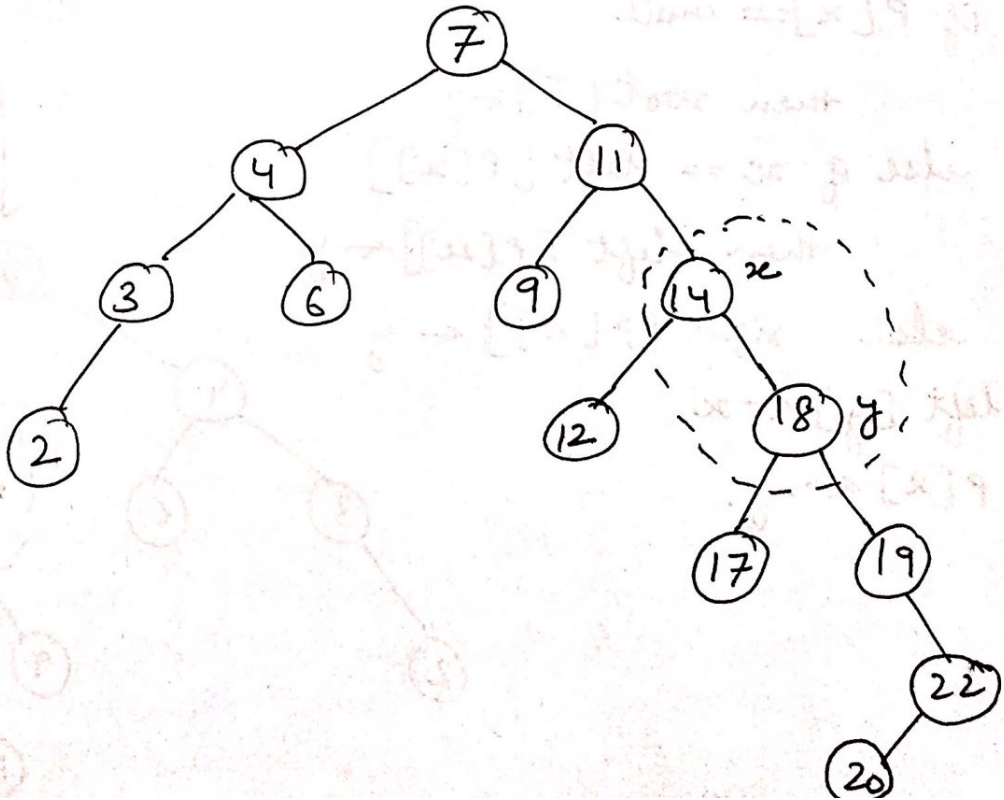
Right Rotate (T, y)

1. $x \leftarrow$ left[y]
2. left[y] $\rightarrow$ right[x]
3. P[right[x]] $\leftarrow$ y
4. P[x] $\leftarrow$ P[y]
5. If P[y] == NULL
6. then root[T] $\leftarrow$ x
7. else if y = right[P[y]]
8. then right[P[y]] $\leftarrow$ x
9. else left[P[y]] $\leftarrow$ x
10. right[x] $\leftarrow$ y
11. P[y] $\leftarrow$ x

eg:



Right Rotate (T, y)

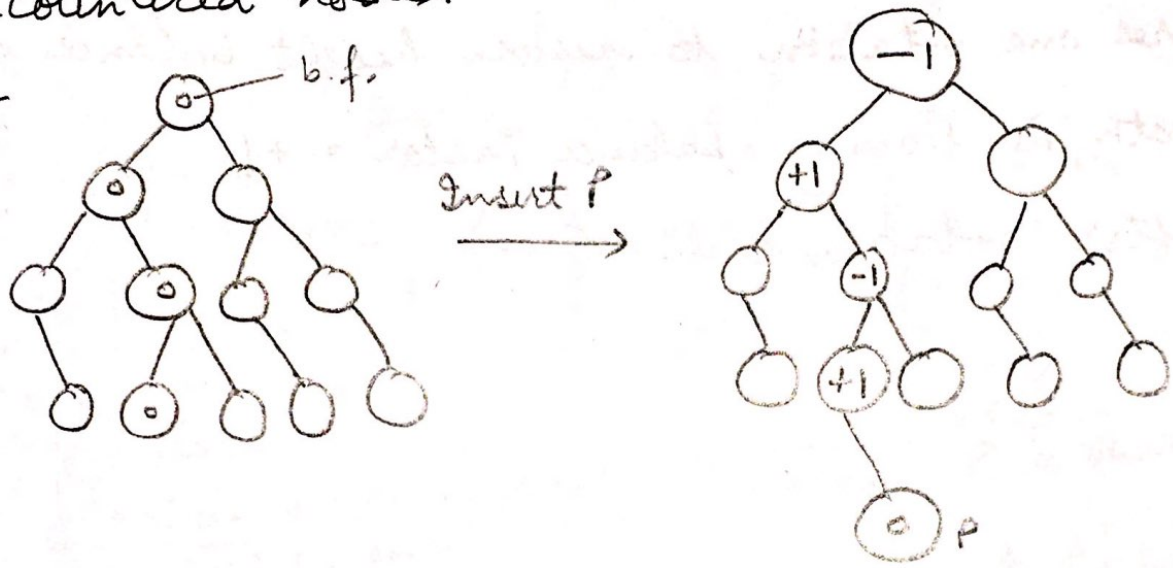


Initial case to be considered :-

(A) Assumption I

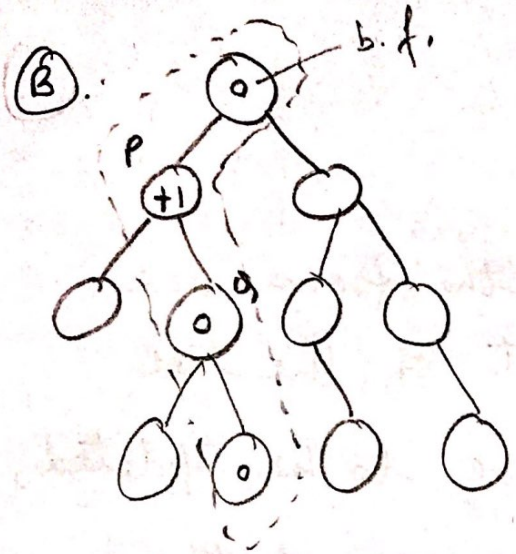
If the balance factors on the path from the newly inserted node to the root of the tree are all 'zero', all of them have to be updated, but no rotation is needed for any of the encountered nodes.

eg :-

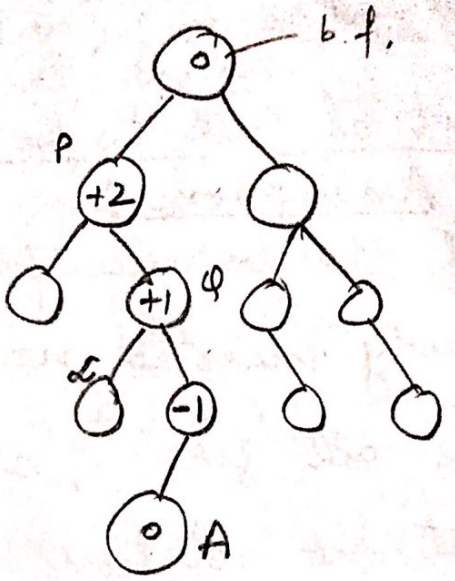


A new node 'P' is inserted in an AVL Tree but no rotation is needed. (or we can say - NO height adjustments are needed) because balance factor on the path from newly inserted node to the root of the tree are all 'zero'.

Assumption II: If after inserting new node, b.f. of all nodes along the path (from newly inserted node to the root) lies between 0, 1 & -1, then no rotations are needed. Simply update balance factors.

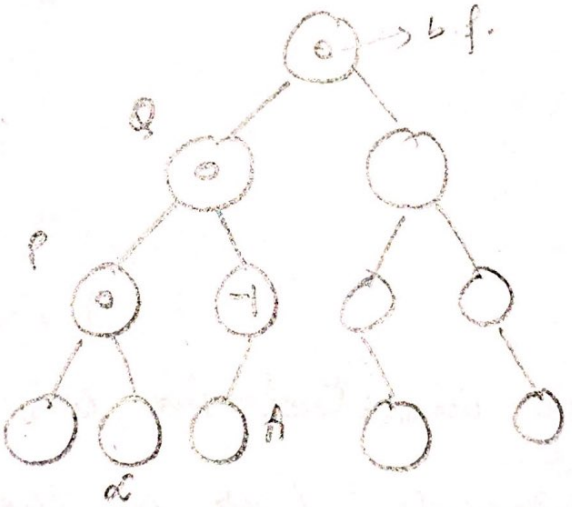


Insert (A)



requires one rotation to restore height balance since the path is having balance factor = +1.

Thus after rotation (left rotion).



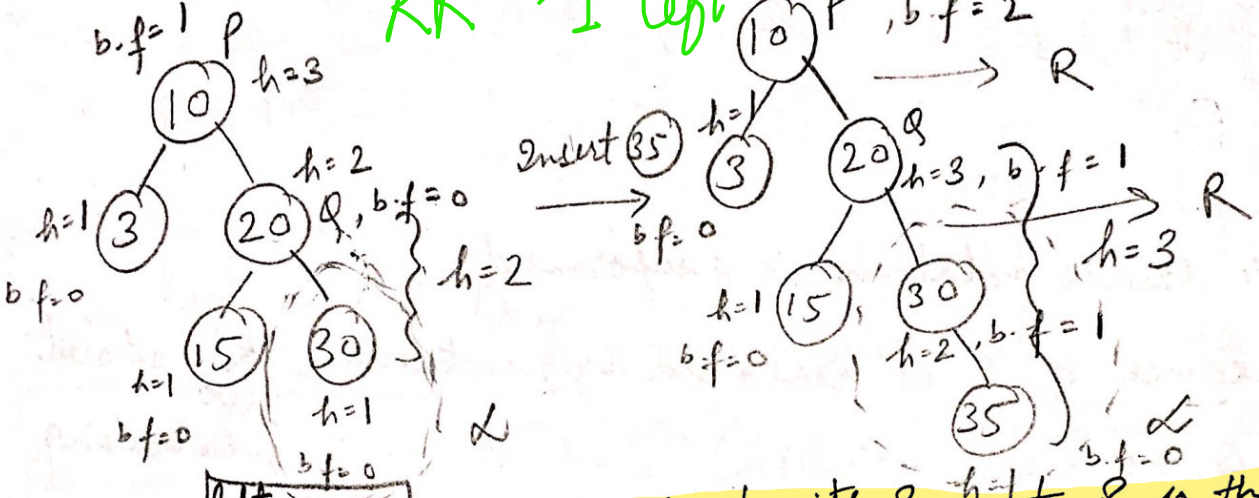
Proceed to cases 1, 2, 3, 4. if Initial **Case B** is to be considered.

If balance factor of any node in an AVL tree becomes less than -1 or greater than 1, the tree has to be balanced.

Insertion in AVL Trees:-

Case I. Inserting a node in right subtree of right child (R) (α)
(Also called RR Rotation)

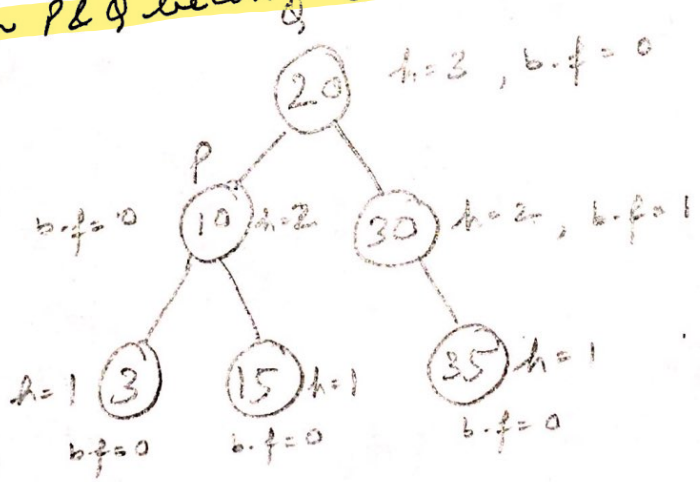
RR → 1 Left Rotation



Sol^n: Left rotate node Q about its parent P so that b.f. of both P & Q becomes zero.

Perform Left Rotation

Also known as RR Rotation.
(Right child - Right subtree)



How to find 'P' node for which bal. factor becomes unacceptable?

Sol^n: Move upward toward the root of the tree from the position in which the new node has been inserted and by updating the bal. factors of node encountered. Nodes having b.f = ±1 will change to ±2. The first node whose b.f. changed will become node P.

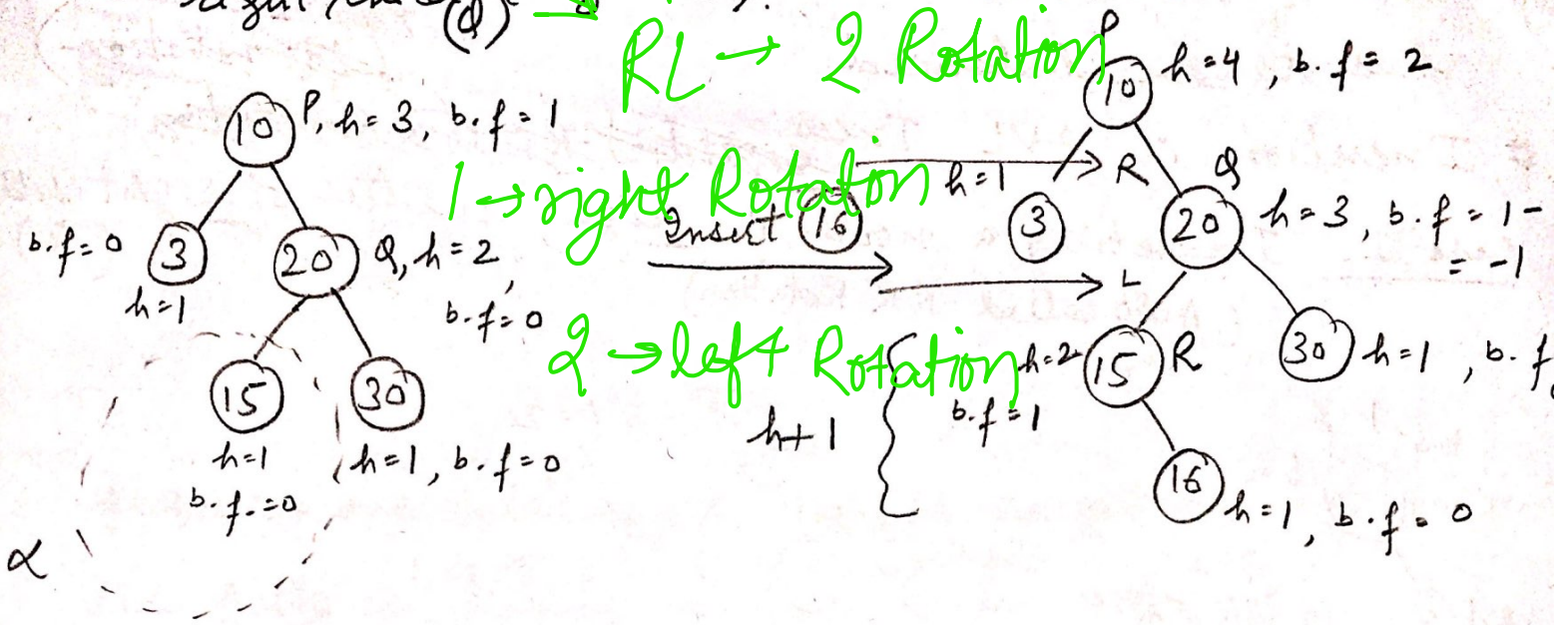
(Right child - Left subtree - RL Rotation)

Case II. Inserting a node in the left subtree of the right child (Q) (eg - α).

RL → 2 Rotations

1 → right Rotation

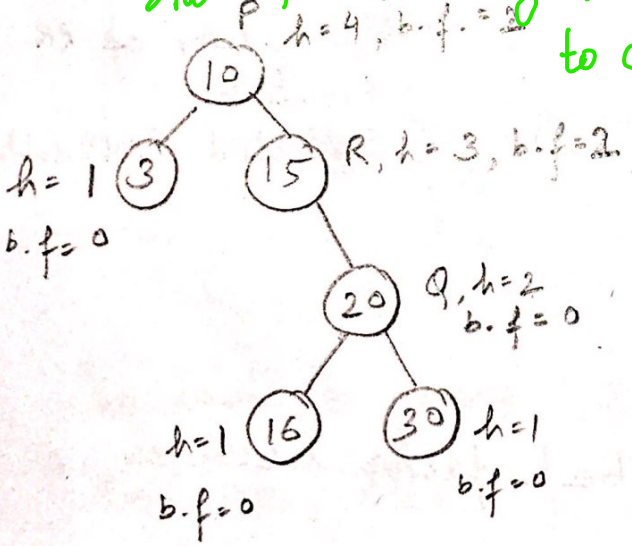
2 → left Rotation



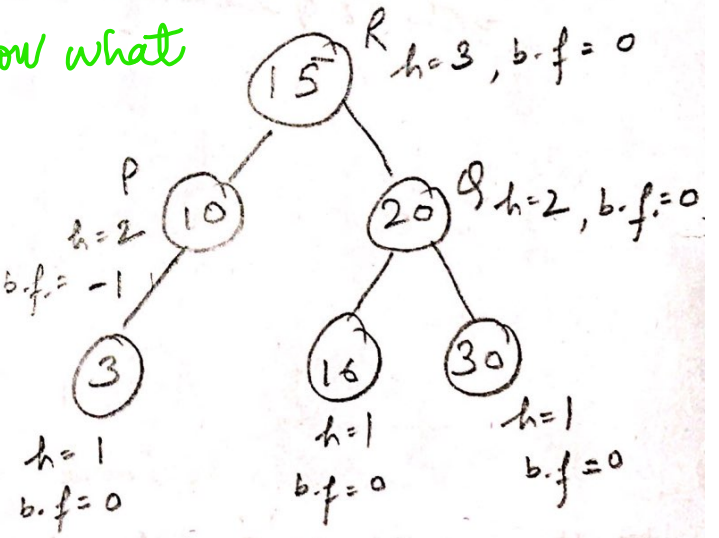
Sol^n:- A double rotation is performed.

(a) The balance of P is restored by rotating 'R' about right node Q. (First rotation) and then by rotating left R again about node P. (second rotation).

↳ Start rotations by base to know what to do first.



(first rotation)

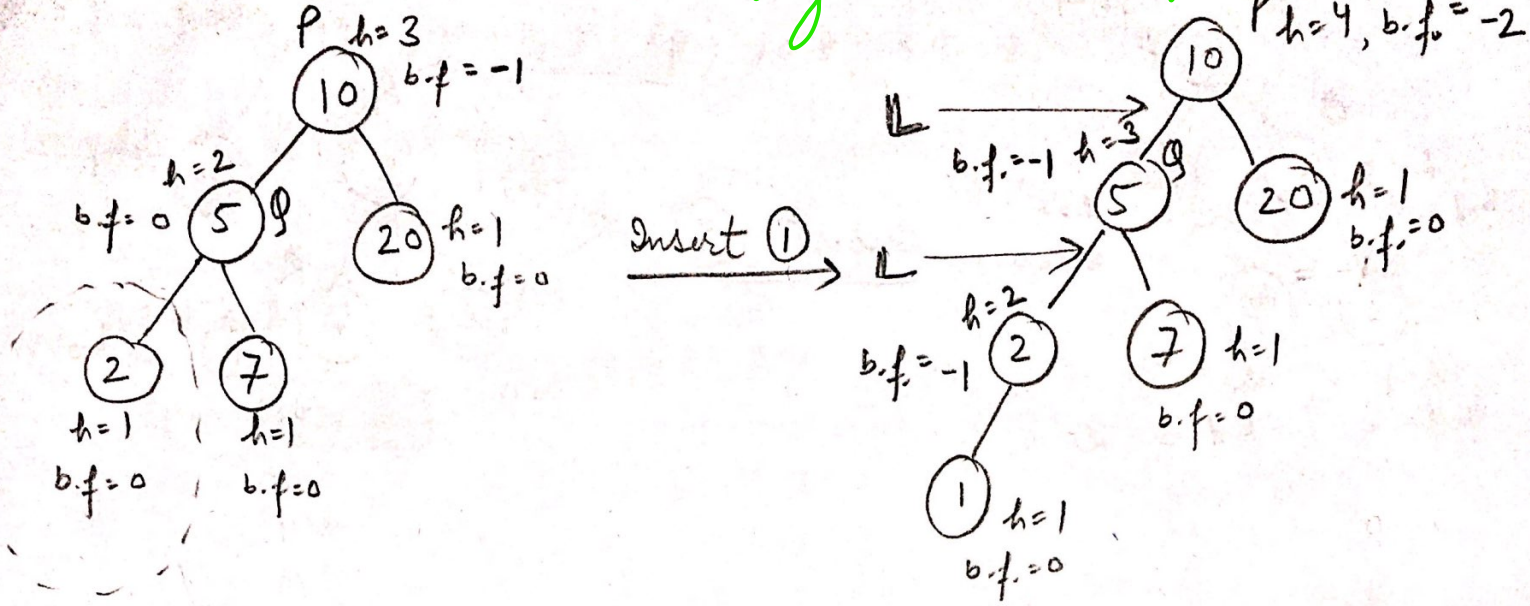


(second rotation)

Also called RL Rotation.

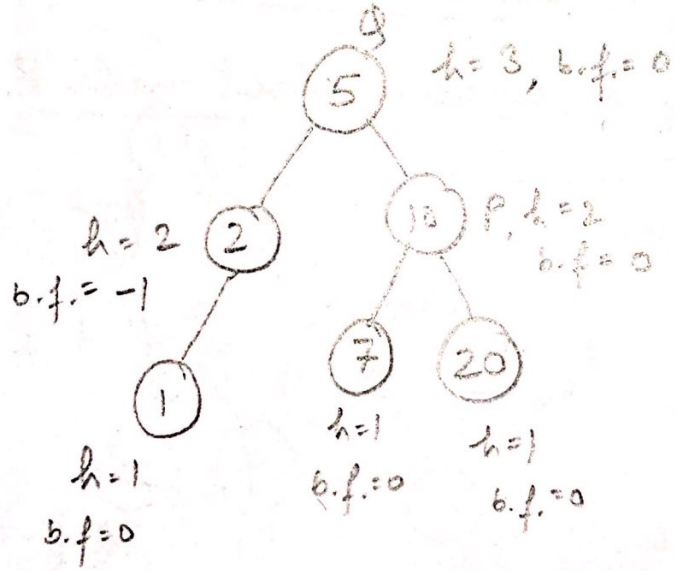
Case III Inserting a node in the left subtree of left child. (α)

LL → L Right rotation



α.

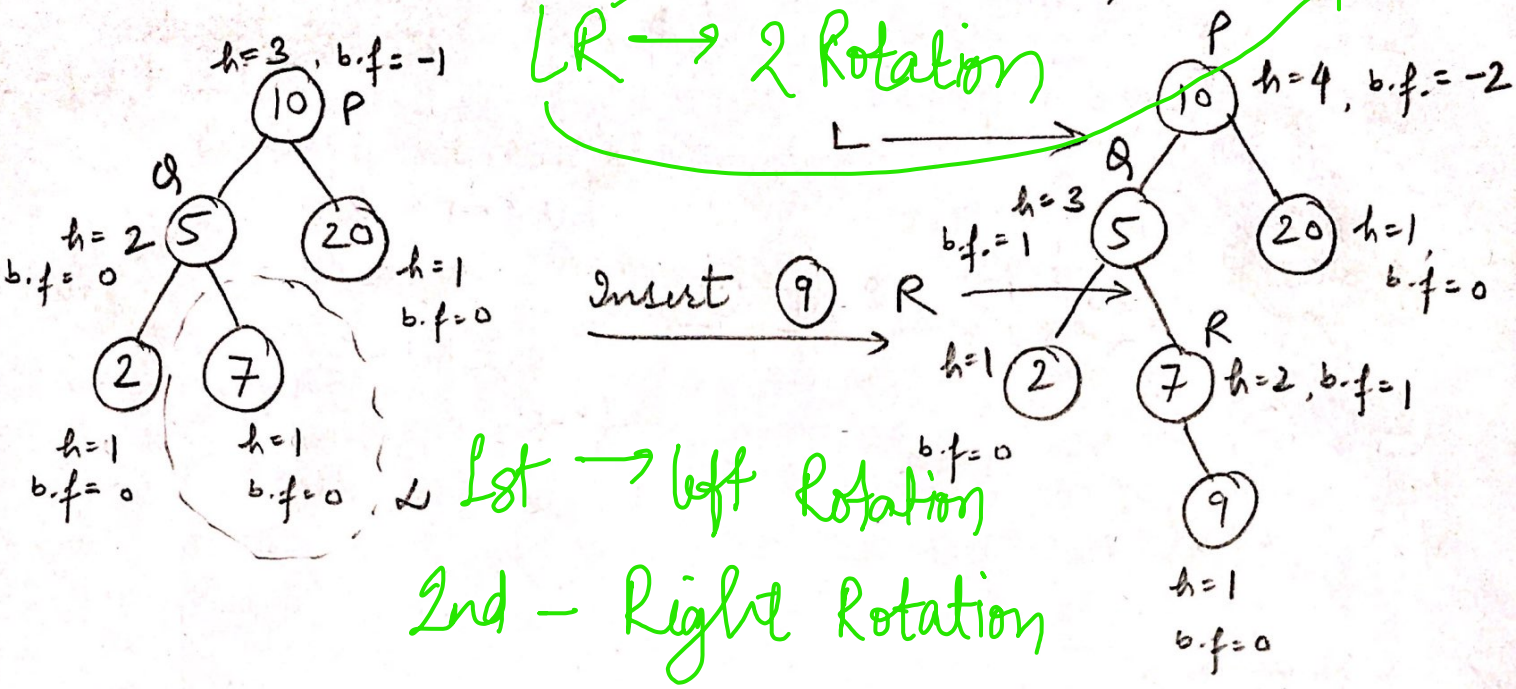
sol^n: Right Rotate node Q about its parent P so that balance factor of both P & Q becomes 0.



#

Right rotation is performed. [Also known as LL Rotation - Left child Left subtree]

Case IV. Inserting a node in the right subtree of left child. (Q)

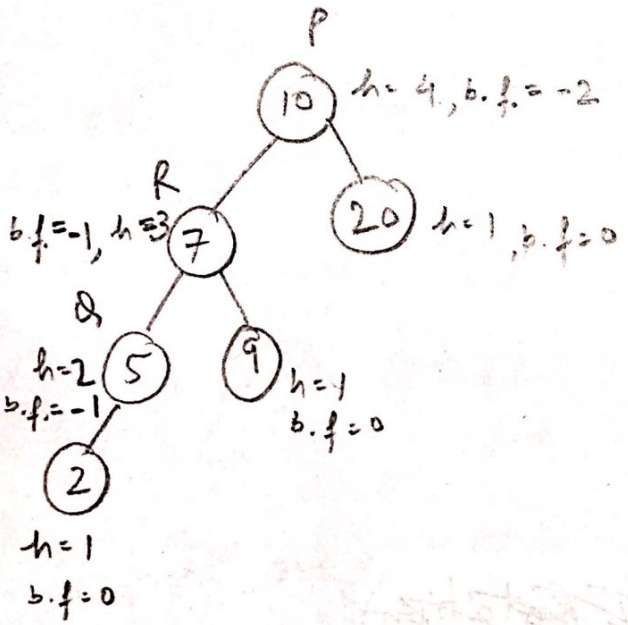


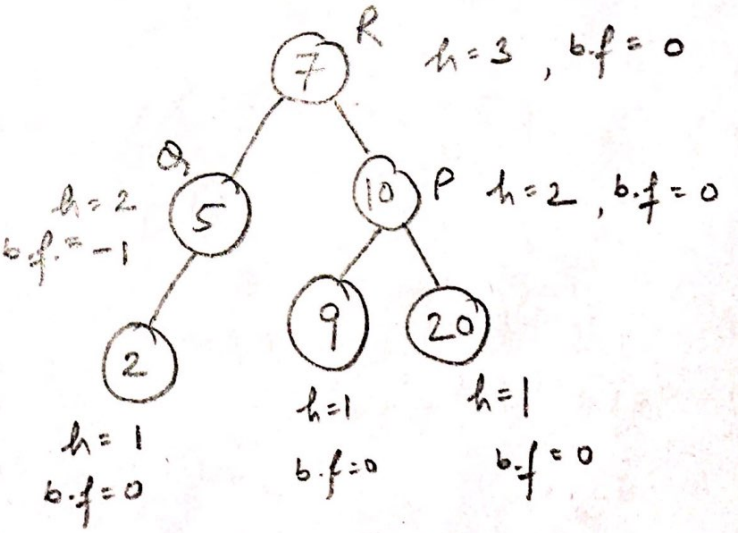
Sol^{n} :- A double rotation is performed.

The balance of P is restored :-

rotation 1 :- By rotating R about node Q, and then

rotation 2 :- By rotating R again about node P.





(first rotation) (second rotation).

Left child - Right subtree (LR Rotation)

Q:- Construct an AVL Tree by inserting the following elements in the given order :-

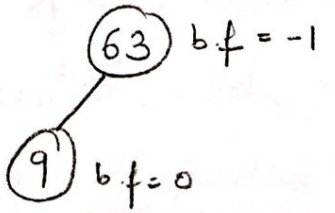
63, 9, 19, 27, 18, 108, 99, 81

Sol^n.

Node (63) b.f. = 0.

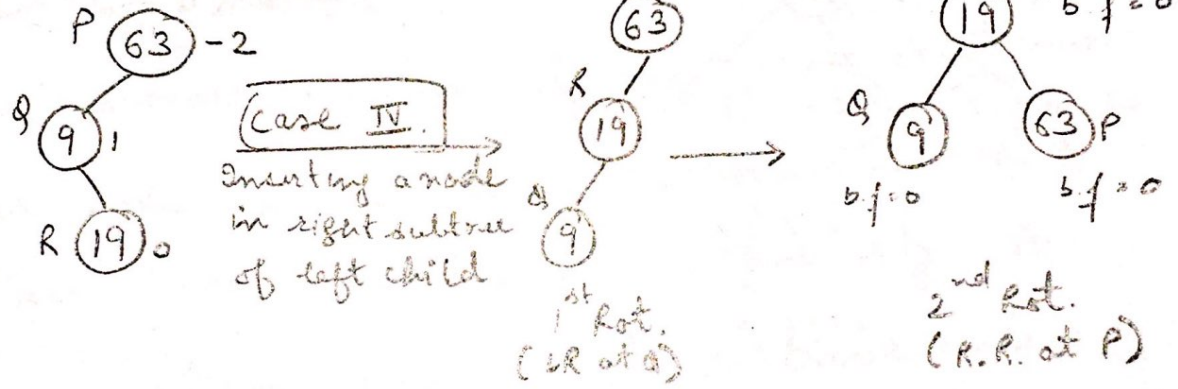
Node (9)

Case A

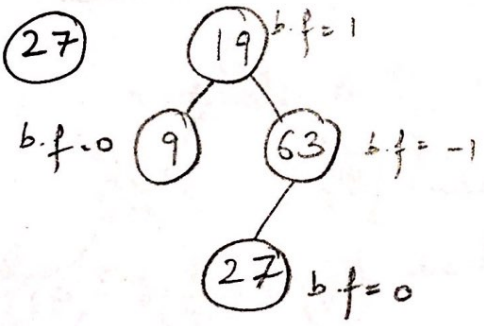


Node (19)

Case B



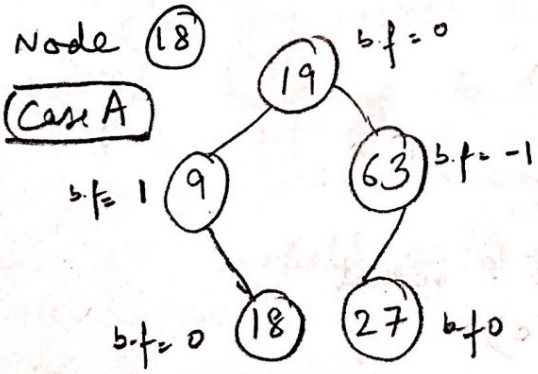
Node (27)



Case A → No Rotations since insertion path contain b.f = 0.

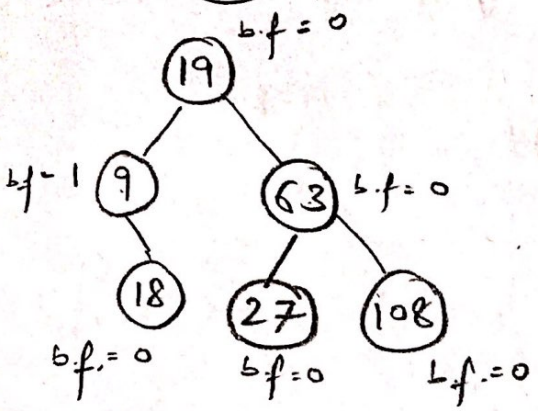
Node (18)

Case A



just update the balance factors.

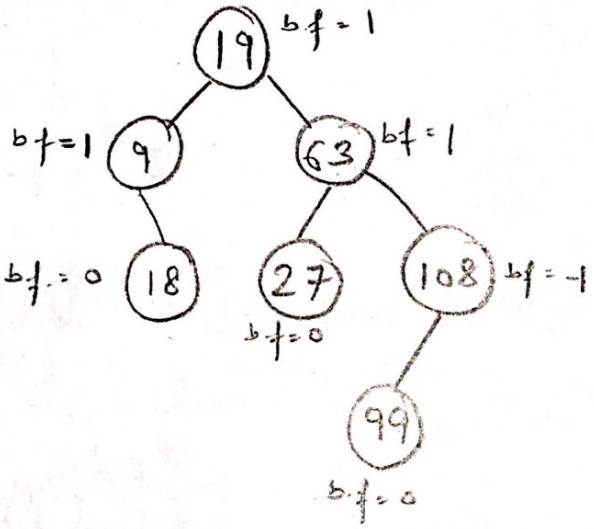
Node (108).



Case - A

No rotations.

Node (99).

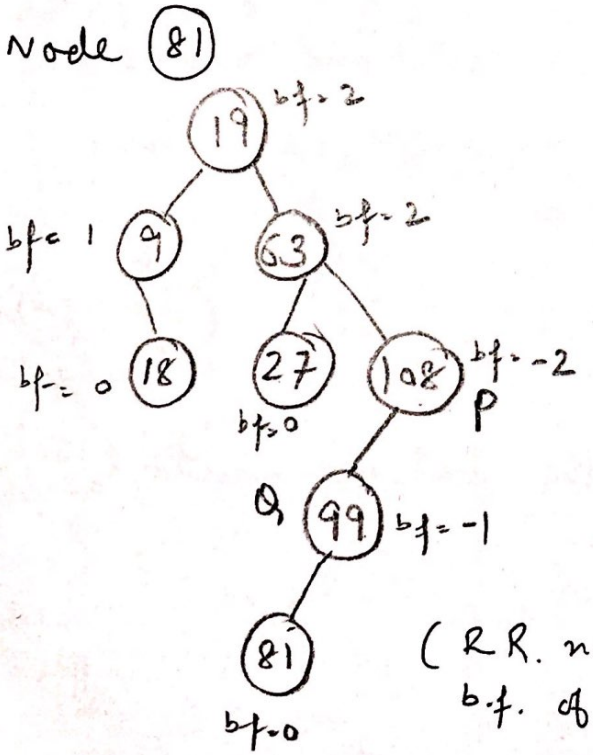


Case A

No rotations.

left child left sub

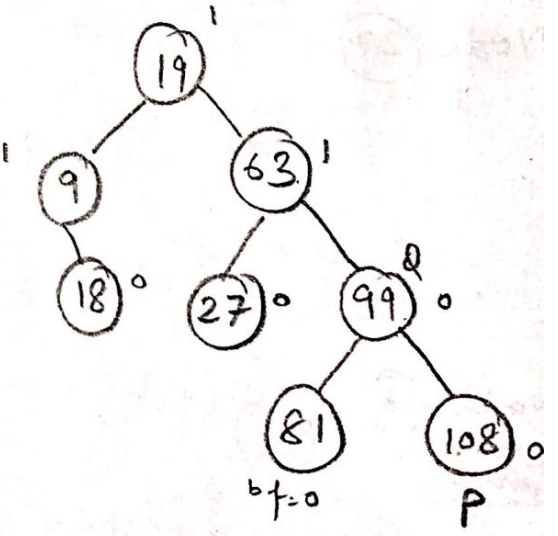
Node (81)



Case B

Inserting a node in left child (Q), left subtree

→ case III



(R.R. node Q about its parent P so that b.f. of both P & Q becomes 0)

Rules

1) After inserting an element, b.f. of only those nodes will be affected that lies on the path from the newly inserted node to the root node.

2) Check the b.f. of only those nodes that lies on the path from newly inserted node to the root. There is no need to check the b.f. of every node.

3) After inserting a node in AVL Tree,

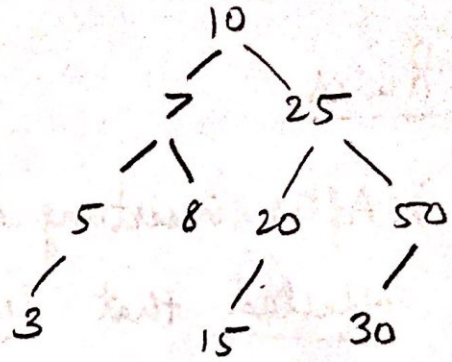
- If tree becomes imbalanced, then there exist one particular node in the tree by balancing which the entire tree becomes balanced automatically. To rebalance the tree, balance that particular node.

(i) To find that particular node, traverse the path from newly inserted node towards the root. Check the b.f. of each node.

(ii) The first node that is imbalanced (ie node having b.f $\neq 1, 0, -1$) is the node that needs to be balanced). [Name it as P, its left or right child as Q, and subsequent child in path as R]

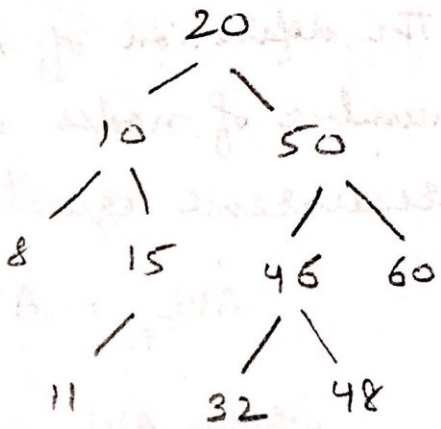
eg:- Construct AVL Tree :-

50, 25, 10, 5, 7, 3, 30, 20, 8, 15



eg:- Construct AVL Tree :-

50, 20, 60, 10, 8, 15, 32, 46, 11, 48



The definition of AVL tree indicates that the minimum number of nodes in a tree is determined by the recurrence equation :-

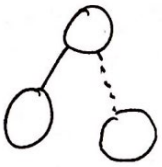
$$AVL_h = AVL_{h-1} + AVL_{h-2} + 1$$

where $AVL_0 = 0$ and $AVL_1 = 1$ are Initial conditions.

Empty AVL Tree i.e. height = 0 , Thus min. no. of nodes = 0. ∴ $AVL_0 = 0$

height = 1 $h=1$ ○ , min. n.o. of nodes = 1 ∴ $AVL_1 = 1$

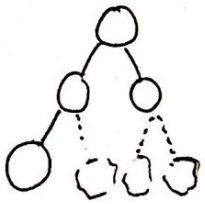
$h = 2$



$$AVL_2 = AVL_1 + AVL_0 + 1$$

$$= 1 + 0 + 1 = 2 \quad \therefore AVL_2 = 2$$

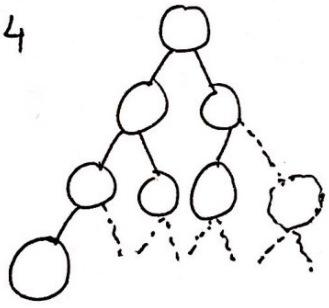
$h = 3$



$$AVL_3 = AVL_2 + AVL_1 + 1$$

$$= 2 + 1 + 1 = 4. \quad \therefore AVL_3 = 4$$

$h = 4$



$$AVL_4 = AVL_3 + AVL_2 + 1$$

$$= 4 + 2 + 1 = 7 \quad \therefore AVL_4 = 7.$$

The formula leads to the following bounds on the height h of an AVL tree depending on the no. of nodes n is

$$\log_2(n+1) \leq h < 1.44 \log_2(n+2) - 0.328$$

(Appendix A.5. - Drozdek)

Therefor 'h' is bounded by $O(\log_2 n)$ and the worst case search requires $O(\log_2 n)$ comparisons.

Deletion in AVL Trees

- Apply deletion by copying and find the actual node to be deleted.
- After deleting the node, update the balance fac. from the parent of deleted node upto the root.
- For each node in this path whose b.f. becomes ±2, a single or double rotation is required.
- Find node 'P' that is encounter in Path (from Parent of deleted node upto root) whose b.f. is ±2.
- The rebalancing does not stop after node P is updated with rotations. (as in insertion cases)(possibility)
- This means deletion leads to atmost $O(\log_2 n)$ rotations in worst case. (every node on path needs rotation)
- Also, deletion of node does not have to necessiate an imediate rotation because it may improve the b.f. of its parent (by changing it from ±1 to 0), but it may also worsen the b.f. of grandparent (by changing it from ±1 to ±2)

Two Types:

1) R deletion :- delete right child of Parent.
2) L deletion :- delete left child of Parent.

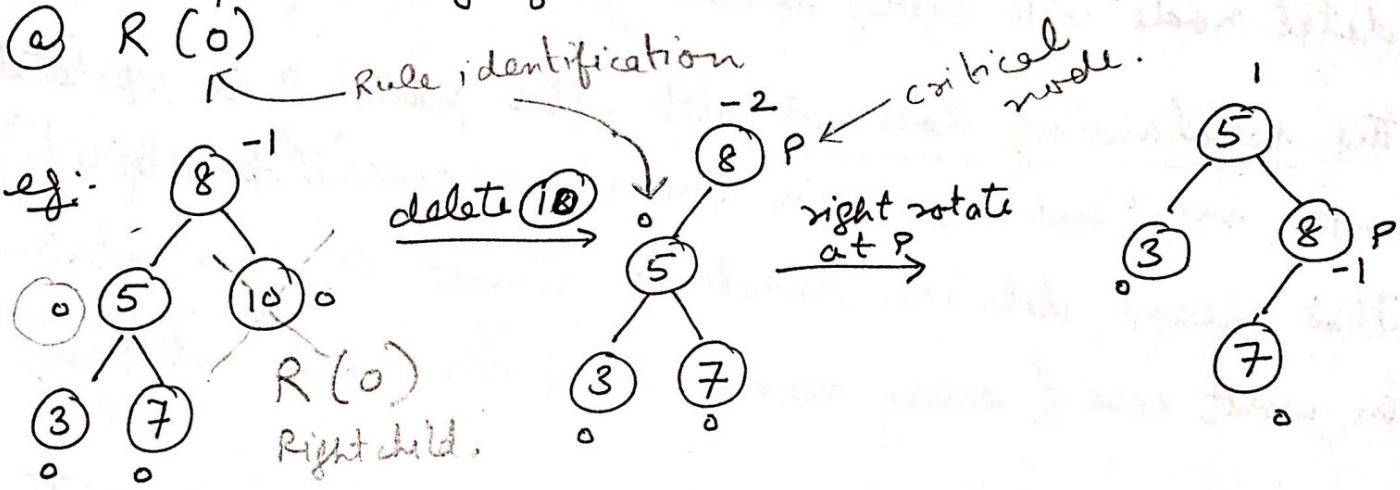
(I) R deletion :- Three subcases :-

R → sibling of deleted node.

ⓐ R (0) → right rotation
ⓑ R (-1) → right rotation
ⓒ R (+1) → Left right rotation.

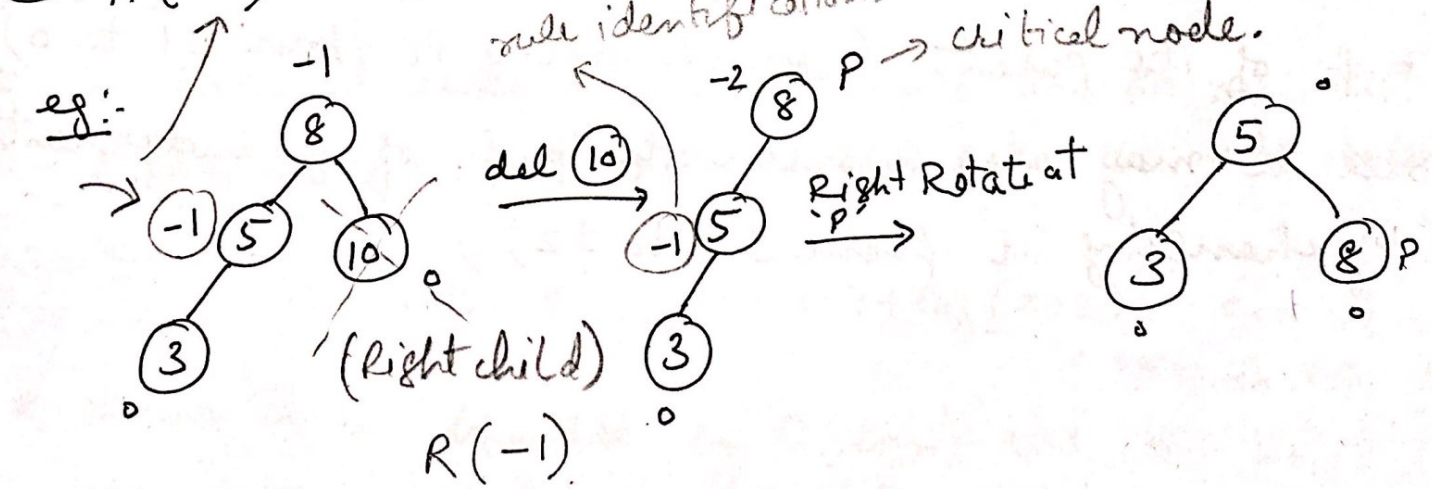
→ sibling of deleted node have b.f. = 0.

ⓐ R (0)



→ sibling of deleted node have b.f. = -1.

ⓑ. R (-1)

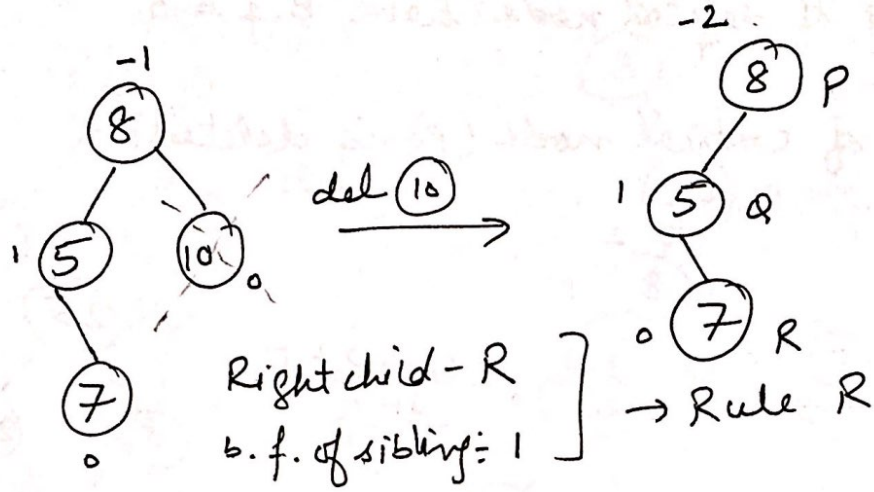


(c). R (+1) → Left Right Rotation.

↳ sibling of deleted node have b.f. = +1.

(i) - Left rotate at 'Q'

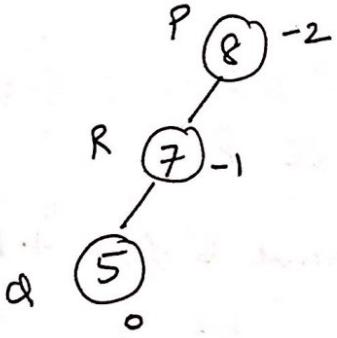
(ii). - Right rotate at 'P'



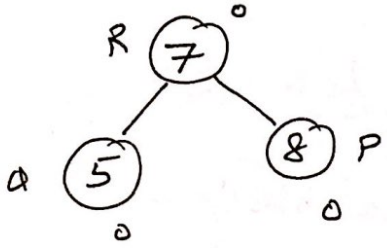
Right child - R
b.f. of sibling = 1

→ Rule R (+1) → LR. Rotation.

(i) Left Rotate at Q



(ii) Right Rotate at P



(II) L deletion :- Three subcases :-

(a) L(0) → Left Rotation

(b) L(+1) → Left Rotation

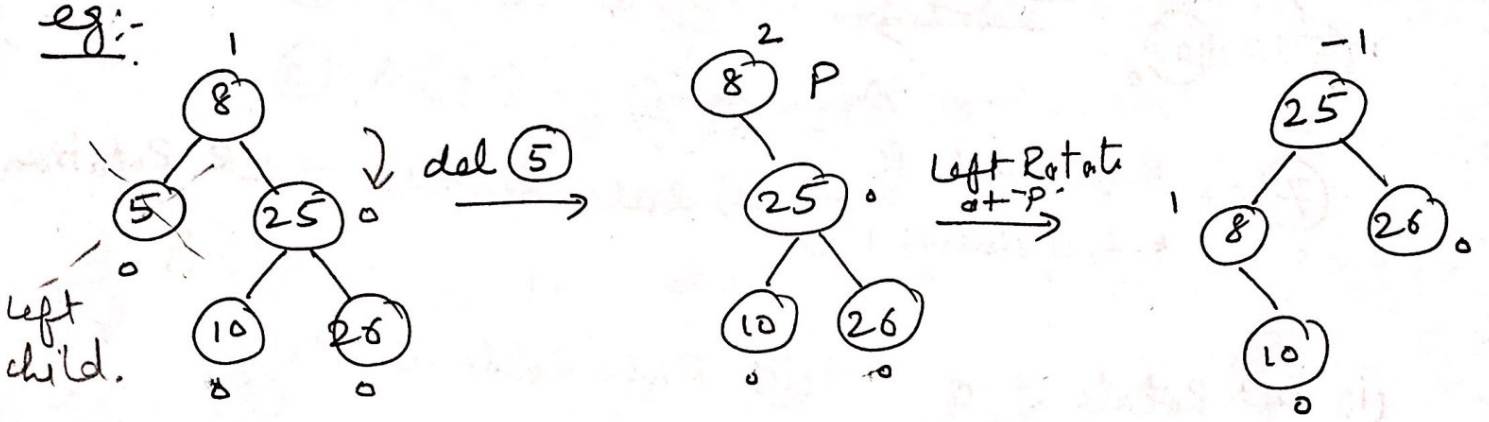
(c) L(-1) → Right Left Rotation.

(a) L(0)

↳ sibling of deleted node have b.f = 0

↳ Left child of critical node (P) is deleted.

eg:-



L - Left child.

b.f. of sibling :- 0

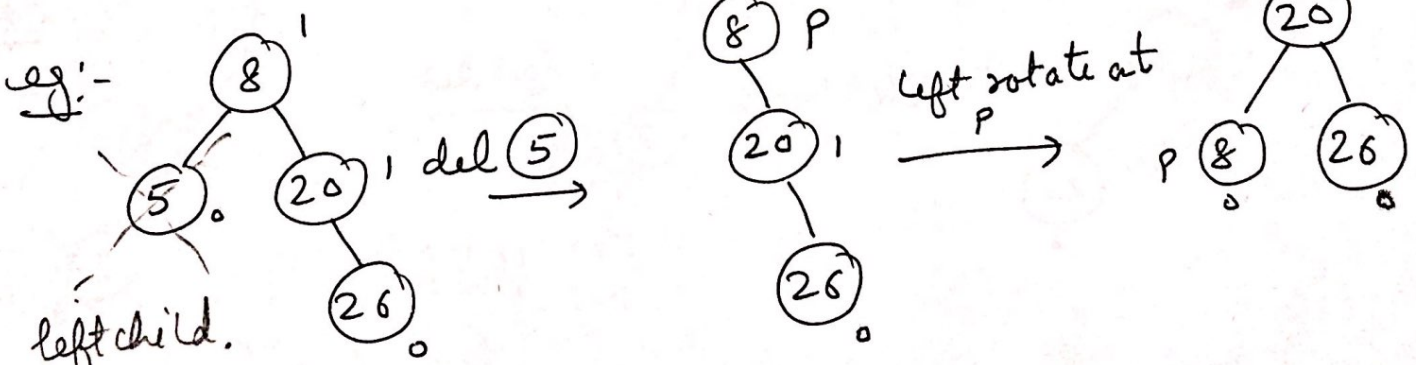
Rule identification :- L(0) → Left Rotate at P.

(b) L (+1) :-

↳ sibling of deleted node have b.f. = +1.

↳ Left child deleted

eg:-



b.f. of sibling = +1

Rule L(+1) :- Left Rotation at P

↳ sibling of deleted node has b.f. = -1

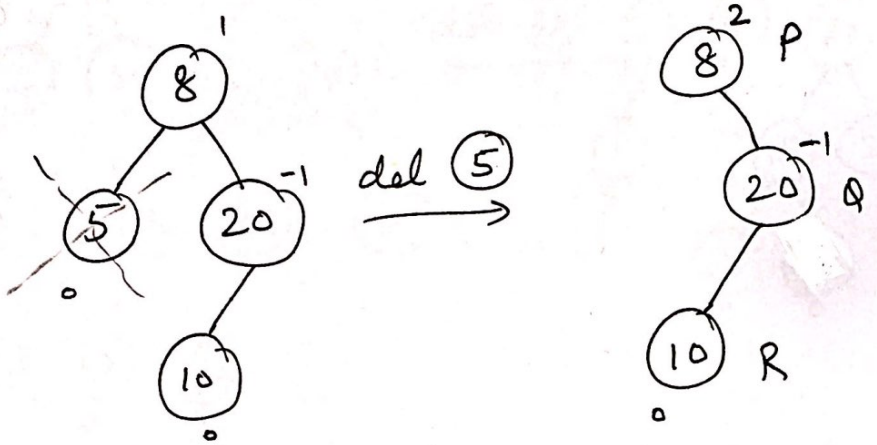
© L(-1) :- Right Left Rotation

↳ left child of 'P' (critical node) deleted.

eg:-

(i) Right Rotate at Q

(ii) Left Rotate at P

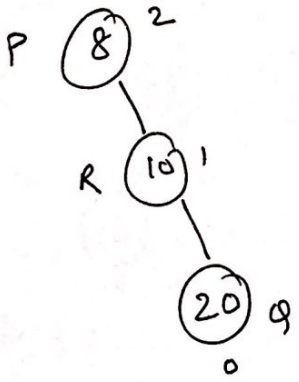


Leftchild - L.

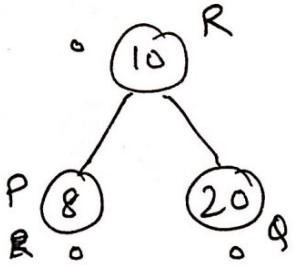
Sibling's b.f:- -1

Rule:- L(-1)

(i) Right Rotate at 'Q'

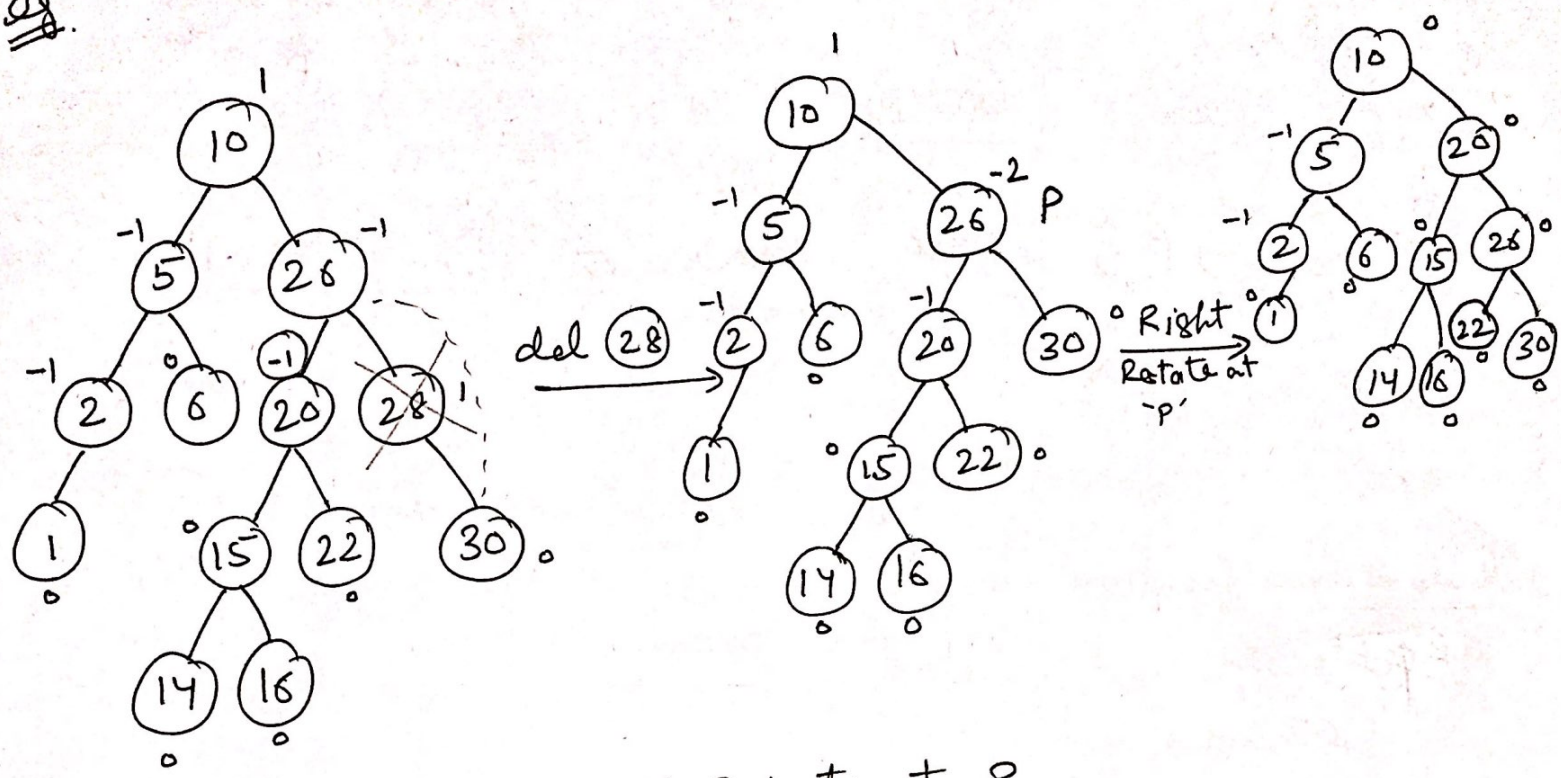


(ii) Left Rotate at P



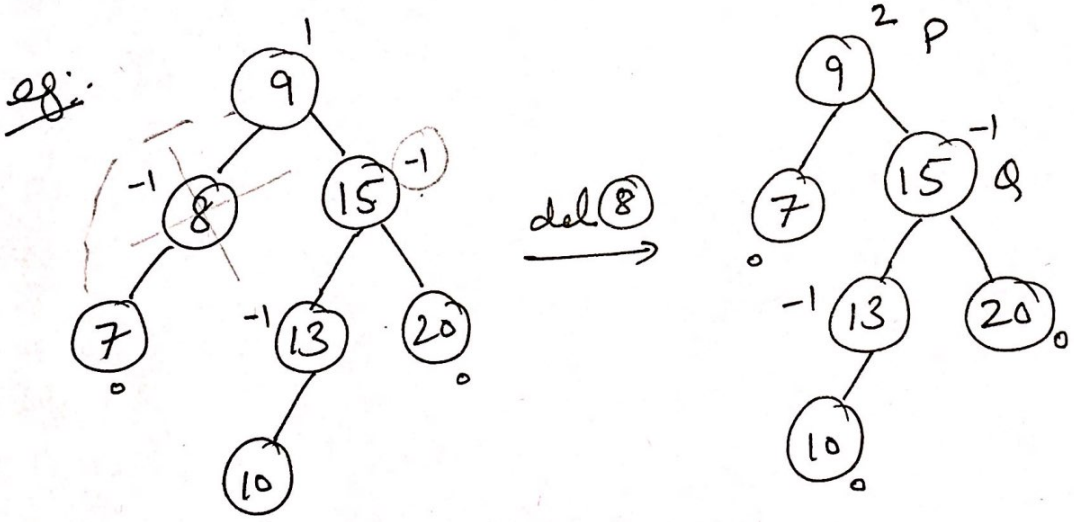
Deletion of a node having one child.

eg:



Rule :- R(-1) :- Right Rotate at P.

eg:



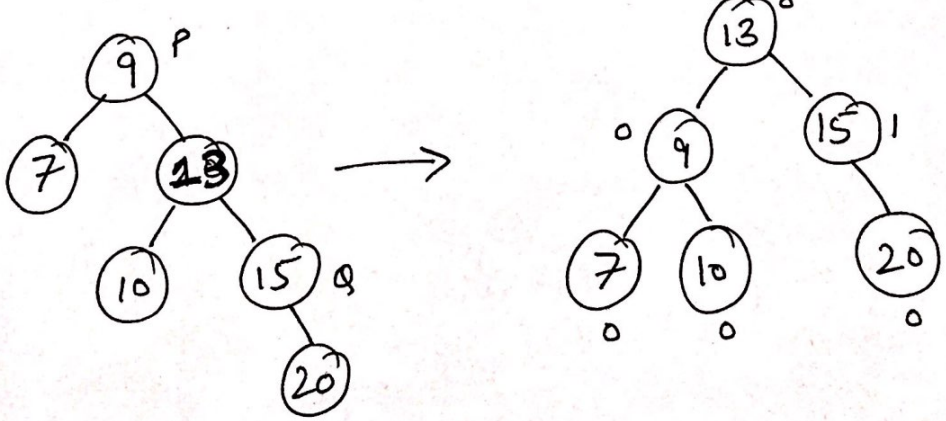
Left child : L

sibling's b.f := -1

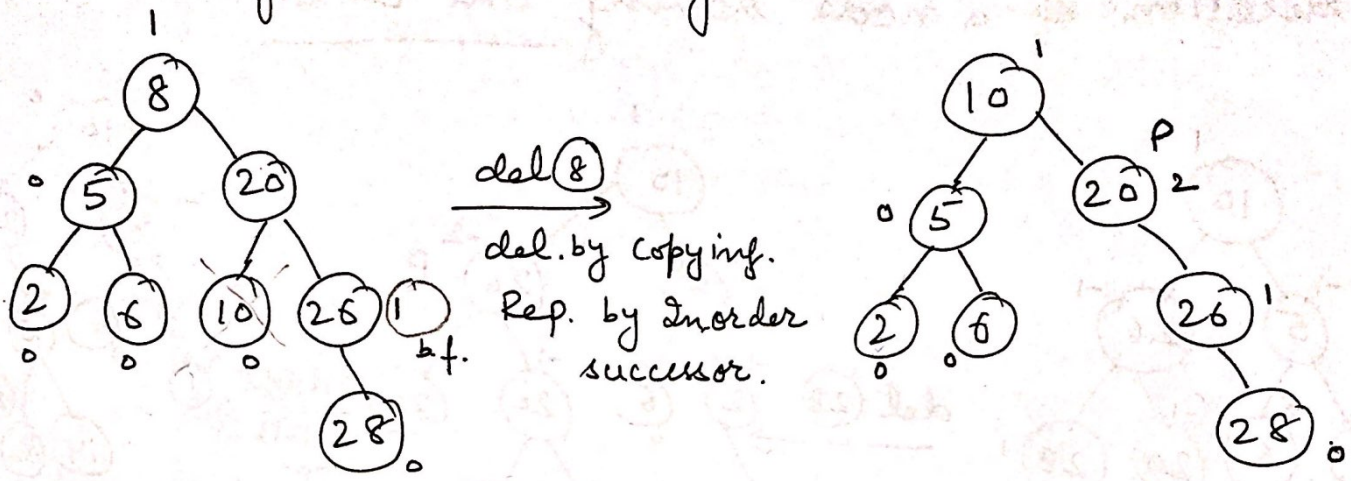
Rule L(-1)

(i) Right Rotate at Q

(ii) Left Rotate at P



Deletion of a node having two child :-



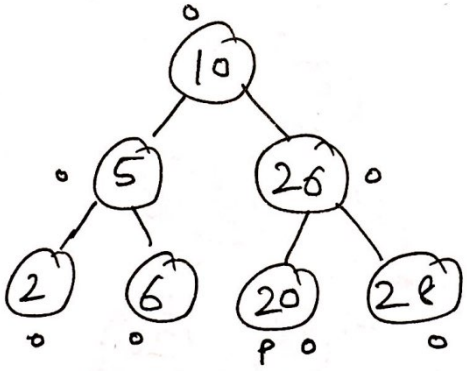
Actual node deleted :- (10).

Rule Identification :-

Left node deleted :- L

Sibling's b.f :- 1.

Rule :- L (+1) :- Left Rotate at P



A. Drozdek

ch-7 Multiway Trees (Pg- 301- 318)

7.1. The family of B Trees (w.e.f Jan'12)

(w.e.f. Jan'2012). (New course) (7.1.1, 7.1.2, 7.1.3).

ch-7. Multiway Trees

A multiway search tree of order 'm' or 'm-way search tree' is a multiway tree in which;-

1). each node has (atmost) 'm' children & (atmost) 'm-1' keys

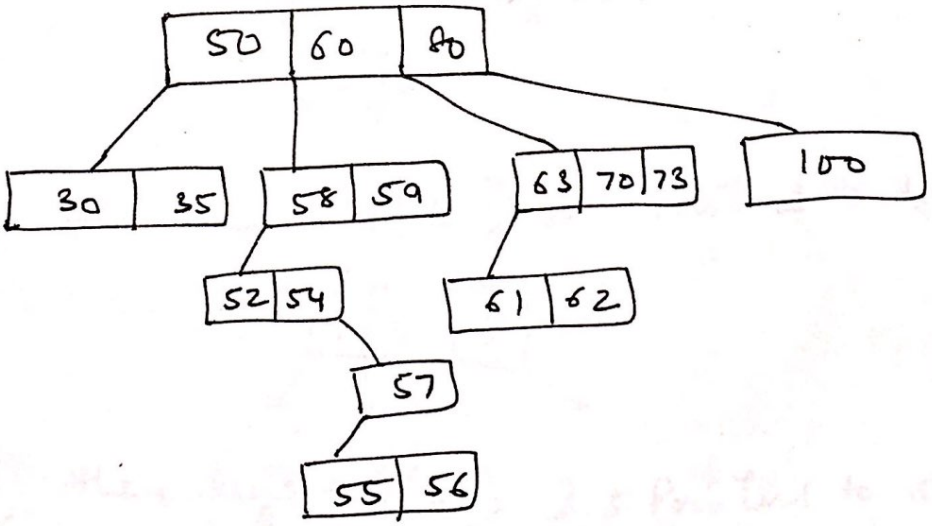
2). The keys in each node are in ascending order.

3). The keys in the first 'i' children are smaller than the 'i'th key.

4). The keys in the last 'm-i' children are larger than the i^{th} key.

eg:

4-way Tree (m = 4).



7.1.1. 'B- Trees'

A 'B-Tree' of order m is a multiway search tree with following properties :-

1). The root has atleast two subtrees unless it is a leaf.

2). Each non-root & non-leaf node holds 'k-1' keys & 'k' pointers to subtrees where $\lceil m/2 \rceil \leq k \leq m$.

eg :- B-Tree of order -5 (Here m=5).

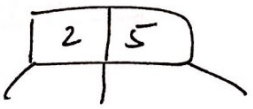
according to second condition each non-root & non-leaf node holds :-

$$\lceil \frac{5}{2} \rceil \leq k \leq 5$$

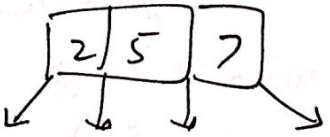
or

$$3 \leq k \leq 5.$$

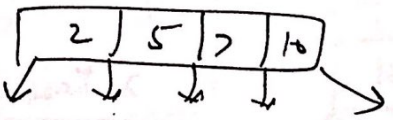
If k = 3, the keys = 2(k-1) & 3 pointers to subtrees.

ie 

If k = 4, the keys = 3(k-1) & 4 pointers to subtrees.



If k = 5, then keys = 4(k-1) & 5 Pointers to subtrees.



ie a node can contain maximum of 4 keys & 5 children.
& a node can contain minimum of 2 keys & 3 children.
(in case of B-Tree of order - 5).

3) Each leaf node holds $(k-1)$ keys where $\lceil m/2 \rceil \leq K \leq m$.
(same as second cond except that since we are talking about leaf nodes, it does not contain pointers or children).

4) All leaf are at the same level.

According to these conditions, a B-Tree is always at least half full, has few levels, & is perfectly balanced.

Inserting a Key into a B-Tree

While inserting bottom up manner is followed.

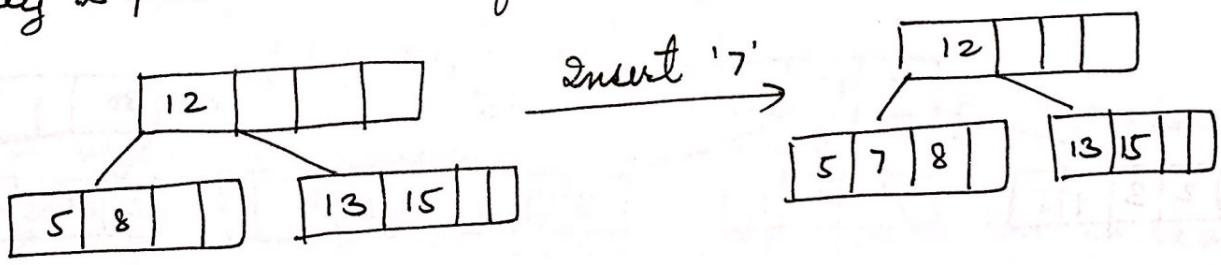
When a new key is to be inserted, leaves are checked first & place there if free space is available (case 1)

When a leaf is full, another leaf is created, the keys are divided between these leaves, & one key is promoted to the parent. (case 2).

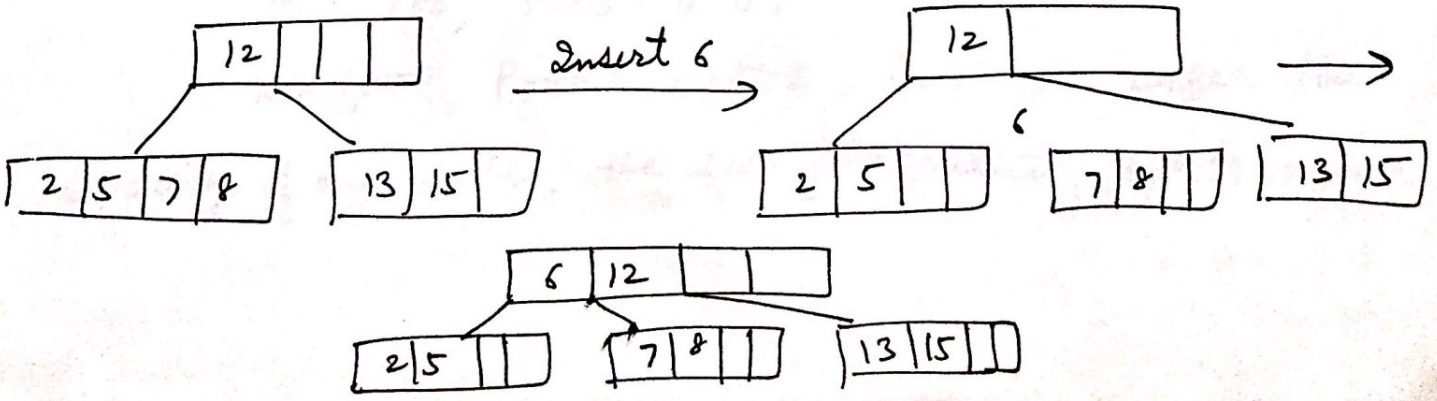
If the parent is full, the process is repeated until the root is reached & a new root created. (case 3)

Three common situations :-

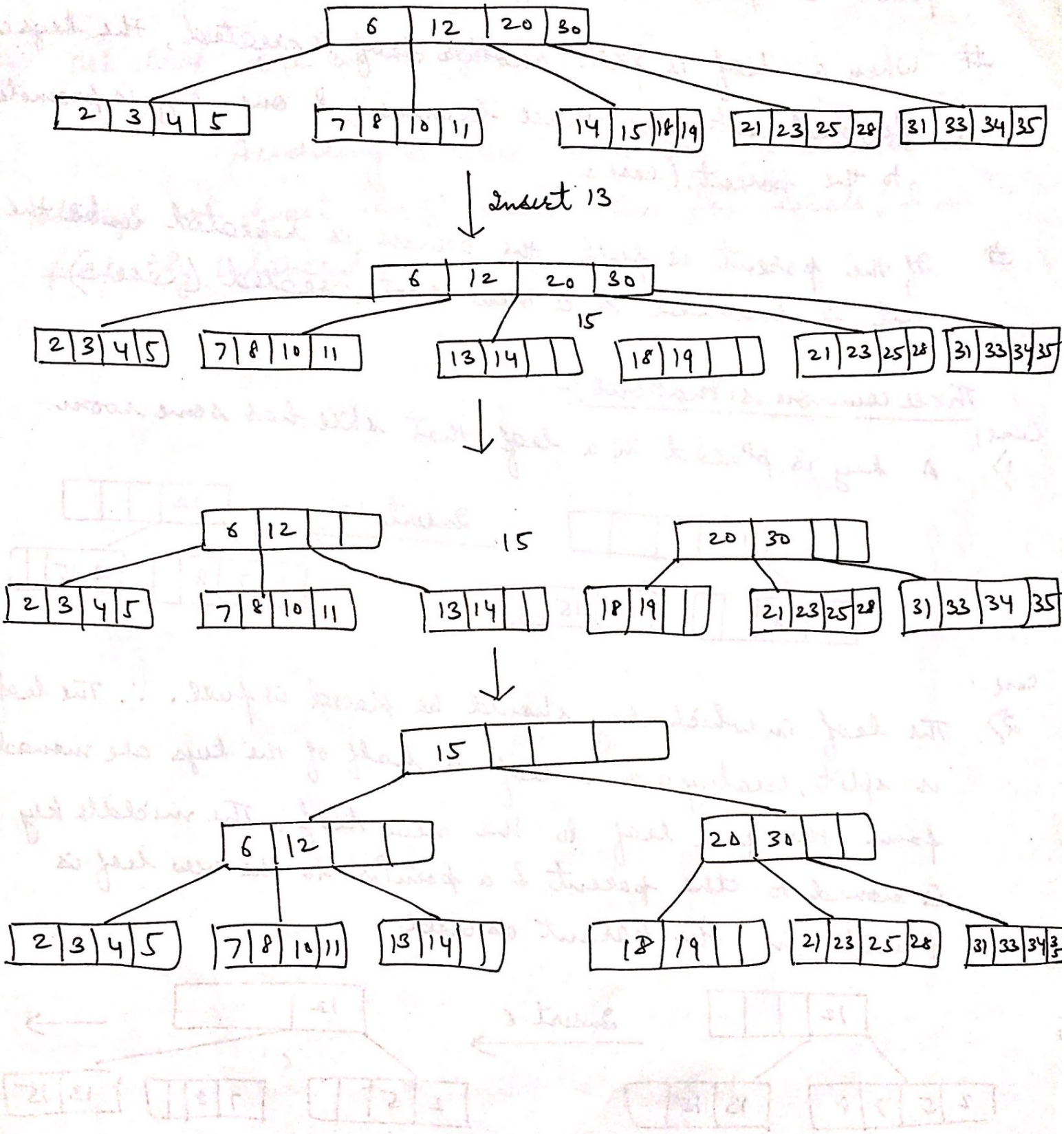
Case 1
1) A key is placed in a leaf that still has some room.



Case 2
2). The leaf in which key should be placed is full. ∴ The leaf is split, creating a new leaf & half of the keys are moved from the full leaf to the new leaf. The middle key is moved to the parent & a pointer to the new leaf is placed in the parent as well.



Case 3. This case arises when root of B-Tree is full. In this case, a new root & a new sibling of the existing root have to be created. This split results in two new nodes in the B-Tree.



A split of the root node of a B-Tree creates two new nodes. All other splits add only one more node to the B-Tree.

9-3=6

During the construction of B-Tree of 'P' nodes, 'P-h' splits have to be performed, where 'h' is the height of the B-Tree. Also in a B-Tree of 'P' nodes, there are at least

root key. max. root node =

$$1 + (\lceil m/2 \rceil - 1)(P-1) \text{ keys.}$$

$1 + (2)(8) = 17$

P = 9
h = 3.

see tree on next Page.

The rate of splits w.r.t. the no. of keys in the B-Tree is given by,

$$\frac{P-h/P-h}{\frac{1}{P-h} + (\lceil m/2 \rceil - 1)(P-1)}$$

$\frac{9-3}{17} = \frac{6}{17}$

P-h P-h

After dividing the numerator & denominator by 'P-h' & observing that $\frac{1}{P-h} \to 0$ & $\frac{P-1}{P-h} \to 1$ with the increase of P, the average probability of split is,

$$\frac{1}{\lceil m/2 \rceil - 1}$$

For eg:- If $m = 10$, Prob = 0.25,

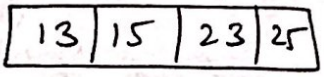
$m = 100$, Prob = 0.02

$m = 1,000$ Prob = 0.002. Thus, larger the capacity of one node, the less frequently splits occur.

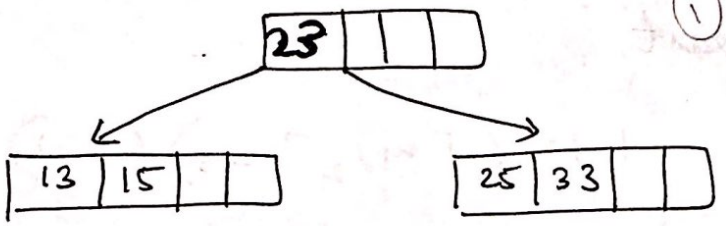
eg. Insert foll. keys in B-Tree of order 5.

13, 25, 23, 15, 33, 19, 27, 37, 31, 21, 45, 29, 43, 51, 17, 35, 39, 47, 49, 41

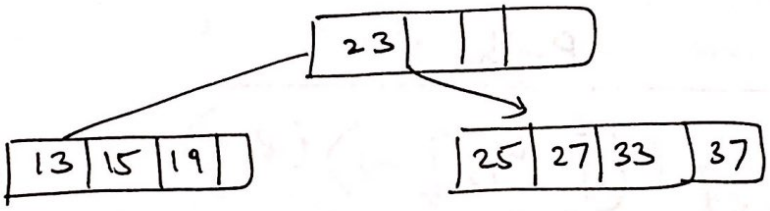
(a) Insert 13, 25, 23, 15



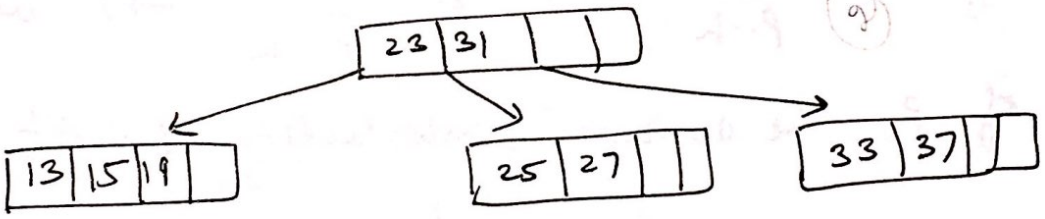
(b) Insert 33



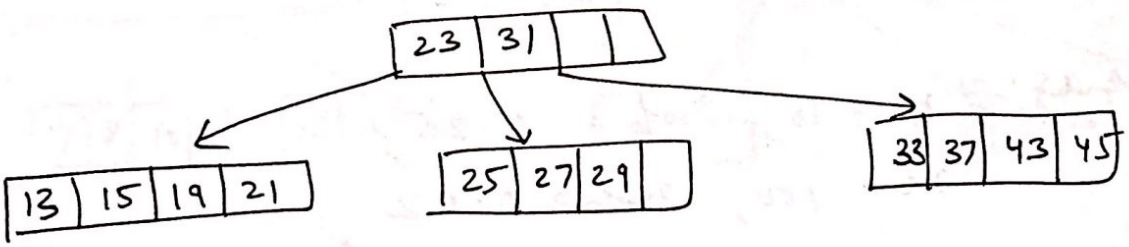
(c) Insert 19, 27, 37



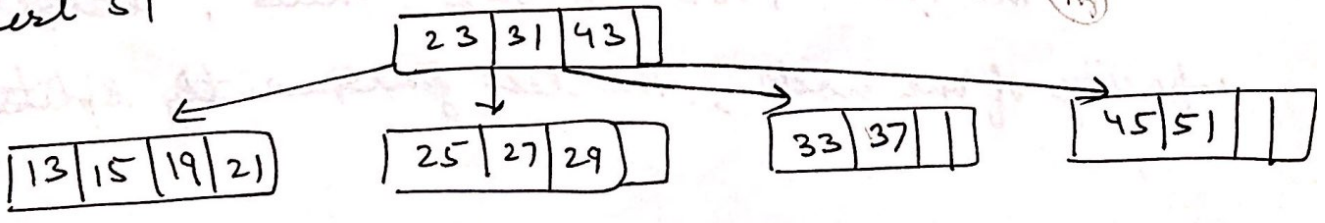
(d) Insert 31.



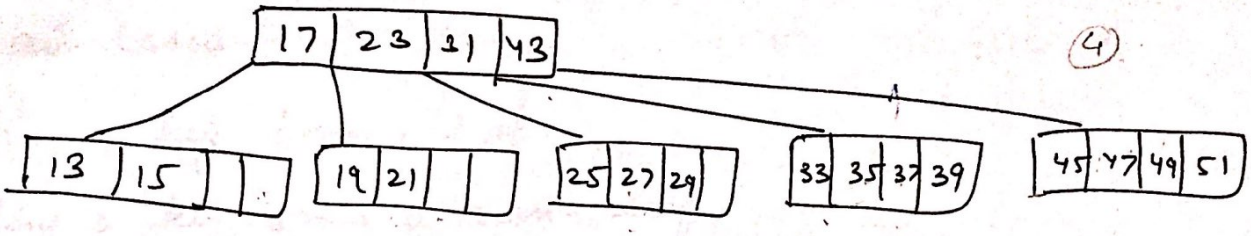
(e) Insert 21, 45, 29, 43



(f) Insert 51



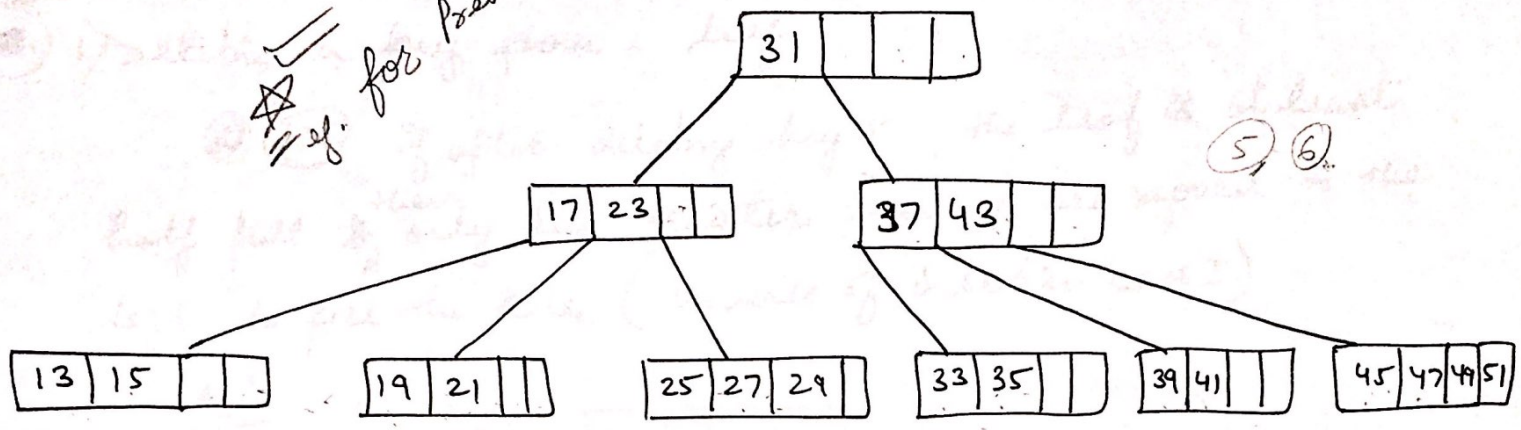
(3) Insert 17, 35, 39, 47, 49



(4)

(4) Insert 41.

Eg. for previous



(5) (6)

Deleting a key from a B-Tree

Two main cases :-

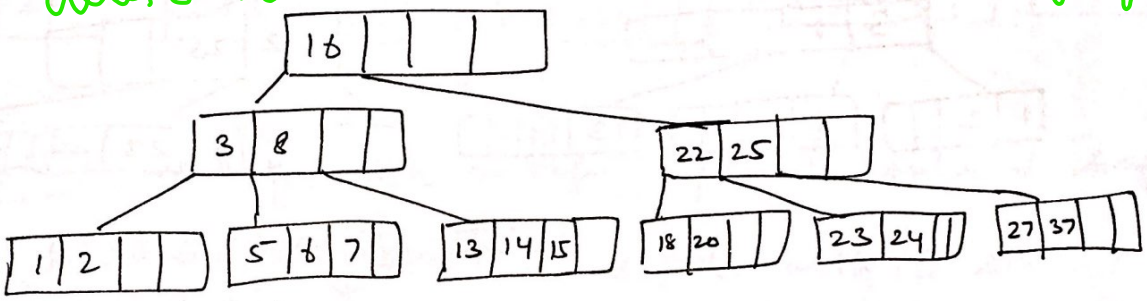
1) Deleting a key from a leaf.
2) Deleting a key from a non-leaf.

Case.

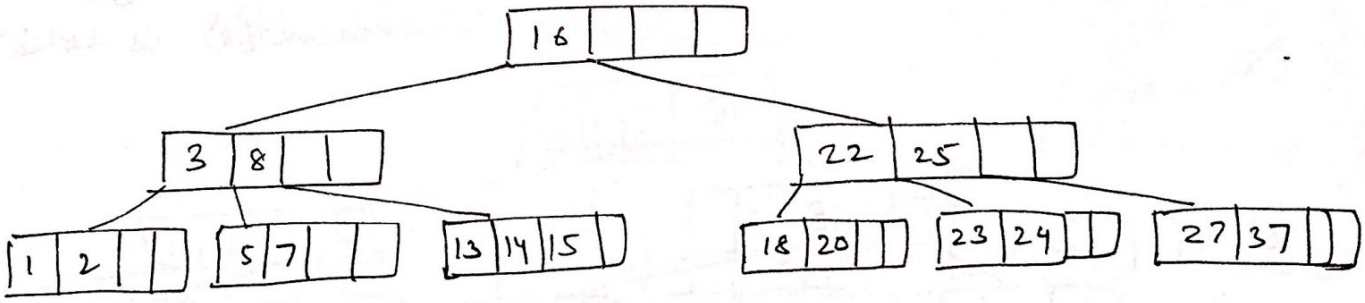
1) Deleting a key from a leaf.

(1.1) If after deleting key K, the leaf is at least half full then only keys greater than K are moved to the left to fill the hole. (inverse of insertion case I).

eg :- delete karne ke baad at least half full



Delete 6 ↓

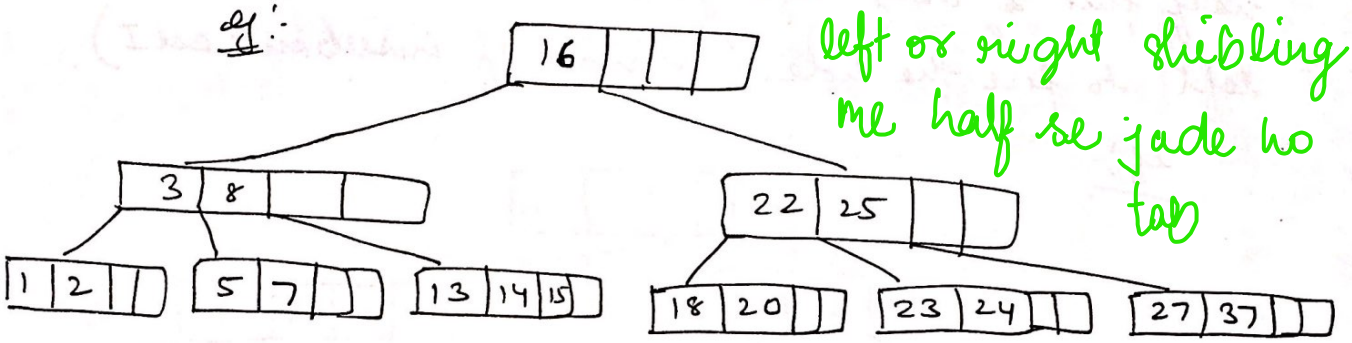


(1.2) If after deleting 'K', the no. of keys in the leaf is less than $\lceil m/2 \rceil - 1$ (or less than 2 if $m = 5$), it causes an underflow.

after deletion, if no. of keys is less than half, then it underflow.

(1·2·1) If underflow is caused & there is a left or right sibling with the no. of keys exceeding the minimal $\lceil m/2 \rceil - 1$ (or 2 when $m=5$), then all keys from this leaf & sibling are redistributed b/w them by moving the separator key from the parent to the leaf & moving the middle key from the node & the sibling combined to the parent.

Eg:-

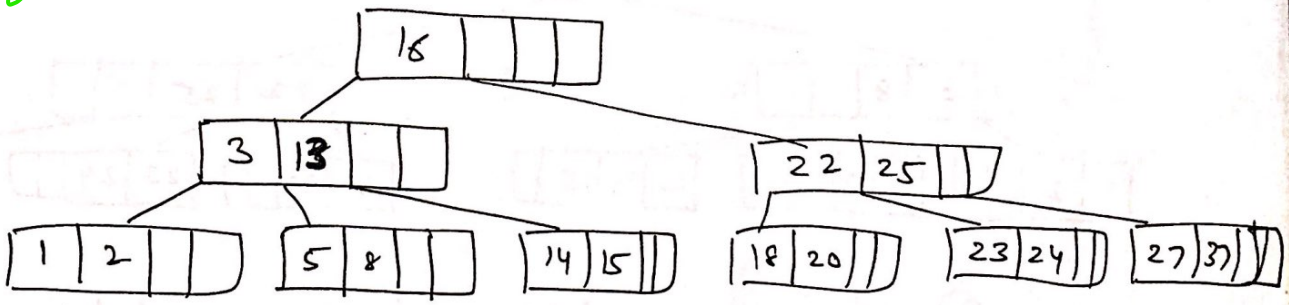


left or right sibling me half se jade ho tab

Delete (7) → (7) is deleted, right sibling is having more than 2 keys, ∴ leaf (5), sibling (13,14,15) & parent (8) is redistributed.

5 → parent

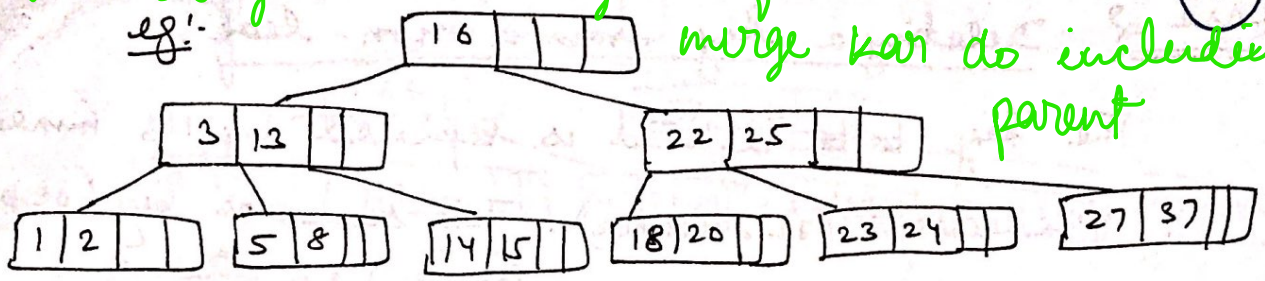
4 → merge



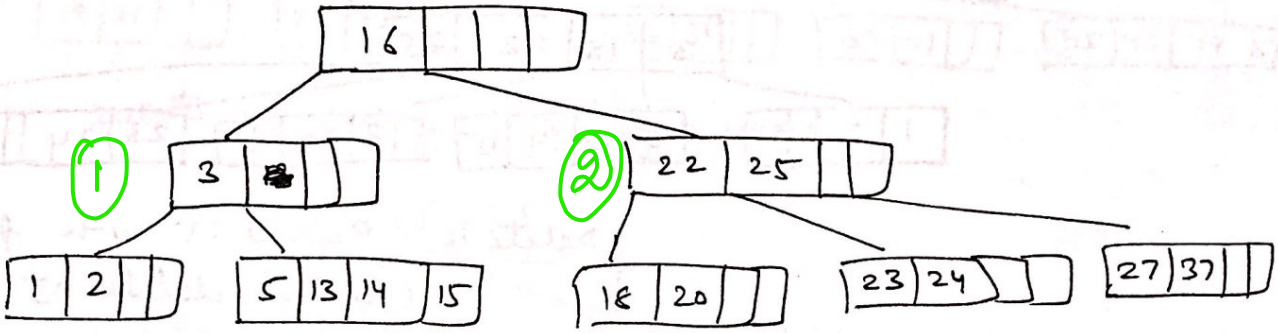
(1·2·2). If underflow is caused & there is a left or right sibling with exactly $\lceil m/2 \rceil - 1$ keys. (2 keys if $m=5$), then leaf & sibling are merged. The keys from the leaf, from its sibling & separating key from the parent are all put in the leaf & sibling node is discarded.

Left or right sibling me exactly half ho, to merge kar do including parent

eg:-



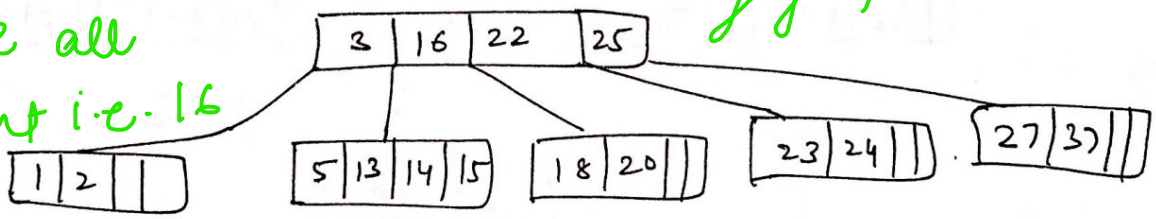
delete (8). The sibling (14,15) of leaf (5,8) is having exactly 2 keys. ∴ (5, 13, 14, 15) are merged & sibling is discarded.



(1·2·2·1) Now if after merging parent contains single key, then again it is merged with its sibling. (case 1·2·1 or 1·2·2). for (1) & (2) again upar wala case lag gaya,

eg:- delete & contd....

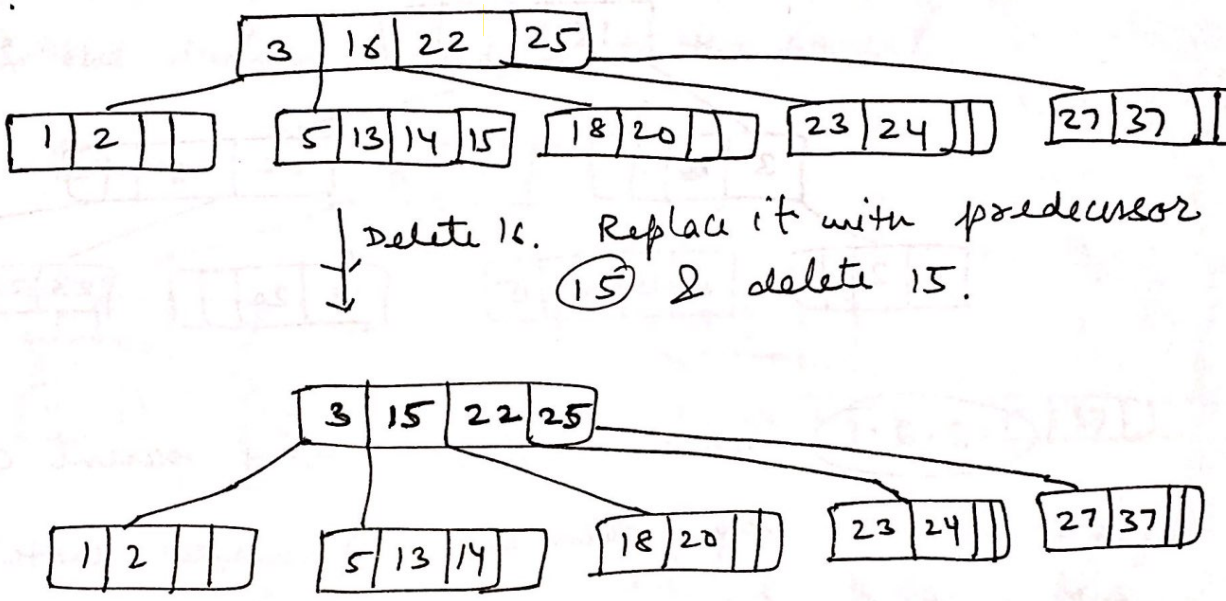
We will merge all including parent i.e. 16



Case 2: Deleting a key from a non-leaf.

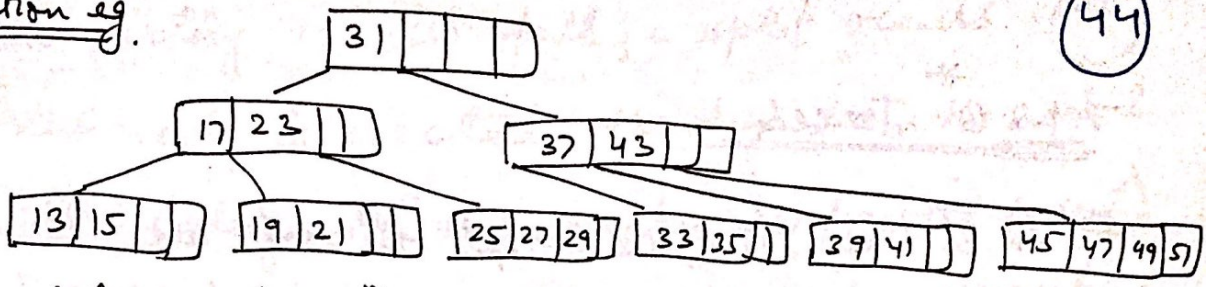
The key to be deleted is replaced by its immediate predecessor (or successor). This successor key (or predecessor) is deleted from the leafs, which again brings us to case (1).

eg:-

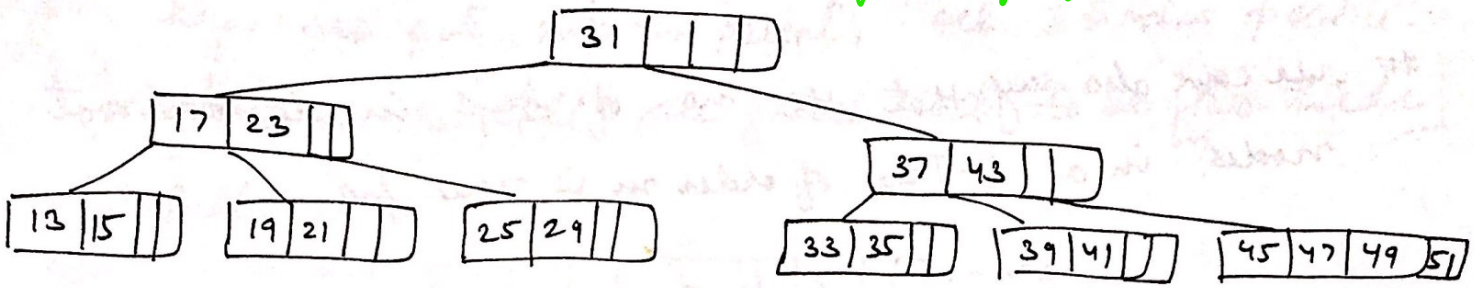


Since delete karne ke baad half and half se jade keys hai, thus no change required.

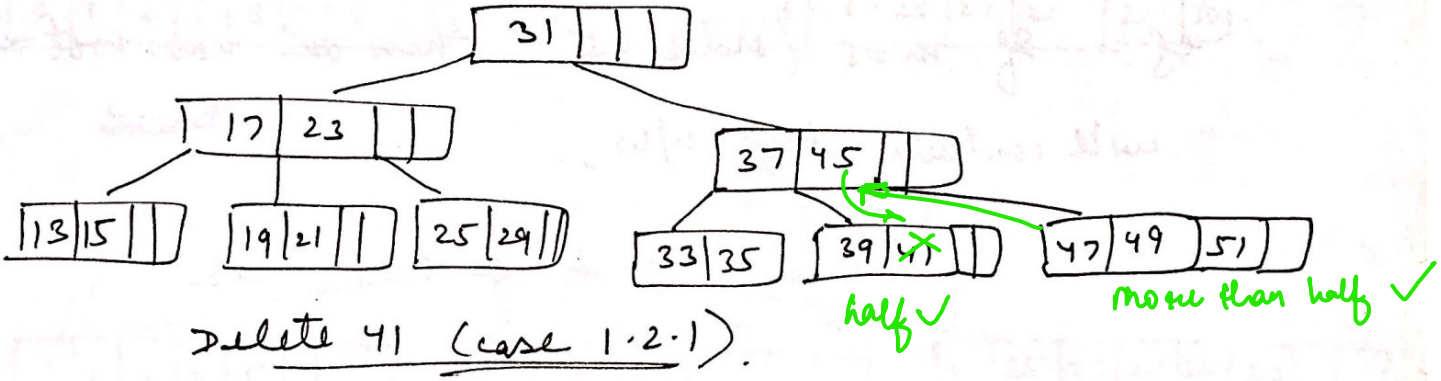
Deletion eg.



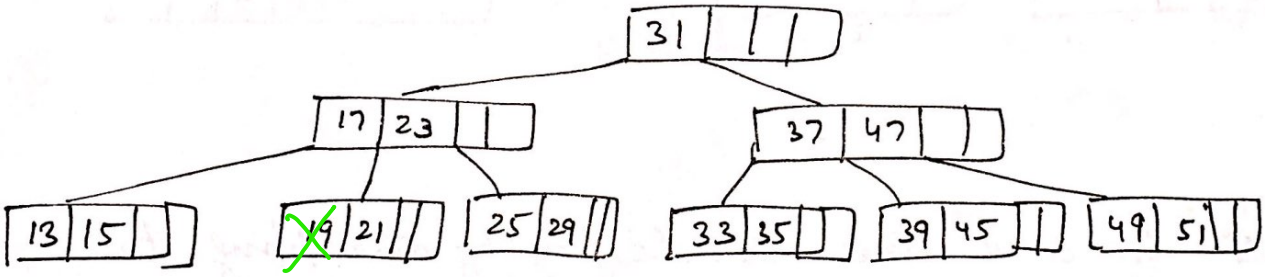
Delete 27 (case 1) → again half keys & C to R subtrees



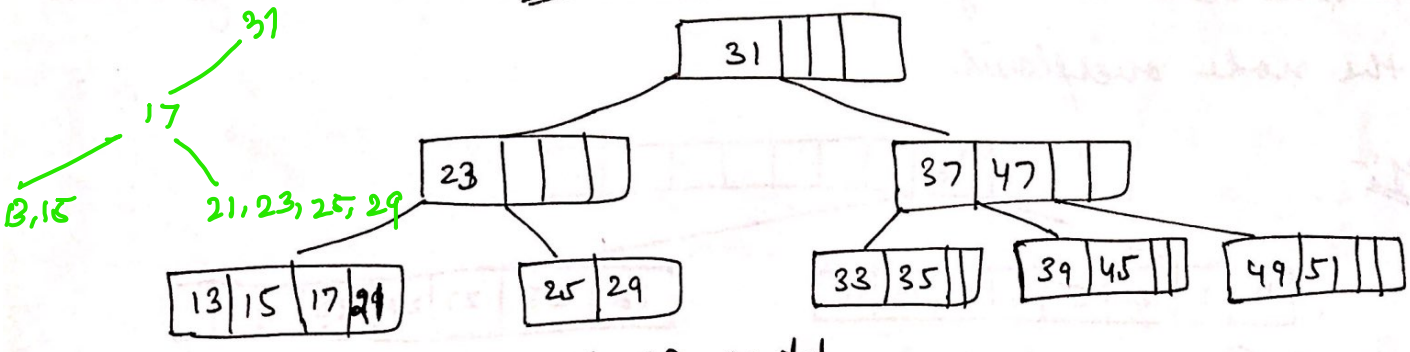
Delete 43 (case 2). (Replace by successor).



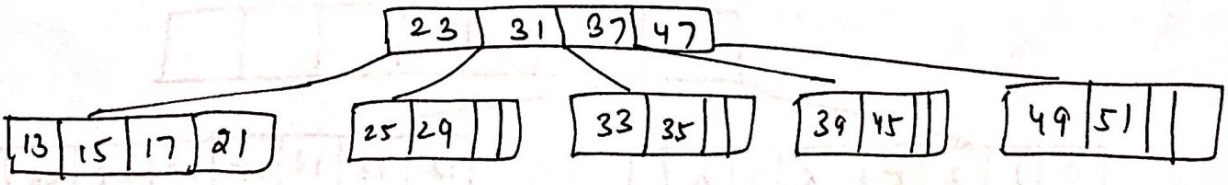
Delete 41 (case 1.2.1).



Delete 19.



Delete 19 contd...



Not in syll

B → trees at least half full

B* at last 2-third full

7.1.2 B* Trees

In B* tree, all nodes except the root are required to be at least two-thirds full & not just half full as in B-Tree.

we can also say, that the no. of keys in all non-root nodes in a B-Tree of order m is now for

$$\left\lfloor \frac{2m-1}{3} \right\rfloor \leq k \leq m-1.$$

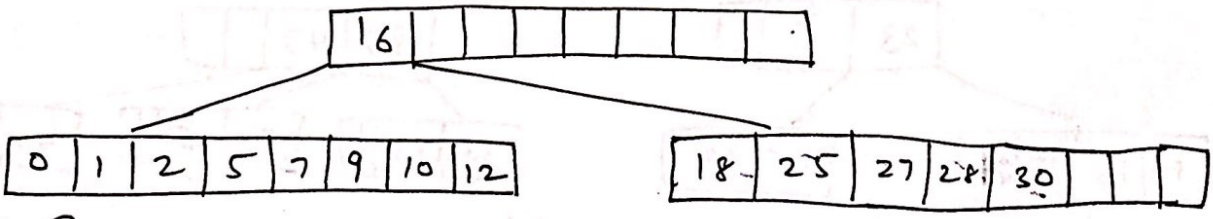
eg:- If $m=5$ (order = 5), then all non-root nodes will contain keys b/w,

$$\left\lfloor \frac{2.5-1}{3} \right\rfloor \leq k \leq 5-1 \quad \text{or}$$

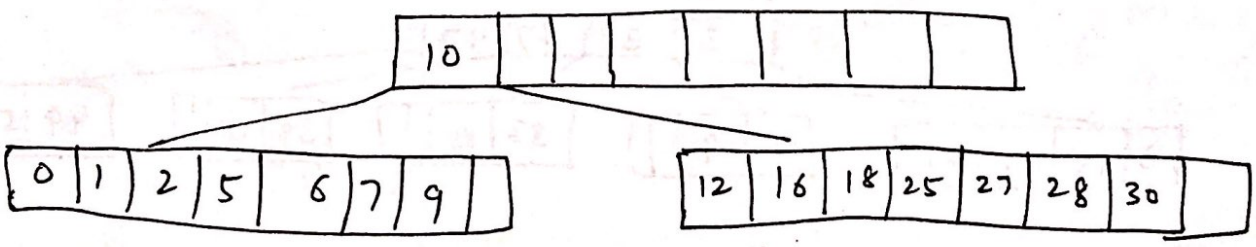
$$3 \leq k \leq 4.$$

A split in a B* Tree is delayed by attempting to redistribute the keys b/w a node and its sibling when the node overflows.

eg:-

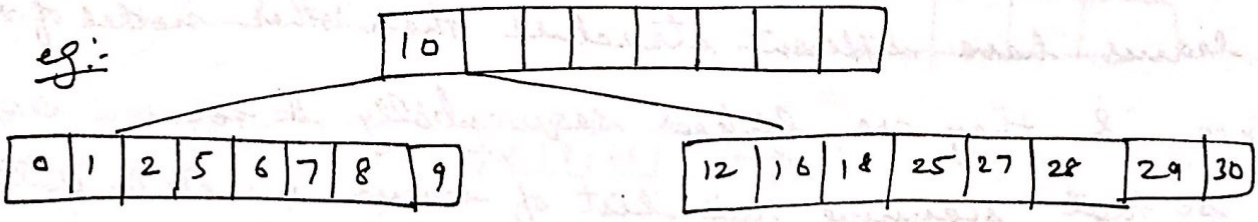


Insert (6).

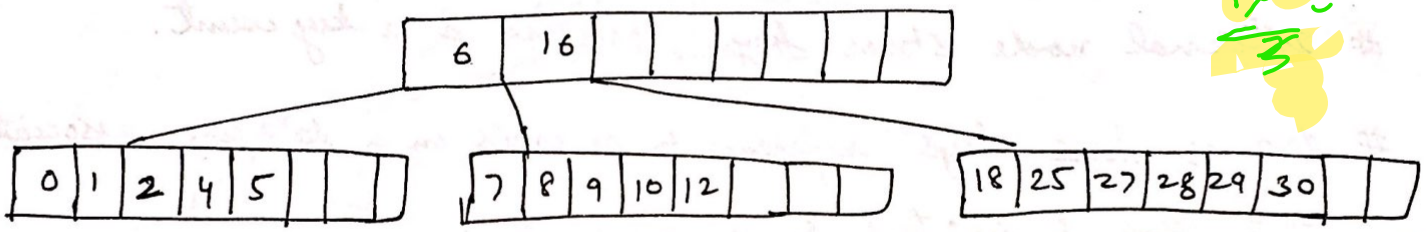


If the sibling is also full, a split occurs. One new node is created, the keys from the node & its sibling (along with the separating key from the parent) are evenly divided among three nodes & two separating keys are put into the parent. All 3 nodes participating in the split are guaranteed to be [two-thirds] full.

eg:-



Insert 4



* NOT in syll

7.1.3 B^+ Trees

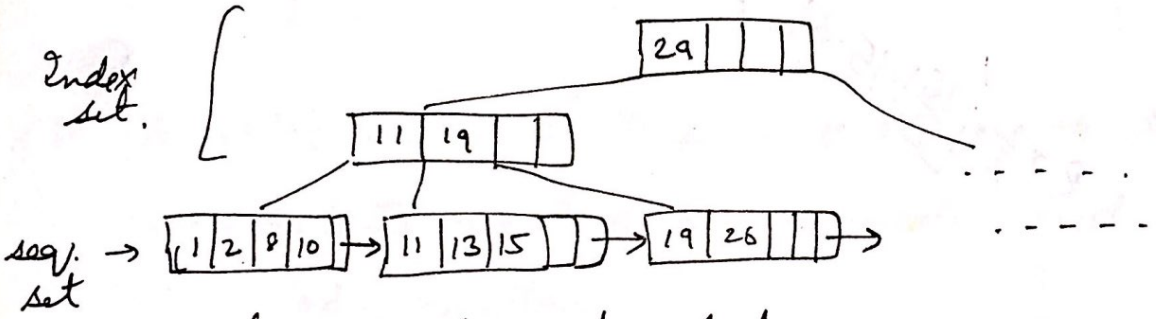
In B^+ Trees, Refrence to key value is made only from the leaves.

The internal nodes of a B^+ Tree are indexes for fast access of data. (○ → internal node) This part of tree is called index set.

The leaves have different structure than other nodes of the B^+ Tree & they are linked sequentially to form a sequence set so that scanning this list of leaves results in data given in ascending order.

internal node stores keys, pointers & a key count.

Leaves stores keys, reference to records in a data file associated with keys & pointers to the next leaf.



Insertion of a key into a leaf

1) If leaf has some empty space then simply reorders the key values along with new key. No changes are made in index set.

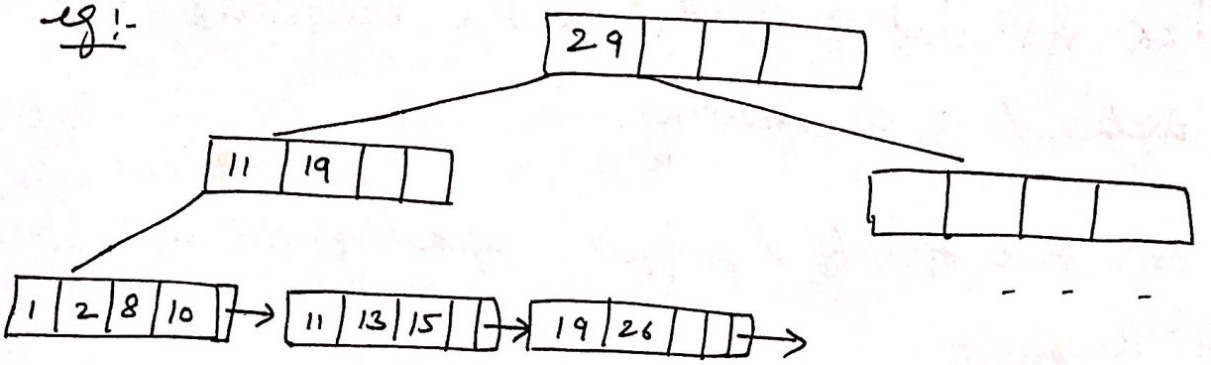
2) If key is inserted in full leaf :- the leaf is split. The new leaf node is included in the sequence set, all keys are distributed evenly b/w old & new leaves & first key from the

new node is copied (not moved as in B-Tree) to the paren

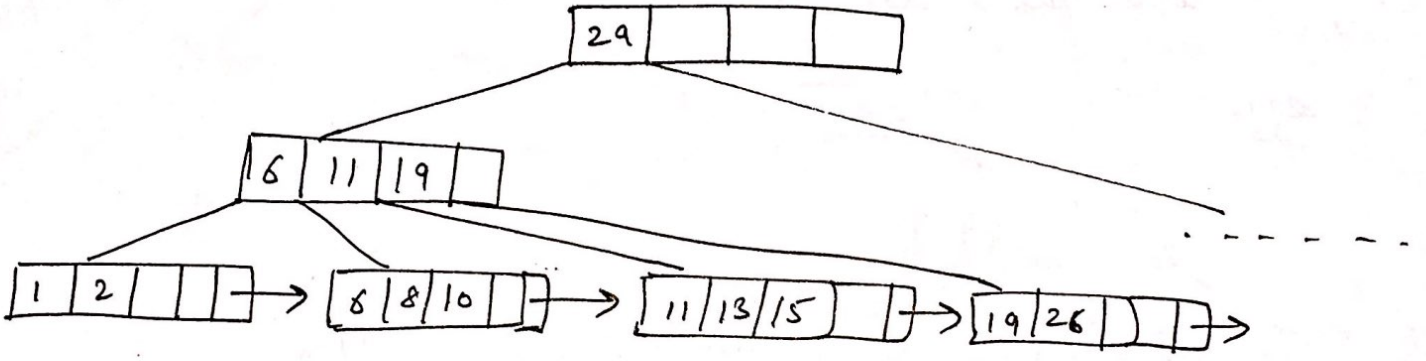
ⓐ If the Parent is not full, reorder the keys of parent.

ⓑ If the Parent is full, splitting process is performed in the same way as in B-Trees.

eg:-



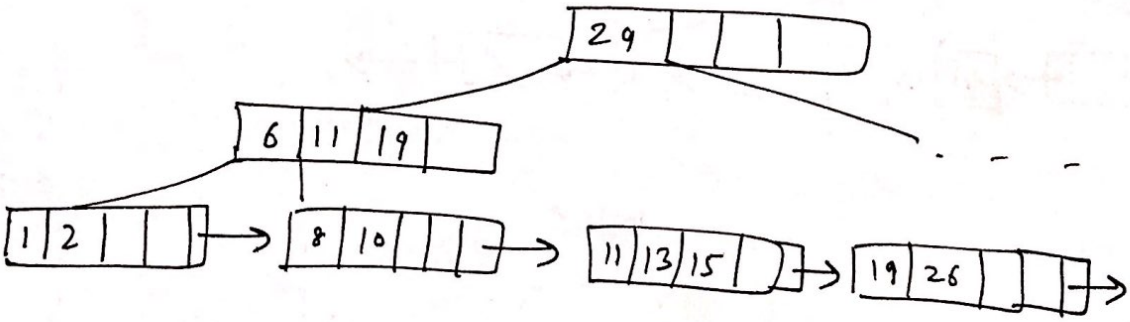
Insert 6.



Deleting a key from a leaf

① If no underflow occurs then put the remaining keys in order No changes are made to index set. (i.e key is deleted only from the sequential set & not from index set.)

eg:- Delete 6 from above B^+ Tree.



(2) When deletion of a key from a leaf causes an underflow then use any one of the steps :-

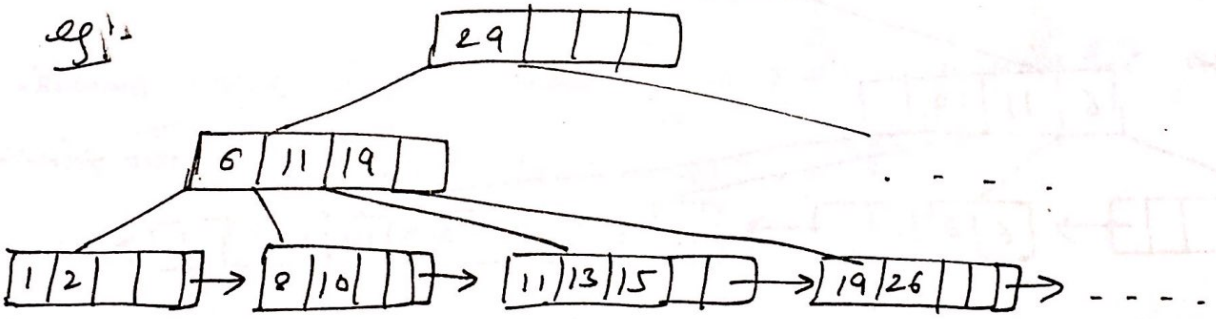
(a) keys from the leaf & keys of a sibling are redistributed b/w this leaf & its sibling.

(b) OR the leaf is deleted & the remaining keys are included in its sibling.

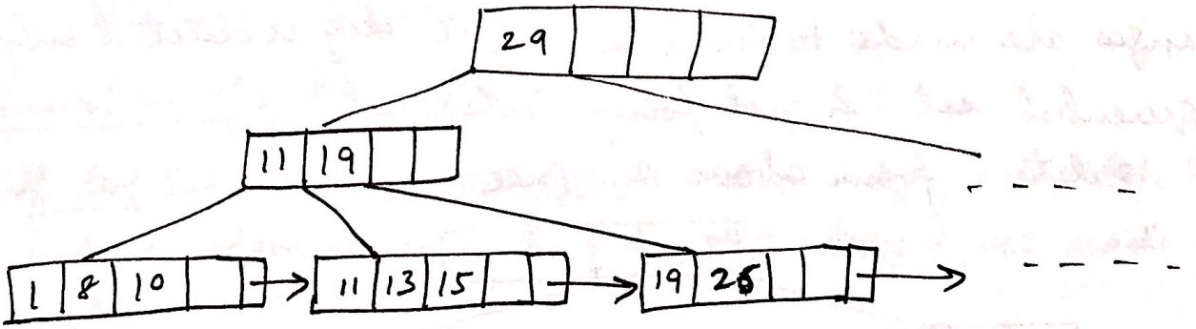
Both these operations require updating the separator in the parent.

Also, removing a leaf may initiate merge operation in the index set.

eg :-



Delete (2). [use 2(b) Case].



Heapsort

A (binary) heap data structure is an array object that can be viewed as a nearly complete binary tree.

Two kinds of binary heaps:

1) **max heaps** :- max-heap property

is that for every node i other than the root,

$$A[\text{Parent}(i)] \geq A[i]$$

That is, the largest element in a max-heap is stored at the root and the subtree rooted at node contain values no larger than that contained at node itself.

2) **min-heaps** : min-heap property

is that for every node i other than the root,

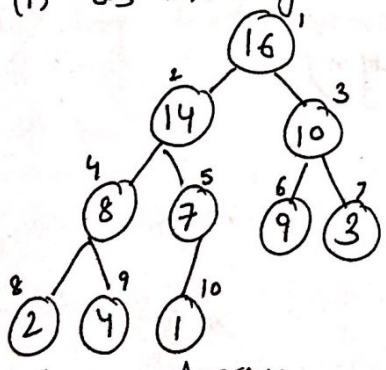
$$A[\text{Parent}(i)] \leq A[i]$$

The smallest element in a min-heap is at the root.

Structure of

(1) max-heap

(i) as binary tree

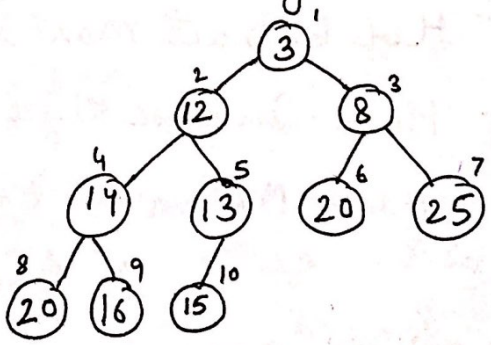


(ii) as an Array

1	2	3	4	5	6	7	8	9	10
16	14	10	8	7	9	3	2	4	1

(2) min-heap

(i) as binary tree



(ii) as as Array

1	2	3	4	5	6	7	8	9	10
3	12	8	14	13	20	25	20	16	15

To find Parent, left and right nodes of node 'i'.

1) Parent (i)
   return $\lfloor i/2 \rfloor$

2) left (i)
   return 2i

3) right (i)
   return 2i + 1

Height of Heap

The height of Complete binary tree is $\theta(\log n)$ where 'n' is the no. of elements. Since the heap of 'n' elements is based on complete binary tree, its height is $O(\log n)$.

The basic operations on heaps run in time atmost proportional to the height of the tree and thus take $O(\log n)$ time.

height of heap → $O(\log n)$
Max-heapify procedure → $O(\log n)$
Build max-heap → $O(n)$ [linear time]
Heap sort procedure → $O(n \log n)$
Priority Queues → $O(\log n)$
   - Max-heap Insert()
   - Heap Extract max()
   - Heap-Increase Key()
   - Heap-Maximum()

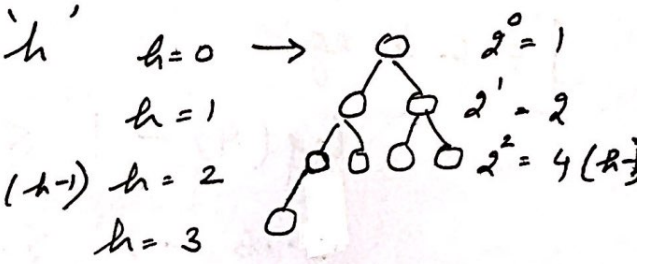
   → $O(\log n)$

Height of a Heap

Th. A heap T storing n entries (nodes) has height

$$h = \lfloor \log n \rfloor$$

Proof From the fact that T is complete (Almost complete), we know there 2^i nodes at each level. $(0 \le i \le h-1)$ and atleast 'one node' at level 'h'



Thus, T has atleast

$$(1 + 2 + 4 + \ldots + 2^{h-1}) + 1$$

$$= (2^{h-1} - 1) + 1 = 2^h \text{ nodes.}$$

Level h has atmost 2^h nodes. ($2^3 = 8$ eg.).

Thus T has atmost

$$(1 + 2 + 4 + \ldots 2^{(h-1)}) + 2^h = 2^{h+1} - 1 \text{ nodes.}$$

Thus given no. of nodes (n) in heap lies between:-

$$2^h \le n \le 2^{h+1} - 1 \qquad [\, 8 \le 8 \le 15 \ \text{(eg.)} \,]$$

Taking log both sides

$$2^h \le n$$

$$h \log_2 2 \le \log_2 n$$

$$h \le \log n$$

$$n \le 2^{h+1} - 1$$

$$(n+1) \le 2^{h+1}$$

$$\log (n+1) \le 2^h \cdot 2 \qquad (\log a + \log b)$$

$$\log (n+1) \le \log 2^h + \log_2 2$$

$$\log (n+1) \le h + 1$$

$$\log (n+1) - 1 \le h$$

$$\log(n+1) - 1 \leq h \leq \log n$$

Since h is an integer (positive height), the two inequalities imply

$$h = \lfloor \log n \rfloor.$$

[acc. to eg:-

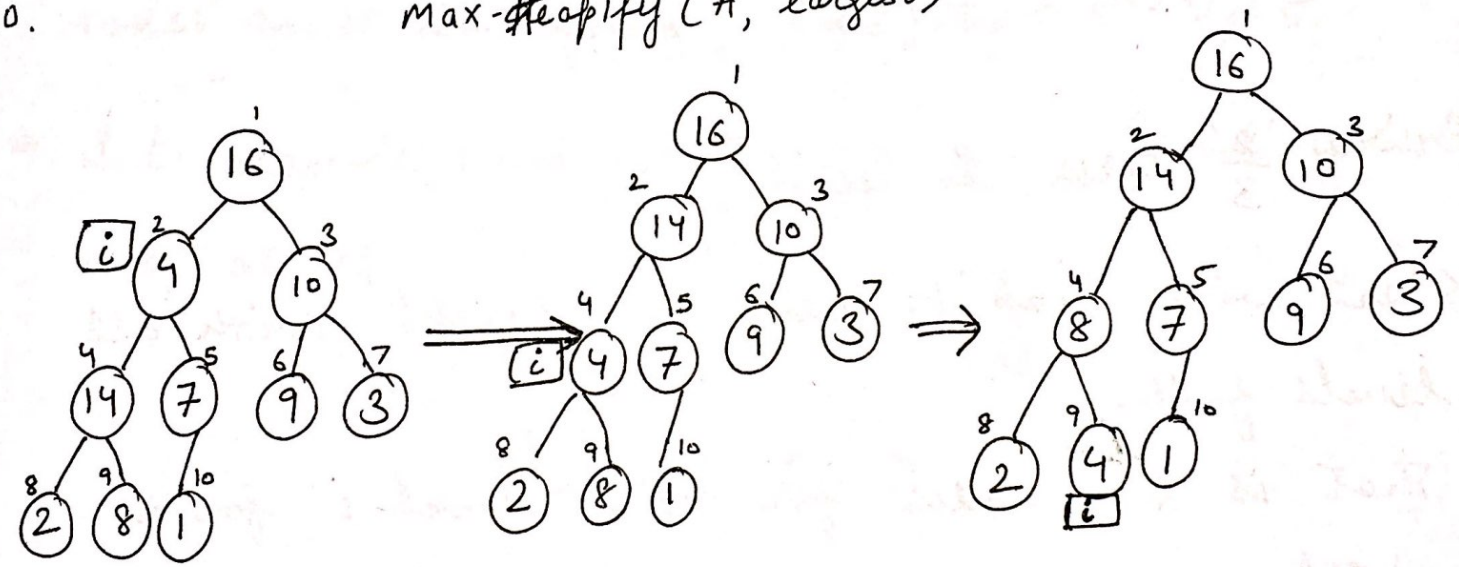
$$\log(9) - 1 \leq 3 \leq \log_2 8$$

$$3.16 - 1 \leq 3 \leq \log_2 8$$

$$2.16 \leq 3 \leq 3. \quad]$$

Max-Heapify (A, i)

1. $l \leftarrow$ left (i)
2. $r \leftarrow$ right (i)
3. If $l \leq$ heapsize [A] && A[l] > A[i]
4. then largest $\leftarrow l$
5. else largest $\leftarrow i$
6. If $r \leq$ heapsize [A] && A[r] > A [largest]
7. then largest $\leftarrow r$
8. If largest $\neq i$
9. then exchange A[i] $\leftrightarrow$ A [largest]
10. Max-Heapify (A, largest)



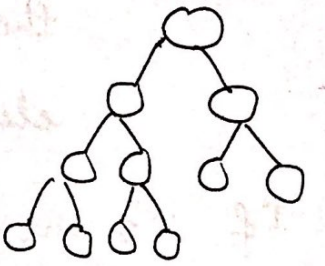
eg:- Max-Heapify (A, 2)

heapsize [A] = 10

Running time of Max-heapify [given - size = n].

constant time to fix relationships

- A[i]
- A[Left(i)]
- A[Right(i)]



Plus

Time to run Max-Heapify.

- children subtree each have size atmost '$\frac{2n}{3}$'.
- Max-Heapify can therefore be described by the recurrence

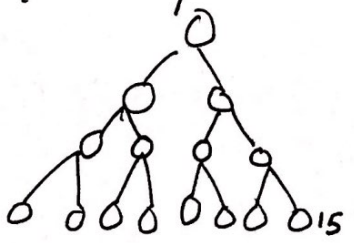
$$T(n) \leq T\left(\frac{2n}{3}\right) + \Theta(1)$$

why '$\frac{2}{3}$' term is used?

Start with heap 'H' with 'h height' with all levels full.

That is 2^{i-1} nodes for each level i for a total no. of nodes in heap given as

$$|H| = 2^h - 1.$$



$h = 4$.

level i $(1 \leq i \leq h)$.

level	no. of nodes (2^{i-1})
1	$2^0 = 1$
2	$2^{2-1} = 2$
3	$2^{3-1} = 4$
⋮	⋮

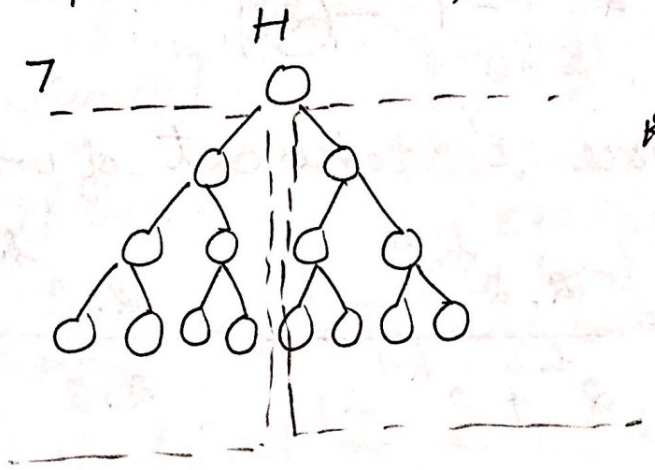
Total

$|H| = 2^4 - 1 = 15$.

Let L and R denote left & Right sub-heaps with total no. of nodes :-

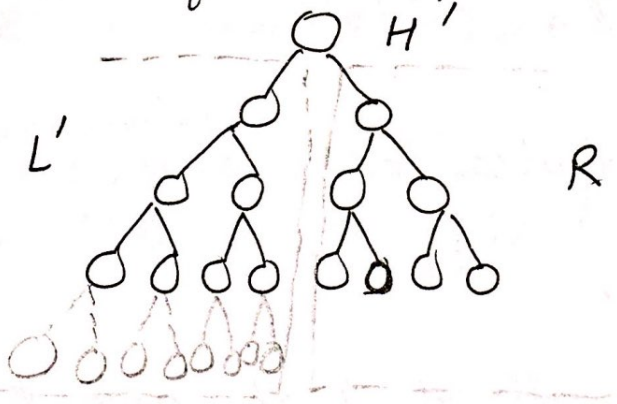
$|L| = 2^{h-1} - 1$, $|R| = 2^{h-1} - 1$

$= 2^{4-1} - 1 = 7$ $= 2^{4-1} - 1 = 7$



Since binary heap is complete binary tree, so new nodes must be added from left to Right.

Lets denote modified heap as L' and whole heap as H'.



This addition will require 2^{h-1} $(2^3 = 8)$ nodes in left heap. Thus total no. of nodes in L' is:

$$|L'| = (2^{h-1} - 1) + 2^{h-1}$$

$$= 2 \cdot 2^{h-1} - 1 \quad -(1)$$

$$|L'| = 2 \cdot 2^{4-1} - 1$$

$$= 15$$

Total no. of nodes in modified heap H' is:

$$|H'| = (2^h - 1) + 2^{h-1}$$

$$= 2^h + 2^{h-1} - 1 \quad -(2)$$

$$|H'| = 2^4 + 2^3 - 1$$

$$= 16 + 8 - 1 = 23$$

Amount of space L' takes out of whole H'

$$\frac{|L'|}{|H'|} = \frac{(1)}{(2)} = \frac{2.2^{h-1} - 1}{2^h + 2^{h-1} - 1} = \frac{2.2^{h-1} - 1}{2.2^{h-1} + 2^{h-1} - 1}$$

$$= \frac{2.2^{h-1} - 1}{(2+1)2^{h-1} - 1} = \frac{2.2^{h-1} - 1}{3.2^{h-1} - 1}$$

$$\lim_{h \to \infty} \frac{L'}{H'} = \lim_{h \to \infty} \frac{2(2^{h-1} - 1)}{3(2^{h-1} - 1)} = \frac{2}{3}$$

Here L' has twice as many elements as R.

$|L'| = 15$, $|R| = 7$.

So L' makes $\frac{2}{3}$ of heap and R has $\frac{1}{3}$ of heap elements.

Adding another node will begin to rebalance the heap by filling the right half.

Thus we can say the <u>worst case</u> occurs when the last row of the tree is <u>exactly half-full</u>.

Therefore the running time of max-heapify can be described by the recurrence

$$T(n) \leq T\left(\frac{2n}{3}\right) + \Theta(1)$$

[case 2 of master Th]. , $a = 1, b = \frac{3}{2}, f(n) = 1$

$n^{\log_b a} = n^{\log_{3/2} 1} = n^0 = 1 = f(n)$.

$\therefore \varepsilon = 0$ (case II)

$\therefore T(n) = O(\log_2 n)$

$\Theta(1 \cdot \log n) = \Theta(\log n)$.

we can also say, the running time of Max-Heapify on a node of height h as $O(h)$.

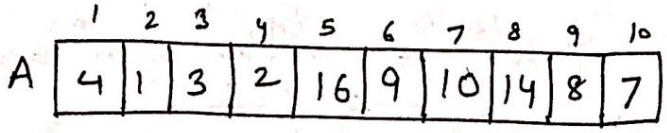
Summary.

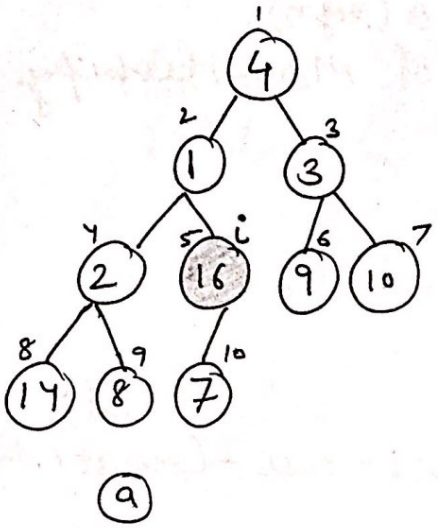
Max Heapify.

1) It traces path from root to leaf (worst case - Longest path) (h)
2) At level level it makes exactly 2 comparisons.
3) Total no. of comparisons $= 2h$.
4) Running time $O(h)$ or $O(\log_2 n)$ [since $h = \log_2 n$]

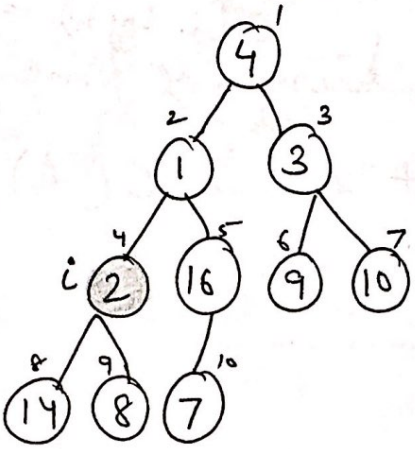
Build max Heap (A)

1. heap_size[A] ← length[A]
2. for i ← ⌊length[A]/2⌋ down to 1
3. do max_heapify (A, i)

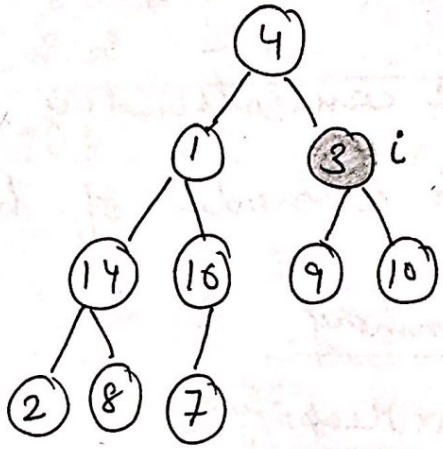




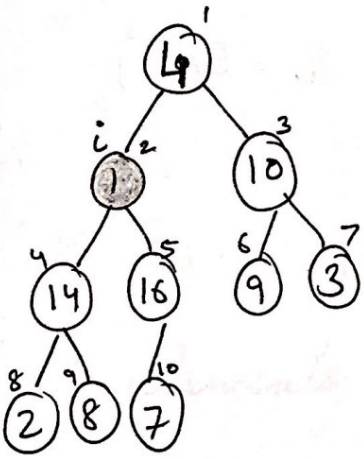
(a)



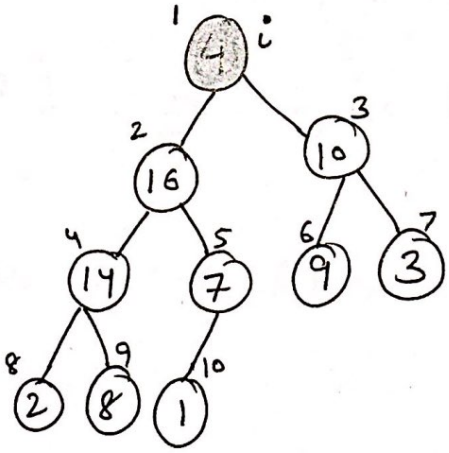
(b)



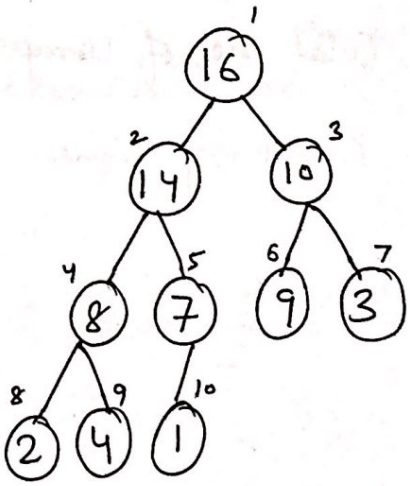
(c)



(d)



(e)



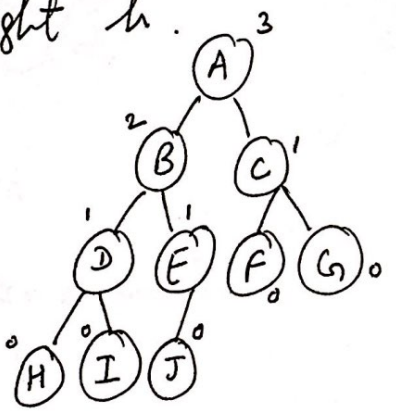
(f)

Build max Heap Analysis

Each call to max_Heapify cost $O(\log n)$ time and there are $O(n)$ such calls. Thus we can say the time taken for Build_Max_Heap is $O(n \log n)$, but this is not asymptotic tight upper bound.

'n' element heap has height $\lfloor \log_2 n \rfloor$ (since heap is considered as complete Binary Tree (almost)).

'n' element heap has atmost $\left\lceil \frac{n}{2^{h+1}} \right\rceil$ nodes of any height 'h'.



$n = 10$

$h = 0$ (F, G, H, I, J). $= \left\lceil \frac{10}{2^1} \right\rceil = 5$ nodes

$h = 1$ (C, D, E) $= \left\lceil \frac{10}{2^2} \right\rceil = \lceil 2.5 \rceil = 3$ nodes

$h = 2$ = (B) $= \left\lceil \frac{10}{2^3} \right\rceil = \lceil 1.25 \rceil = 2$ nodes (atmost)

$h = 3$ (A) $= \left\lceil \frac{10}{2^4} \right\rceil = \lceil 0.6 \rceil = 1$ node

Therefore time required by Max_Heapify when called on a node of height 'h' is $O(h)$.

Thus the cost of Build_Max_heap can be expressed as,

$$\sum_{h=0}^{\lfloor \log n \rfloor} \left\lceil \frac{n}{2^{h+1}} \right\rceil O(h) = O\left(\frac{n}{2} \sum_{h=0}^{\lfloor \log n \rfloor} \frac{h}{2^h} \right)$$

The running time of Build_Max_Heap can be bounded as

$$O\left(n \sum_{h=0}^{\lfloor \log n \rfloor} \frac{h}{2^h}\right)$$

$$\leq O\left(n \sum_{h=0}^{\infty} h\left(\frac{1}{2}\right)^h\right)$$

$$\leq O(n.2)$$

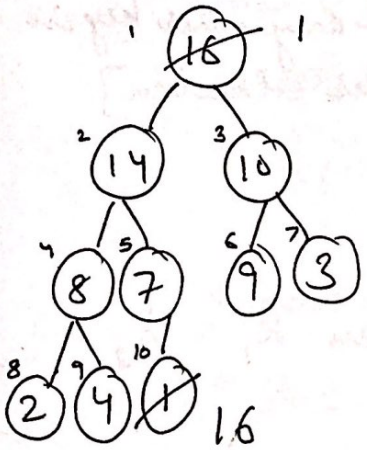
$$= O(n)$$

$$\left[\sum_{K=0}^{\infty} K x^K = \frac{x}{(1-x)^2} \text{ for } |x|<1.\right.$$

$$\left.\therefore \sum_{h=0}^{\infty} h\left(\frac{1}{2}\right)^h = \frac{\frac{1}{2}}{\left(1-\frac{1}{2}\right)^2} = 2.\right]$$

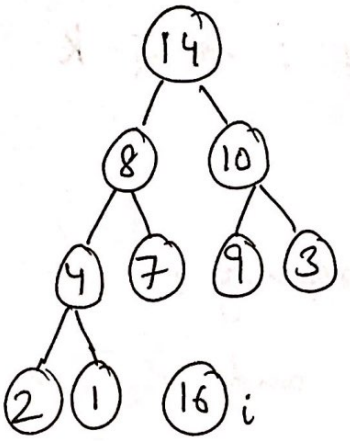
HeapSort (A)

1. Build_Max_Heap (A)
2. for i ← length(A) down to 2
3. do exchange A[1] ↔ A[i]
4. Heapsize [A] ← heapsize[A] − 1
5. Max_Heapify (A, 1)

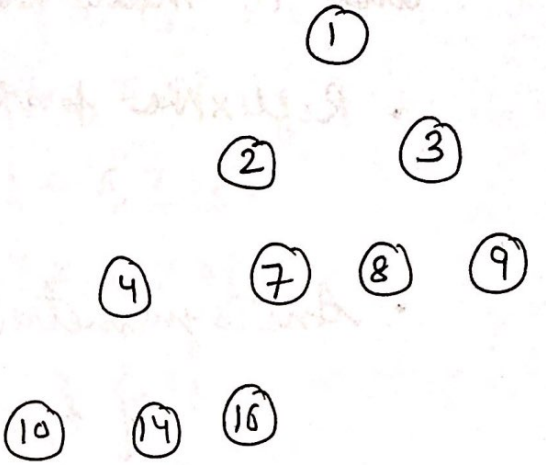


swap index elements 1 and 10.

(a)



(b)



A	1	2	3	4	7	8	9	10	14	16

(c)

Analysis.

Since call to Build_Max_Heap takes $O(n)$ time and each of the 'n−1' calls to max_Heapify takes $O(\log n)$ time.

∴ The Heapsort Procedure takes:-

$O(n \log n)$ time.

Priority Queues (Total order relations)

A priority queue needs a comparison rule that never contradicts itself. Comparison denoted by $\leq$ is known as total order relation.

This rule '$\leq$' is defined for every pair of keys and it must satisfy the following properties:

- Reflexive property : $k \leq k$ [k - any other key in our collection]

- Antisymmetric property :

  If $k_1 \leq k_2$ and $k_2 \leq k_1$, then $k_1 = k_2$

- Transitive property :

  If $k_1 \leq k_2$ and $k_2 \leq k_3$, then $k_1 \leq k_3$.

Priority Queue supports the following functions:-

1) size() :- returns no. of elements in P.
2) empty() :- returns true if P is empty & false otherwise
3) insert(e) :- Insert a new element e into P.
4) min() :- Returns a reference to an element of P with smallest value. (do not remove it).
5) removeMin() :- remove the element referenced by min().

Eg. series of operations and their effects on an initially empty priority queue. P.

operation	output	Priority queue.
insert (5)	–	{5}
insert (9)	–	{5, 9}
insert (2)	–	{2, 5, 9}
insert (7)	–	{2, 5, 7, 9}
min()	[2]	{2, 5, 7, 9}
removeMin()	–	{5, 7, 9}
size()	3	{5, 7, 9}
min()	[5]	{5, 7, 9}
remove_Min()	–	{7, 9}
remove_min()	–	{9}
remove_min()	–	{ }
empty()	true	{ }
remove_Min()	"error"	{ }

Priority Queues:- Queues used for storing prioritized elements that support arbitrary element insertion but removal of elements in priority.

One of the most popular application of heap is its use as priority Queues. Two kinds -

- Max Priority Queue
- Min Priority Queue.

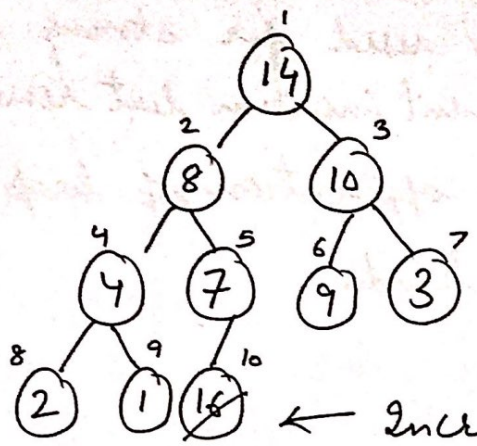
A max Priority Queue supports the following operations :-

① **Heap-Increase Key (A, i, Key)**

Increases the value of element i's key to the new value key, which is assumed to be at least as large as i's current key value.

```
Heap-Increase Key (A, i, Key)
1. If  Key < A[i]                                    :-
2.        then error "new Key is smaller than current Key"
3. A[i] ← Key.
4. while i > 1  &&  A[Parent(i)] < A[i]
5.        do exchange A[i] ↔ A[Parent(i)]
6.           i ← Parent.
```

The running time of Heap-Increase Key on an n-element heap is $O(\log n)$, since the path traced from the node updated in line 3 to the root has length $O(\log n)$



← Increase Key to 26.

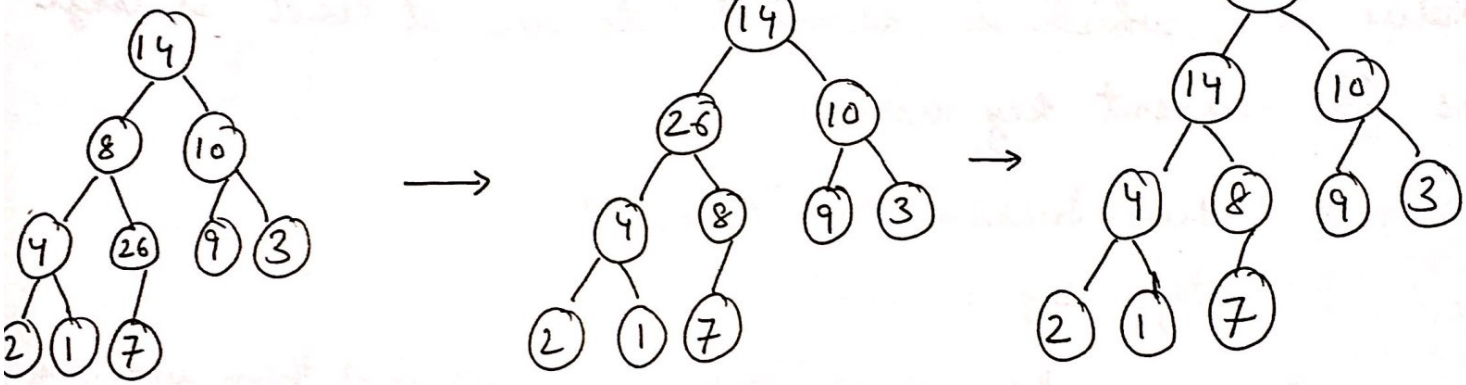
f^n call :- Heap Increase Key (A, 10, 26)

Here 26 > A[10]

exchange 26 ↔ 7

26 ↔ 8

26 ↔ 14.

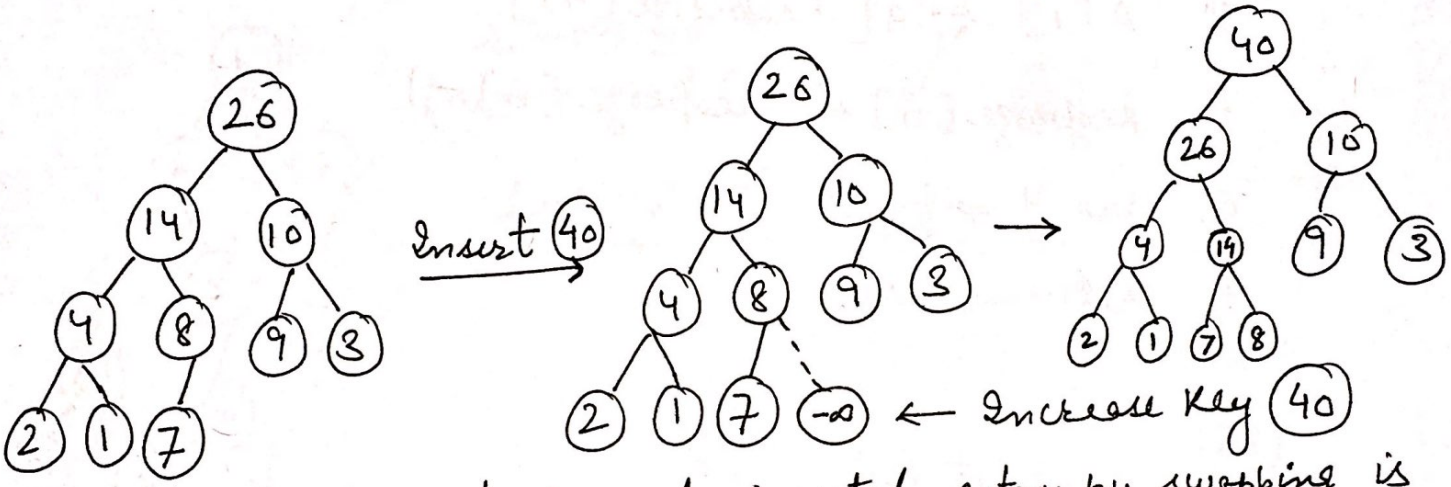


(2) Max-Heap Insert (A, Key)

- Insert the element 'Key' into the array 'A'.

Max-Heap-Insert (A, Key)

1. heap-size [A] ← heap-size [A] + 1
2. A [heap size [A]] ← $-\infty$
3. Heap-Increase Key (A, heapsize [A], Key)



The upward movement of newly inserted entry by swapping is known as up-heap bubbling.

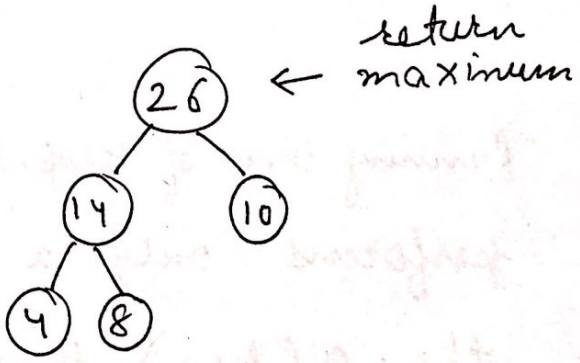
The running time of max-Heap Insert on an n-element heap is O (log n).

(3) Heap Maximum (A) :- returns the element of A with the largest key.

Heap Maximum (A)

1. return A [1]

Analysis :- O(1).

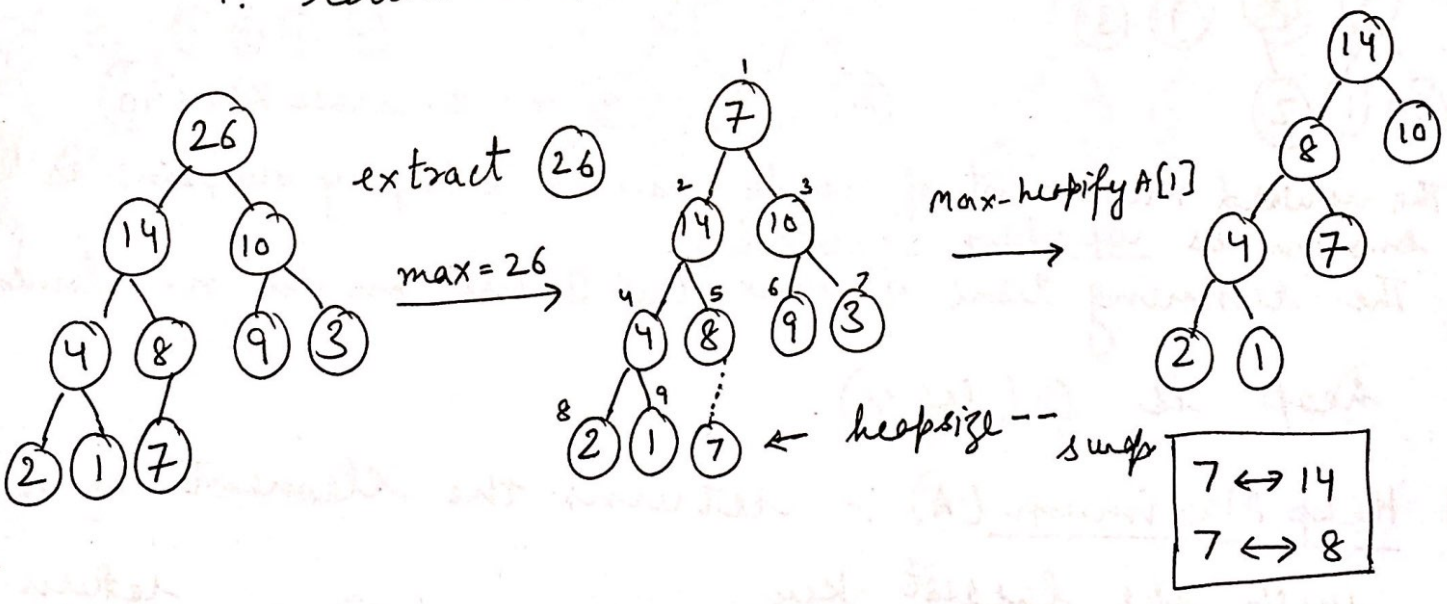


(4.) Heap-Extract Max(A):

removes and returns the element of A with the largest key.

Heap-Extract Max(A)

1. If heap-size[A] < 1
2. then error "heap underflow"
3. max ← A[1]
4. A[1] ← A[heap-size[A]]
5. heapsize[A] ← heapsize[A] − 1
6. Max-Heapify(A, 1)
7. return max.



Running time of Heap-Extract max is $O(\log n)$, since it performs only a constant amount of work on top of the $O(\lg n)$ time for Max-Heapify.

Similarly,

A min Priority Queue supports the following operations

1) Heap-Decrease key
2) Min-Heap Insert
3) Heap-minimum
4) Heap extract min. (Also known as Down Heap Bubbling)