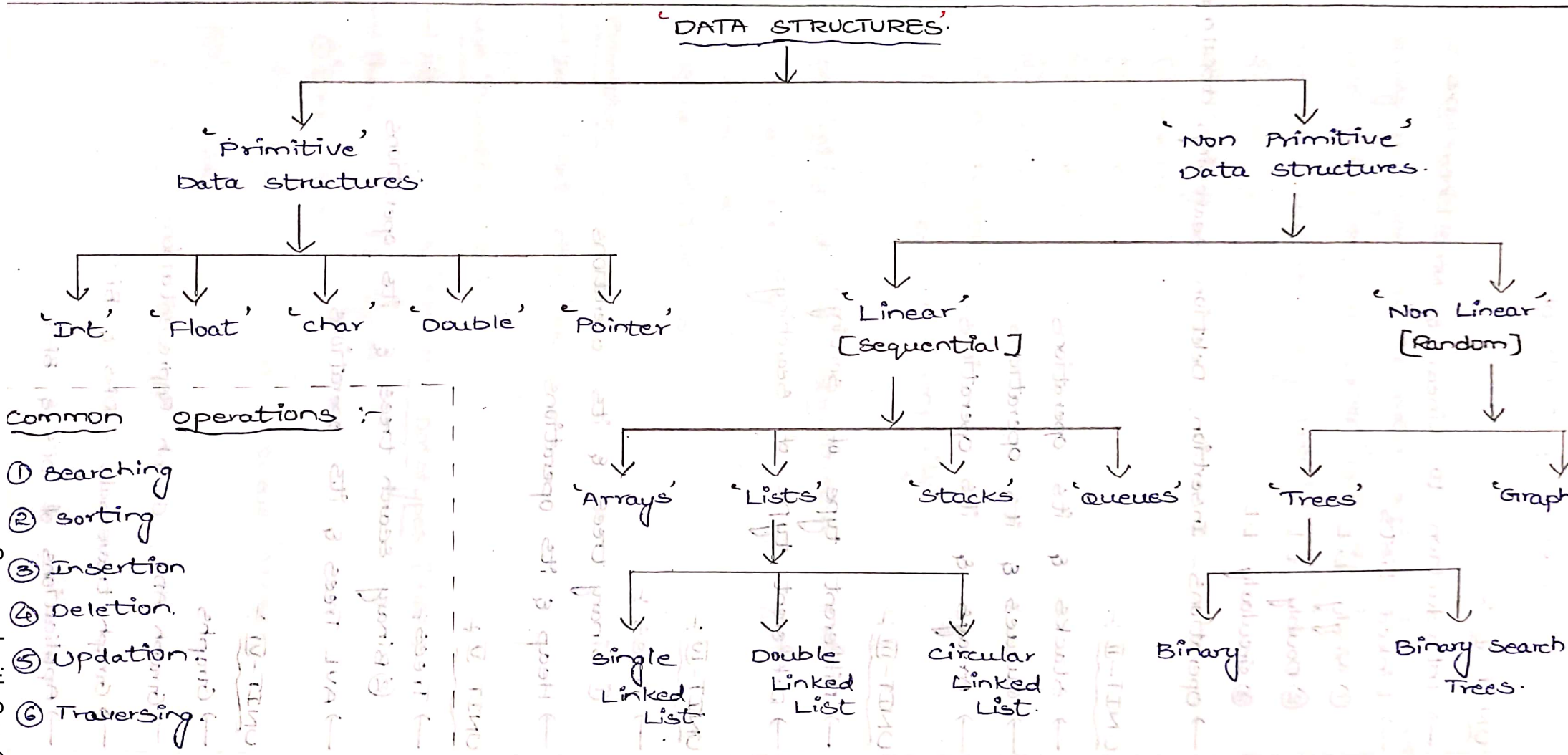
DATA STRUCTURES



Common operations :-

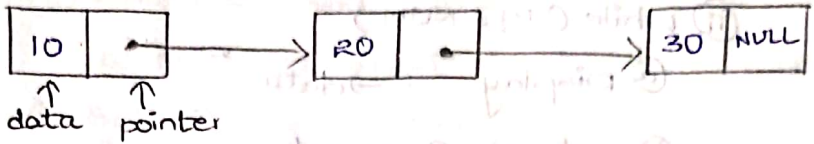
1. Searching
2. Sorting
3. Insertion
4. Deletion
5. Updation
6. Traversing

03-03-2023

UNIT - I

* Single Linked List :-

→ contains one Node with two fields [Data field and Node type of pointer field].



* Defining a single Linked List :-

```
struct node
{
    int data;
    struct node * next; // self Referential structure member.
} *start=NULL, *temp, *p;
```

*start :- Always points to starting node.
*temp :- It is used to create/delete a node.
*p :- It is used for traversing.

* Algorithm for creation of single Linked List :-

Step-1 :- create a new node i.e, temp.

[temp = (struct node *) malloc (size of (struct node));

Step-2 :- Input the data to be inserted [using x].

step-3 :- set temp → data = x ; temp → next = NULL ;

Step-4 :- if (start == NULL) then

(i) set start = temp.

(ii) set P = temp

step-5 :- else

(i) P → next = temp.

(ii) set P = temp.

Step-6 :- If you want create one more node repeat steps 1-5 otherwise exist.

* Algorithm for Traversal of S.L.L :-

step-1 :- if (start == NULL) then

(i) Display "Linked List is Empty"

step-2 :- else

(i) set P = start

(ii) while (P != NULL)

(a) Display "P → data"

(b) set P = P → next.

step-3 :- Exit.

* Insertion :-

* Algorithm for Inserting a node at beginning of the single Linked List :-

step-1 :- Create a new Node i.e. temp.

[temp = (struct node *) malloc (size of (struct node));]

step-2 :- Input the data to be inserted [using x variable]

step-3 :- set temp → data = x., temp → next = NULL.

step-4 :- if (start == NULL) then

(i) set start = temp.

step-5 :- else

(i) set temp → next = start

(ii) set start = temp.

step-6 :- If we want insert another node at begining repeat steps 1 to 5 otherwise Exit.

* Algorithm for Inserting a node at end of the single Linked List :-

Step-1 :- create a new node i.e, temp.

step-2 :- Input the data to be inserted [using x variable];

step-3 :- set temp→data = x, temp→next = NULL

step-4 :- if (start == NULL) then

(i) set start = temp

step-5 :- else

(i) set P = start

(ii) while (P→next != NULL)

(a) set P = P→next.

(iii) Set P→next = temp

step-6 :- If you want insert another node at ending of single Linked List then Repeat from steps 1-5 otherwise EXIT.

* Algorithm for inserting a node at specified position of the single Linked List :-

step-1 :- create a new node i.e temp.

step-2 :- Input the data to be inserted [using x variable]

step-3 :- set temp→data = x, temp→next = NULL

Step-4 :- if (start == NULL) then

(i) set start = temp.

step-5 :- else

(i) set count = 0

(ii) P = start

(iii) while (P != NULL)

(a) set count = count + 1

(b) set P = P→next.

(iv) Input the position of inserting value [using position variable].

(v) if (position == 1)

(a) call insert_node_beg_sLLC()

(vi) if (position == count + 1)

(a) call insert_node_end_sLLC()

(vii) else if (Position > 1 and Position <= count)

(a) set P = start

(b) for (k = 1; k < Position - 1; k++)

(A) set P = P→next.

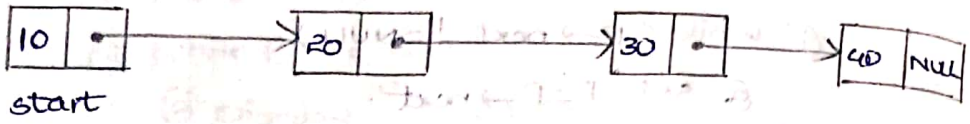
(c) set temp→next = p→next

(d) Set P→next = temp

(iii) else
  (a) Display "Invalid position"

step-6 :- EXIT.

* DELETION.



* Algorithm for Deleting a Node at Begining of S.L.L

step-1 :- if (start == NULL) then
  (i) Display "L.L is Empty"

step-2 :- else
  (i) Set temp = start.
  (ii) Set start = start → next.
  (or)
  Set start = temp → next.
  (iii) set temp → next = NULL.
  (iv) Delete temp [free (temp);]

step-3 :- EXIT.

* Algorithm for Deleting a Node at end of the S.L.L :-

step-1 :- if (start == NULL) then
  (i) Display "L.L is Empty"
  (ii) EXIT.

step-2 :- set p = start.

step-3 :- while (p → next → next != NULL)
  (i) set p = p → next

step-4 :- (ii) set p.temp = p → next.

step-5 :- set (p → next = NULL.

step-6 :- Delete temp.

step-7 :- EXIT

step :- else if (start → Next = NULL)
  (i) set temp = start
  (ii) set start = NULL
  (iii) Delete(temp)
  (iv) EXIT.

* Algorithm for Deleting a Node at specified position of S.L.L :-

step-1 :- if (start == NULL)

(i) Display "Linked List is Empty"

step-2 :- else if (start→next = NULL)

(i) Set temp = start
(ii) set start = start→Next
(iii) Delete temp.

} No need.

step-3 :- else

(i) set count = 0.
(ii) set P = start
(iii) while (P != NULL)
  (a) Set count = count + 1;
  (b) set P = P→next.
(iv) input the position of the deleting Node. using 'pos' variable
(v) if (pos == 1)
  (a) call delete_node_beg_SLL ( ).
(vi) else if (pos == count)
  (b) call del_node_end_S.L.L ( ).
(vii) else if (posint > 1 and posint < count)
  (a) set P = start.
  (b) for (k = 1; k < pos - 1; k++)
    (A) set P = P→next
  (c) set temp = P→next.
  (d) set P→next = temp→next.
  (e) Set temp→next = NULL.
  (f) Delete (temp)
(viii) else
  (a) Display "Entered position is Invalid".

step-4 :- EXIT.

SEARCHING

* Algorithm for searching :-

void searching (int x)
int flag=0;

step-1 :- if (start == NULL) then
  (i) Display "Linked List is Empty"

step-2 :- else
  (i) p = start
  (ii) while (p != NULL)
    (a) if (p → data == x)
      (A) set flag = 1.
      (B) break;
    (b) p = p → next.
  (iii) if (flag == 1)
    (a) Display "search is successful."
  (iv) else
    (a) Display "Element is not found."

step-3 :- EXIT.

SORTING

* Algorithm for sorting :-

*temp1, *temp2.

step-1 :- if (start == NULL)
  (i) Display "LL is Empty."

step-2 :- else if (start → next = NULL)
  (i) Display "LL having single Node."

step-3 :- else
  (i) for (temp1 = start; temp1 != NULL; temp1 = temp1 → next)
    (a) for (temp2 = start → next; temp2 != NULL; temp2 = temp2 → next)
      (A) if (temp1 → data < temp2 → data)
        (I) set x = temp1 → data.
        (II) set temp1 → data = temp2 → data
        (III) set temp2 → data = x.

step-4 :- EXIT.

REVERSING A SINGLE L.L

*P1, *P2, *P3.

* Algorithm for Reversing a single Linked List :-

step-1 :- if (start == NULL)

(i) Display "Linked List is empty".

step-2 :- else if (start → next = NULL).

(ii) Display "Linked List contains single Node".

step-3 :- else

(i) set P_1 = start.

(ii) set $P_2 = P_1 \rightarrow next$

(iii) set $P_3 = P_2 \rightarrow next$

(iv) set $P_1 \rightarrow next = NULL$.

(v) set $P_2 \rightarrow next = P_1$

(vi) while (P_3 != NULL)

(a) set $P_1 = P_2$

(b) set $P_2 = P_3$

(c) Set $P_3 = P_2 \rightarrow next$ (or) $P_3 \rightarrow next$.

(d) set $P_2 \rightarrow next = P_1$

(vii) set start = P_2

step-4 :- EXIT.

CONCATENATION

* Algorithm for concatenation of two S.L.L :-

struct node * creation (struct node * start).

step-1 :- set start1 = creation (start1). *start1 = NULL;

step-2 :- Set start2 = creation (start2). *start2 = NULL.

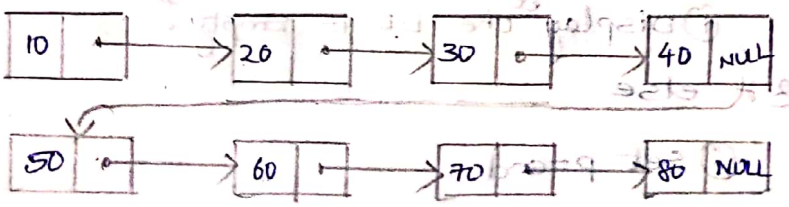
step-3 :- set p = start1

step-4 :- while (P → next != NULL)

(a) P = P → next.

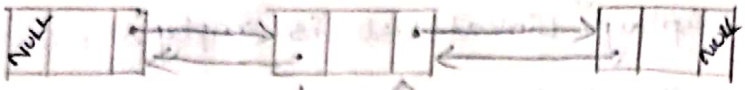
step-5 :- Set P → next = start2.

step-6 :- EXIT.



** DOUBLE LINKED LIST

→ It contains Nodes. one Node having three fie
[one data field and two pointer fields].

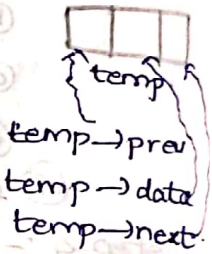


* Defining a D.L.L:-

```
struct node
{
  struct node *prev;
  int data;
  struct node *next;
} *start=NULL, *end, *P, *temp;
```

* Algorithm for creating of D.L.L:-

step-1:- create a new node i.e, temp.



step-2:- Input the data to be inserted [using x].

step-3:- set temp→data=x,
set temp→prev=NULL,
set temp→next=NULL.

step-4:- if (start==NULL) then
(i) set start=temp
(ii) set end=temp

step-5:- else
(i) set end→next=temp.
(ii) set temp→previous=end.
(iii) set end=temp.

step-6:- EXIT.

* Algorithm for Traversal of D.L.L in Backward directi

step-1:- if (start==NULL)
(i) Display "the L.L is Empty."

step-2:- else
(i) set p=end

(ii) while (P != NULL)

(a) Display "P→data"

(b) Set P = P→prev.

Step-3 :- EXIT.

* Algorithm for Traversal of D.L.L in Forward Direction :-

step-1 :- if (start == NULL)

Display The L.L is Empty.

step-2 :- else

(i) set p = start

(ii) while (P! = NULL)

(a) Display P→data

(b) set P = P→next

step-3 :- EXIT.

INSERTION

* Algorithm for Inserting a Node at begining in D.L.L :-

step-1 :- create a new node i-e temp

step-2 :- Input the data to be inserted [using x]

step-3 :- set temp→data = x,

set temp→prev = NULL,

set temp→next = NULL.

step-4 :- if (start == NULL) then

set start = temp

set end = temp.

step-5 :- else

(i) set temp→next = start

(ii) set start→prev = temp.

(iii) set start = temp.

step-6 :- EXIT.

* Algorithm for inserting a Node at end of D.L.L:

step-1 to step-4 :- same as creation.

step-5 :- else

(i) set end → next = temp

(ii) set temp → prev = end

(iii) set end = temp

step-6 :- EXIT.

* Algorithm for Inserting a Node at specified position in D.L.L :-

step-1 :- create a New node temp.

step-2 :- Input the data to be inserted using x.

step-3 :- set temp → data = x,
set temp → prev = NULL,
set temp → next = NULL.

step-4 :- if (start == NULL).

(i) set start = temp

(ii) set end = temp

step-5 :- else

(i) set count = 0

(ii) set P = start

(iii) While (P != NULL)

(a) set (count = count++)

(b) P = P → next

(iv) Input the position of the value to be inserted (using pos)

(v) if (pos == 1)

(a) call insert_node_begining_dll ( ).

(vi) else if (pos == count+1)

(a) call insert_node_end_dll ( ).

(vii) else if (pos > 1 and pos <= count).

(a) set P = start

→

(b) for (k=1; k<pos-1; k++)

(A) set p=p→next;

(c) set temp→next = p→next;

(d) set p→next→prev = temp

(e) set p→next = temp

(f) set temp→prev = p

(viii) else

Display "Invalid position."

step-6: EXIT.

DELETION

* DELETION at Begining :-

* Algorithm for Deletion of a Node at begining of D.L.L:-

step-1 :- if (start == NULL)

(i) Display "L.L is Empty"

step-2 :- else.

(i) Set temp = start

(ii) Set start = start→next (or) temp→next

(iii) set start→prev = NULL

step-3 (iv) free (temp)

step-4 :- EXIT.

* Deletion at ending of a D.L.L :-

Step-1 :- If (start==NULL)

(i) Display "L.L is Empty".

step-2 :- else if (start→next ==NULL)

(i) set temp = start

(ii) set start = NULL

(iii) free (temp)

step-3: ELSE.

(i) set temp = start

(ii) while (temp→next != NULL)

(A) temp = temp→next

(iii) temp→prev→next = NULL

(iv) free (temp).

step-4: EXIT.

* Algorithm for Deletion of a Node at specified position in Doubly Linked List :-

Step-1 :- if (start == NULL)

(i) Display "Linked List is empty."

Step-2 :- else.

(i) set count = 0

(ii) set P = start

(iii) while (P != NULL)

(a) set (count = count++)

(b) P = P → next

(iv) Input the position to be deleted (using pos variable).

(v) if (pos == 1)

(a) call del_node_begin( );

(vi) else if (pos == count+1)

(a) call del_node_end_DLLC( );

(vii) else if (Pos > 1 and Pos <= count)

(a) set P = start

(b) for (k=1 ; k <= Pos-1 ; k++)

(A) set P = P → next.

(c) set temp = p → next

(d) set P → next = temp → next;

(e) set temp → next → prev = P.

(f) Delete temp.

(viii) else

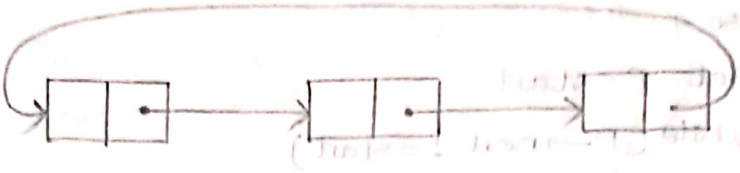
(a) Display "Invalid position."

Step-3 :- EXIT.

Circular Linked List (CLL).

* Single circular Linked List [S.C.L.L]

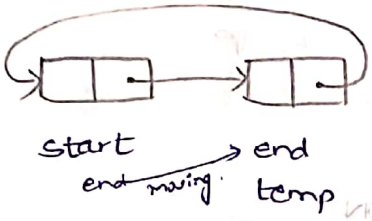
→ Node consists of two fields [Data, Pointer fields].



For single Node :-

start→next = start
(or)
end→next = start.

For Two nodes :-



start→next = temp
temp→next = start.

* Algorithm for creating single circular List :-

~~Step-1~~ :-

structure :-

```
Struct node
{
   int data;
   Struct node *next;
} *start=NULL, *temp, *p/*end;
```

Step-1 :- create a New node i.e temp.

Step-2 :- Input the data to be inserted (using x).

Step-3 :- Set temp→data = x, temp→next = NULL.

Step-4 :- if (start == NULL) then

(i) Set start = temp.

(ii) Set start→next = start.

(iii) Set end = temp/start.

Step-5 :- else

(i) Set end→next = temp.

(ii) Set temp→next = start.

(iii) Set end = temp.

Step-6 :- EXIT.

* Algorithm for Traversal of S.C.L.L :-

Step-1 :- if (start == NULL) then

(i) Display "L.L is Empty."

Step-2 :- else

(i) Set P = start.

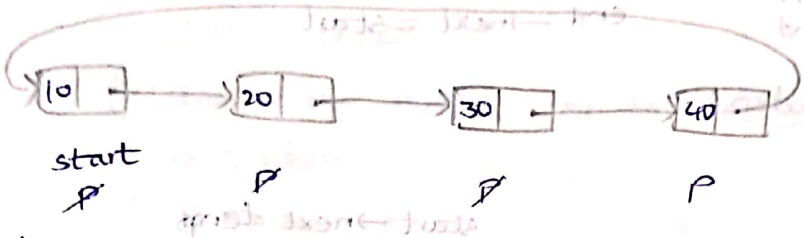
(ii) while (P → next != start)

(a) Display P → data

(b) set P = P → next.

step-3 :- ~~Exit~~ Display p → data

step-4 :- EXIT.



O/P :- 10 20 30 40.

INSERTION

* Algorithm to insert a Node at begining of S.C.L.L

Step-1 to 4 :- Same as creation.

Step-5 :- else.

(i) set temp → next = start.

(ii) set end → next = temp.

(iii) set (~~temp →~~) start = temp

(or)

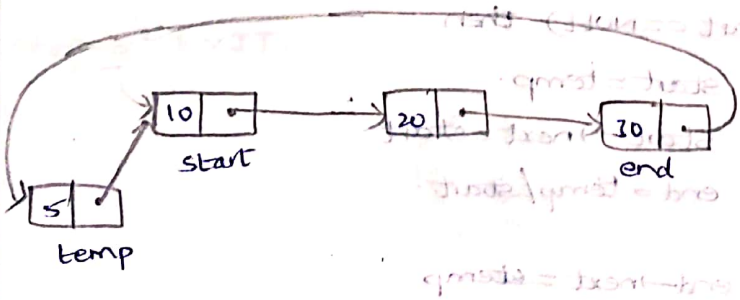
Step-5 :- else

(i) Set temp → next = start

(ii) set start = temp

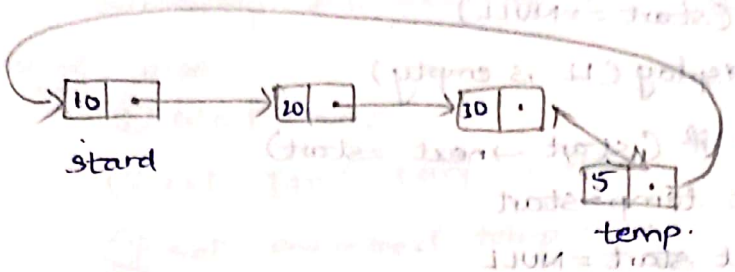
(iii) set end → next = start

Step-6 :- EXIT



* Algorithm to insert a node at end of S.C.L.L:

step-1 to 3 :- same as creation of S.C.L.L.



* Algorithm to insert a Node at specific position of SCLL:

step-1 to 4 :- same as creation

step-5 :- else

(i) set count = 1

(ii) set P = start

(iii) while (P->next != start)

(a) count = count + 1

(b) set P = P->next;

(iv) Input the position of the Node to be inserted. [using pos]

(v) if (pos == 1)

(a) call insert_begin_S.C.L.L();

(vi) else if (pos <= count+1)

(a) call insert_end_S.C.L.L();

(vii) else if (Pos > 1 and Pos <= count)

(a) set P = start

(b) for(k=1; k<pos-1; k++)

(A) set P = P->next

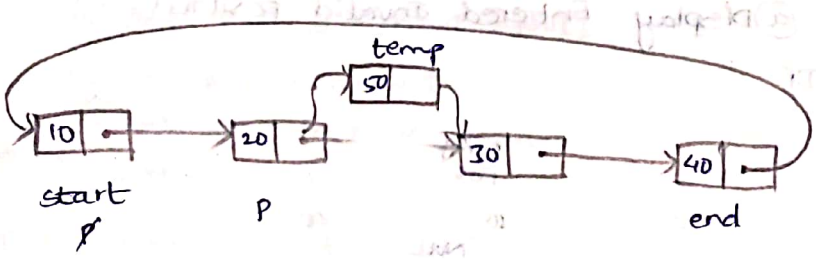
(c) set temp->next = P->next

(d) set P->next = temp.

(viii) else

(a) Display Entered Invalid position.

step-6 :- EXIT.



DELETION

* Algorithm for Deleting a Node at specified position in a

step-1 :- if (start == NULL)

(i) Display CLL is empty)

step-2 :- else if (start → next = start)

(i) set temp = start

(ii) set start = NULL

(iii) Set temp → next = NULL

(iv) Delete temp.

step-3 :- else.

(i) set count = 1

(ii) set p = start

(iii) while (p → next != start)

(a) count = count + 1

(b) set p = p → next

(iv) Input the position of Node to be deleted [using p

(v) if (pos == 1)

(a) call del_beg_sCLLC);

(vi) else if (pos == count)

(b) call del_end_sCLLC);

(vii) else if (pos > 1 and pos < count)

(a) Set p = start

(b) for (k = 1; k < pos - 1; k++)

(A) set p = p → next.

(c) set temp = p → next,

(d) set p → next = temp → next

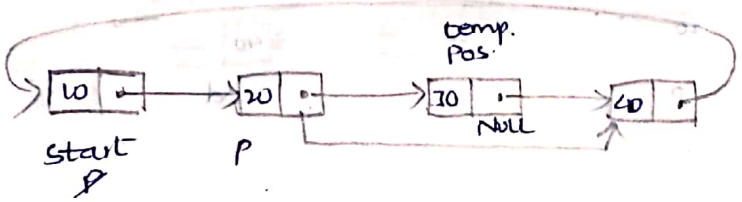
(e) set temp → next = NULL

(f) Delete temp.

(viii) else

(a) Display Entered Invalid position.

step-4 :- EXIT.



* Algorithm for Deleting a Node at begining:-

step-1 :- create a Ne: if (start == NULL)

(a) Display "Linked List is empty."

step-2 :- else

(i) start = temp

(ii) set start = temp->next

(iii) set end->next = temp->next.

(iv) Delete temp

step-4 :- EXIT.

(or)

step-1 :- if (start == NULL).

(i) Display 'Linked List is Empty'

step-2 :- else if (start->Next == start)

(i) start = NULL

(ii) free (start)

step-3 :- else

(i) set p = start

(ii) while (P->next != start)

(a) p = p->next

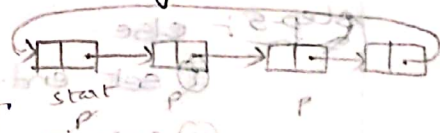
(iii) set P->next = start->next

(iv) Delete (start)

(v) Set start = P->next

step-4 :- EXIT.

* Algorithm for Deletion of a Node at ending in sCLL:-



step-1 :- if (start == NULL)

(i) Display Linked List is empty.

step-2 :- else if (start->next == start)

(i) start = NULL

(ii) Delete (start).

step-3 :- else

(i) set P = start

(ii) while (P->next->next != start).

(a) P = P->next

(iii) set P->next->next = NULL.

(iv) set P->next = start.

(v) Set temp = P->next->next

Step-4 :- Delete temp.

Step-5 :- EXIT.

20-03-2023

UNIT - 2. STACKS AND QUEUES

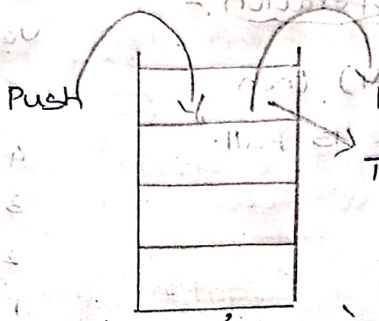
* Stack :-

→ Stack is also known 'LIFO' Data structure.

→ LIFO :- 'Last In First out'

→ For Inserting we use 'Push operation'.

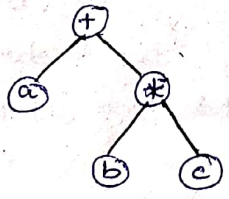
→ For Deletion we use 'Pop operation'.



→ We also use 'peek' and 'Traversal' operations in stacks.

* Applications of stacks :-

1. Recursion.
2. Conversion of Expressions
   Ex :- ① Infix into prefix/postfix
   - 4*5 } Infix expression.
   - a*b }
   - *ab → Prefix expression
   - ab* → postfix expression.
3. Expression Trees.



* Methods for Implementing stacks :-

→ Two ways :-

1. By using 'Arrays'.
2. By using 'Linked List'.

→ For Four operations

① Push ② Pop ③ Peek ④ Traversal.

① Implementing stacks using Arrays :-

→ Initially we initialize Top End as '-1' i.e top = -1.

→ We use Top, s[MAX], x

ⓐ Top :- Top is a 'Index variable' initially 'Top = -1'

ⓑ s[MAX] :- 's' means Array Name and 'MAX' is size.

ⓒ x :- To Read the Input.

[using symbolic constant]

Global
Top = -1, s[MAX]

* Algorithm for Push operation :-

Void Push (int x);

Step-1 :- if (top == MAX-1) then

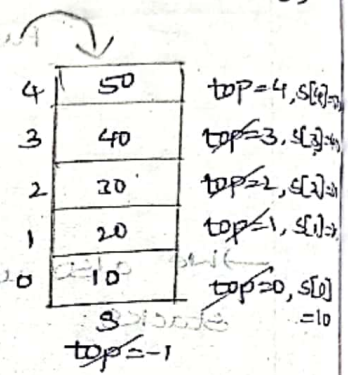
ⓘ Display "stack is Full."

Step-2 :- else

ⓘ Set top = top+1.

ⓘⓘ Set s[top] = x.

Step-3 :- EXIT.



* Algorithm for Pop operation :-

Step-1 :- if (top == -1) then

ⓘ Display "stack is Empty / underflow"

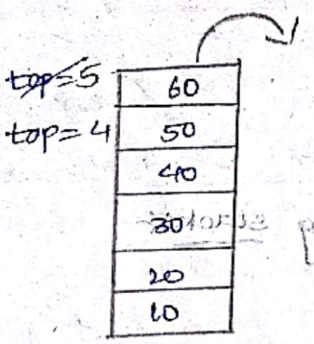
Step-2 :- else

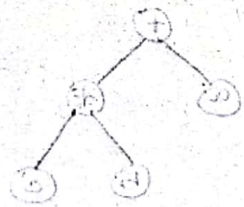
ⓘ Display "Deleting Element is s[top]"

ⓘⓘ Set top = top-1

Step-3 :- EXIT.

* Example :-





s[5] = 60 is Deleted. 4 = 5

push(100) Now top = top+1 4+1 = 5 [∵ top = top+1]

s[5] = 100.

⇒ 4 = 4+1 = 5

pop( ), s[5] = 100 deleted

pop( ), s[4] = 50 deleted

s[0] = 10 deleted then top = -1

* Algorithm for Traversal of stacks:-

step-1:- if (top==-1)

(i) Display "stack is underflow/Empty."

step-2:- else

(i) for(i=top; i>=0; i--)

(a) Display "s[i]."

step-3:- EXIT.

* Algorithm for Peek operation:-

step-1:- if(top==-1)

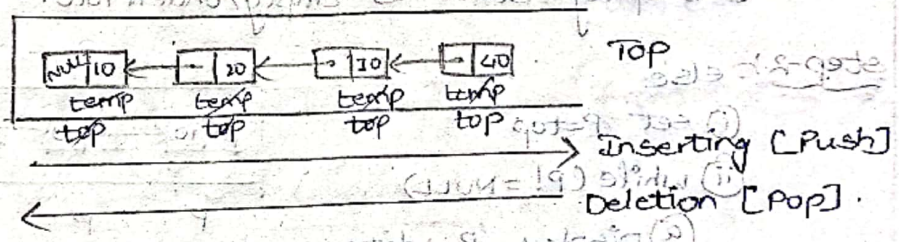
(i) Display "stack is Empty/underflow."

step-2:- else

(i) Display "s[top]"

step-3:- EXIT.

II Implementation of stacks using Linked List:-



structure:-

```
struct node
{
  int data;
  struct node *next;
} *top=NULL, *p, *temp;
```

(*) Algorithm for Push operation:-

step-1:- create a new Node ie temp

step-2:- Input the data to be inserted [using x]

step-3:- (i) Set temp->data=x,

(ii) Set temp->next=NULL.

step-4:- if (top==NULL)

(i) set top=temp.

step-5:- else

(i) set temp→next = top

(ii) Set top = temp.

step-6:- EXIT.

(2) Algorithm for Pop operation :-

step-1:- if (top == NULL) then

(i) Display "stack is Empty."

step-2:- else.

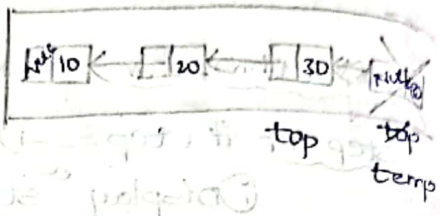
(i) set temp = top.

(ii) set top = top→next.

(iii) set temp→next = NULL.

(iv) Delete temp.

step-3:- EXIT.



(3) Algorithm for Traversal of stacks :-

step-1:- if (top == NULL) then

(i) Display "stack is Empty/underflow."

step-2:- else

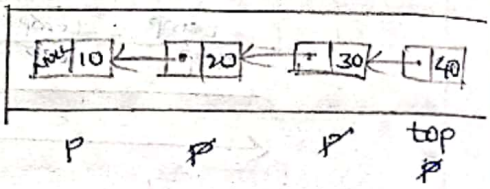
(i) set P = top

(ii) while (P != NULL)

(a) Display P→data

(b) set P = P→next

step-3:- EXIT.



o/p :- 40, 30, 20, 10

(4) Algorithm for Peek operation :-

step-1:- if (top == NULL) then

(i) Display "stack is Empty"

step-2:- else

(i) Display "top→data"

step-3:- EXIT.

* Applications of stacks:-

1] Reversing a string using stacks:-

step-1 :- Read a string using loop [characters one by one].

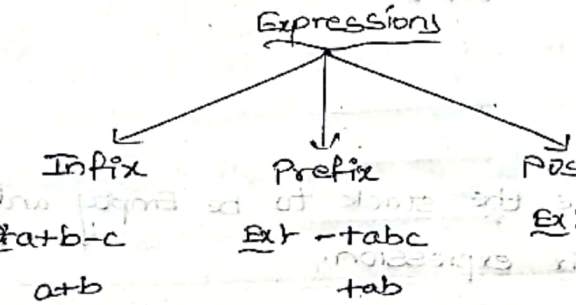
step-2 :- By using push operation, we add characters of given string one by one.

step-3 :- By using pop operation, we add characters one by one to another array. [called b].

step-4 :- Display the array [b].

Step-5 :- EXIT.

2]* Expressions



* Algorithm & Example for conversion from Infix to Prefix Expressions:

Ex:- a+b*c/d-f

⇒ (a+b*c/(d-f)). ⇒))f-d c/c*b+aC.

Input	stack	output
)	)	Empty
)	))	Empty.
f	))	f
-	)) -	f
d	)) -	fd
c	)	fd-
/	) /	fd-
c	) /	fd-c
*	) / *	fd-c
b	) / *	fd-cb
+	) *	fd-cb*/
a	) +	fd-cb*/a.
(	empty	fd-cb*/a+

→ We need to Reverse the output.

fd – cb * / a +

⇒ +a /*bc – fd

(b*c) (d–f)

x / y

+a z

a + z

check → Given exp. (a+b*c/(d–f))

x / y

a + z

a + z

* Algorithm :

Step-1 : Initialize the stack to be empty and reverse the given Infix expression.

Step-2 : Scan the given reverse expression from left to right character by character.

(i) If the input character is Right parenthesis ')' then push it onto the stack.

(ii) else if the input character is Alpha Numeric then append it to the output.

(iii) else if the input is left parenthesis '(' then

(a) Pop the all operators from the stack & append all the operators to output until right parenthesis encountered.

(b) Pop the right parenthesis ')' from the stack and discard it.

(iv) else if Top of stack is Right parenthesis ')' then push the input character onto the stack.

(v) else

(a) Repeat the loop until the stack is not empty and input character is lower precedence than the top of the stack.

(A) Pop the stack and append the popped elements into stack output.

(B) Push the Input Operator into the stack.

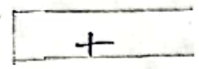
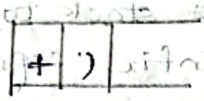
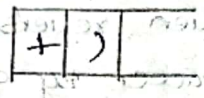
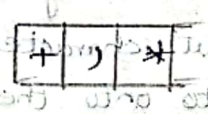
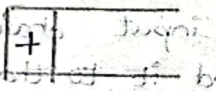
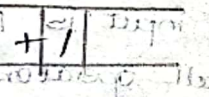
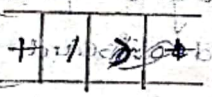
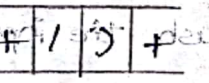
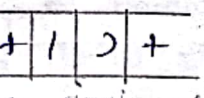
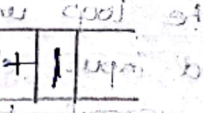
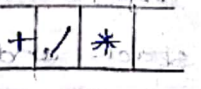
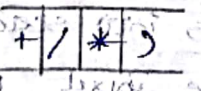
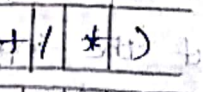
Step-3 : If the end of the input string is encountered then Iterate the loop until the stack is not empty. Pop the stack and append the remaining input string to the output.

Step-4 : EXIT.

* Example :-

Infix expression :- (A-B) *(C+D) / (E*F) + G.

Reversing :- G +) F * E (/) D + C (*) B - A (

Input	stack	output.
G	Empty	G
+		G
)		G
F		G F
*		G F E
(		G F E *
/		G F E *
)		G F E * D C
D		G F E * D
+		G F E * D
C		G F E * D C
(		G F E * D C +
*		G F E * D C +
)		G F E * D C +
B		G F E * D C + B
-		G F E * D C + B *
A		G F E * D C + B A
(		G F E * D C + B A -

* Queues :-

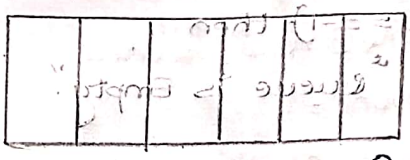
→ 'Queue' is a 'FIFO Data structure' [First In First Out].

→ Always 'Insertion operation' performs at 'Rear End' only.

→ Always 'Deletion operation' performs at 'Front-End' only.

→ 'Insertion operation' is also known as 'EnQueue operation'.

→ 'Deletion operation' is also known as 'Dequeue operation'.



Q[MAX]; → symbolic constant.

front = rear = -1

* Types of Queues :-

→ There are three types of queues.

① Linear Queue

② Circular Queue

③ 'Double Ended Queue'

① Implementation of Linear queue by Arrays :-

* Algorithm for Inserting an element into queue :-

step-1 :- if (rear ≥ MAX-1) then

① display "Queue is overflow".

step-2 :- else

① set rear = rear+1

② set Q[rear] = x.

step-3 :- if (rear == 0)

① set front = 0.

step-4 :- EXIT.

② * Algorithm for Deleting an element from queue:

step-1 :- if (front == -1) then
  (i) Display "Queue is underflow."

step-2 :- else
  (i) Display "Deleting element @[front]"
  (ii) set front = front+1

step-3 :- if (front > rear)
  (i) Set front = rear = -1

step-4 :- EXIT.

③ Algorithm for Traversal of queue :-

step-1 :- if (front == -1) then
  (i) Display "Queue is Empty"

step-2 :- else
  (i) for(i=front; i<=rear; i++)
    (a) Display "@[i]"

step-3 :- EXIT.

* Implementation of queues using Linked List :-

① Algorithm for inserting a node into queue :-

```
struct node
{ int data;
  struct node *next;
};* front=NULL, *rear=NULL, *p, *temp;
```

step-1 :- create a new node i.e. temp.

step-2 :- Input the data to be inserted using 'x' variable.

step-3 :- set temp->data=x;
  set temp->next=NULL.

step-4 :- if (front==NULL) then
  (i) set front=temp.
  (ii) set rear=temp.

step-5 :- else

(i) set rear → next = temp

(ii) set rear = temp

step-6 :- EXIT.

(2) Algorithm for Deleting a Node at begining :-

step-1 :- if (front == NULL)

Display "Circular queue is Empty."

step-2 :- else

(i) set temp = front

(ii) set front = front → next

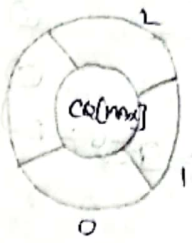
(iii) set temp → next = NULL;

* Circular Queue :-

Front = Rear = -1.

Front = (Front + 1) % MAX

Rear = (Rear + 1) % MAX.



(1) * Algorithm for Enqueue Operation (or) Inserting an Element into circular Queue :-

CQ[MAX], front = -1, rear = -1
void insert CQ(int data)

step-1 :- if (front == (rear+1) % MAX)
  (i) Display "circular queue is Full."

step-2 :- else
  (i) set rear = (rear+1) % MAX
  (ii) set CQ[rear] = data.
  (iii) if (front == -1)
    (a) set front = 0.

step-3 :- EXIT.

(2) Algorithm for Dequeue operation :-

step-1 :- if (front == -1) then
  (i) Display "circular queue is underflow."

step-2 :- else
  (i) Display "Deleting element CQ[front]."
  (ii) if (front = ~~front~~ = rear)
    (a) set front = -1
    (b) set rear = -1
  (iii) else
    (a) set front = (front+1) % MAX.

step-3 :- EXIT.

(3) Algorithm for Traversal of circular Queue :-

step-1 :- if (front == -1) then
  (i) Display "circular queue is Empty."

step-2 :- else
  (i) if (front <= rear)
    (a) for (i = front ; i <= rear ; i++)
      (A) Display "CQ[i]."

(ii) else.

(a) for (i=front; i<=MAX-1; i++)

(A) Display cq[i].

(b) for (i=0; i<=rear; i++)

(A) Display cq[i].

step-3:- EXIT.

(or)

step-3:- else (front<=rear)

(i) for(i=front; i<=rear; i++)

(a) Display "cq[i]".

(ii) else.

(a) i=front+1

(b) do

{

cq[i]

i=(i+1)%MAX

}

while(i!=rear)

(c) Display cq[rear]

step-3:- EXIT.

* Priority queue:-

(1) Ascending PQ.

(2) Descending PQ.

* Algorithm for Enqueue operation:-

Enqueue PQ(int data)

PQ[MAX], itemcount=0

step-1:- if (PQ is Full(-))

(i) if (item count == 0)

(a) set PQ[itemcount++] = data.

(ii) else

(a) for (i=itemcount-1; i>=0; i--).

(A) if (data > PQ[i])

(I) PQ[i+1] = PQ[i];

(B) else

(I) break;

(b) set PQ[i+1] = data;

(c) set itemcount = itemcount+1

step-2:- EXIT.

* Algorithm for Dequeue of PQ :-

step-1 :- if (! isempty( ))

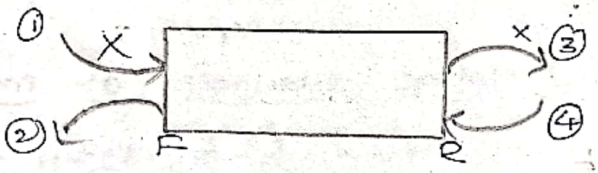
(i) return PQ [--itemcount].

step-2 :- EXIT.

* isfull & isempty :-

```
bool isFull ( )
{
   return itemcount==MAX;
}
bool isempty ( )
{
   return itemcount==0;
}.
```

* Double Ended Queue :-



→ The Double Ended queues are classified into

(1) Input Restricted. [Insert at Rear only]. ①, ③, ④ ✓

(2) output Restricted. [Deletion at Front only]. ①, ②, ④ ✓

* Algorithm for Enqueue operation at Front End :-

DQ[MAX], front = rear = -1

step-1 :- if (front==0)

Display "Insertion not possible at Front end."

step-2 :- else

(i) if (front==-1)

(a) set front = front+1

(b) DQ [front] = data

(c) if (rear == -1)

(A) set rear=0

(ii) else

(a) set front = front-1

(b) DQ [front] = data

step-3 :- EXIT.

* Algorithm for Deletion at Rear End :-

step-1 :- if (front == -1)

(i) Display "Queue is Empty."

step-2 :- else

(i) if (rear == front)

(a) set front = -1

(b) set rear = -1

(ii) else

(a) display Deleting element DQ[rear].

(b) set rear = rear - 1

step-3 :- EXIT.

* Applications of Queues :-

(1) Online Reservations.

(2) schooling

(3) Mails.

(4) Priority Queues.

12-04-2023.

UNIT - 3 : SEARCHING AND SORTING.

* Sorting

(1)* Insertion set Algorithm :-

InsertionSort(a, n):

Step-1 :- Declare i, j, key:

Step-2 :- for (i=1; i<n; i++)

(i) Set key = a[i]

(ii) for (j=i-1; j>=0 && a[j]>key; j=j-1)

(a) Set a[j+1] = a[j]

(iii) Set a[j+1] = key

Step-3 :- Exit.

② Selection Sort Algorithm :-

Selection_sort (a, n)

Step-1 :- Declare i, j, min index.

Step-2 :- for (i=0 ; i<n ; i++)

(i) set min index = i.

(ii) for (j=i+1 ; j<n ; j++)

(a) if (a[j] < a[min index])

(A) set min index = j.

(iii) swap (&a[i], & a[min index]);

Step-3 :- Exit.

③ Quick sort Algorithm :-

int Partition
Quick sort (a, l, h) → Higher index
array, lower index

step-1 :- Declare i, j, pivot;

step-2 :- set i = l-1

step-3 :- set pivot = a[h]

step-4 :- for (j = l; j <= h-1; j++)

ⓘ if (a[j] < pivot)

ⓐ i++

ⓑ swap (&a[i], &a[j]);

step-5 :- swap (&a[i+1], &a[h]);

step-6 :- return i+1.

Quicksort (a, l, h).

step-1 :- Declare PI [Partition Index];
step-2 :- if (l < h);
step-3 :- PI = Partition (a, l, h);
step-4 :- Quicksort (a, l, PI-1);
step-5 :- Quicksort (a, PI+1, h);
step-6 :- EXIT.

④ Merge sort Algorithm :-

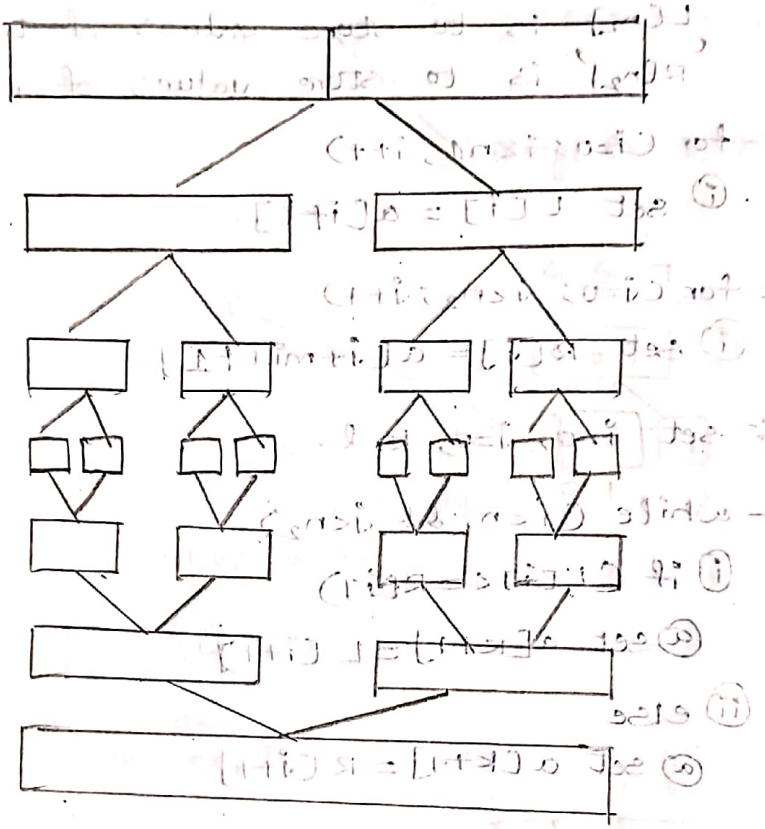
→ It comes under 'External sorting'.

→ External sorting needs 'Additional space' to store the bulk data.

→ The time complexity of merge sort is $n \log N$.

→ The time complexity of quick sort is order n^2.
and the heart of quick sort is 'Partition'.

→ The Heart of merge sort is 'Merging'. and Merge sort follows 'Divide and conquere rule' as follows.



Algorithm :-

Mergesort (a, l, h).

step-1 :- if (l<h)

(i) compute mid = (l+h)/2

(ii) mergesort (a, l, mid).

(iii) mergesort (a, mid+1, h)

(iv) merge (a, l, mid, h).

step-2 :- EXIT.

→ The key process in Mergsort is 'Merging'.

Algorithm for merge :-

merge (a, l, mid, h).

step-1 :- Declare i, j, k, n_1, n_2.

Here i, j, k are 'control variables'
n_1 is 'size of Left half'
n_2 is 'size of Right half'.

step-2 :- set $n_1 = mid - l + 1$. and $n_2 = h - mid$.

step-3 :- Declare L[n_1], R[n_2].

'L[n_1]' is to store values of Left half
'R[n_2]' is to store values of Right half.

step-4 :- for (i=0; i<n1; i++)

(i) set L[i] = a[i+l].

step-5 :- for (j=0; j<n_2; j++)

(i) set R[j] = a[j+mid+1].

step-6 :- set i=0, j=0, k=l.

step-7 :- while (i<n_1 && j<n_2)

(i) if (L[i] <= R[j])

(a) set a[k++] = L[i++].

(ii) else

(a) set a[k++] = R[j++].

step-8 :- while (i<n_1)

(i) a[k++] = L[i++].

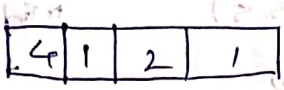
step-9 :- while (j<n_2)

(i) a[k++] = R[j++].

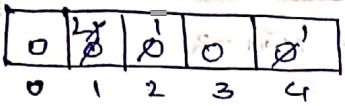
step-10 :- Display the elements of array.

step-11 :- EXIT.

⑤ Counting sort:-

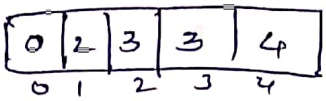


Counting array:-
Count [max+1]



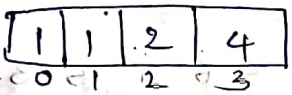
for(i=0; i<n; i++)
count[a[i]]++.

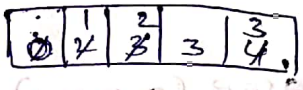
cummulative frequency.



for(i=1; i<=max; i++)
count[i] += count[i-1].

output array:-
output[count[a[i]]-1] = a[i].





Algorithm:-

countingsort(a,n);

Step-1:- Declare output[n], max, i;

Step-2:- Set Max = a[0].

~~Step-3:-~~ ~~if~~

Step-3:- for(i=0; i<n; i++) // Finding max element:-
① if(a[i]>max)
ⓐ set max=a[i].

Step-4:- Declare count[max+1].

Step-5:- for(i=0; i<=max; i++)
ⓘ set count[i]=0.

Step-6:- for(i=0; i<n; i++) // counting array
ⓘ set count[a[i]]++

Step-7:- for(i=1; i<=max; i++) // cummulative frequency
ⓘ set count[i]+=count[i-1]

Step-8:- for(i=n-1; i>=0; i--) // Mapping the elements.
ⓘ output[count[a[i]]-1] = a[i]
ⓘⓘ count[a[i]]--;

Step-9:- for(i=0; i<n; i++)
ⓘ Display "output[i]"

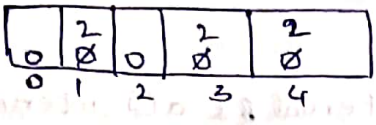
Step-10:- EXIT.

Example :-

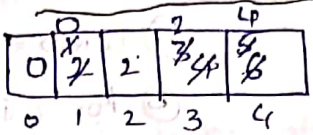
4	1	3	1	4	3

Max element = 4

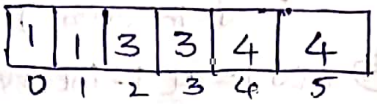
count [max+1] = 5.



cummulative frequency.



output array :-



⑥ Radix sorting :-

		one's	Ten's	Hundred's
437	437	123	123	123
689	689	483	437	145
123 →	123 →	645 →	645 →	437
483	483	145	145	483
645	645	437	483	645
145	145	689	689	689

Algorithm :-

~~step-1 :- for (place=1;~~ ; RadixSort(a,n) → Module.

step-1 :- set max = getmax (a, n)

step-2 :- for (place=1 ; max/place > 0 ; place = place x 10)

① countingsort (a, n, place);

→ change a[i] as (a[i] / place) % 10.

step-3 :- EXIT.

→ The limitation of counting sort is overcomed by Radix sorting [Sorting for large numbers].

Note :-

→ In step-3 of counting sort, Don't display the elements & store them [a[i] = output[i]] in case of radix sort.

⑦ Shell sorting Algorithm:-

shellsort (a, n).

step-1 :- Declare interval, i, j, temp.

step-2 :- for (interval = $\frac{n}{2}$; interval > 0; interval /= 2).

(i) for (i = interval; i < n; i++)

(a) temp = a[i]

(b) for (j = i; j >= interval && a[j - interval] > temp
j = j - interval);

(A) a[j] = a[j - interval].

(c) a[j] = temp;

step-3 :- EXIT.

* Hashing :-

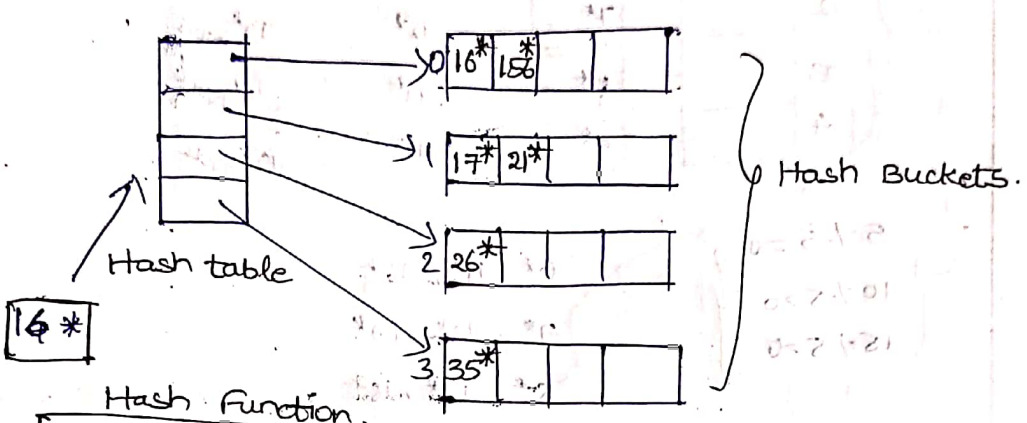
→ Hashing is used to store, and retrieve the data quickly.

→ It has Best time complexity.

→ Before Hashing, we have Direct Addressing [by using arrays] concept.

→ In Direct Addressing key value is Index value.

Hash Table & Hash Function.



K is key value.

M is Hash values.

16* ; h(K) = 16 % 4 = 0

156* ; h(K) = 156 % 4 = 0

17* ; h(K) = 17 % 4 = 1

→ To overcome collisions, we have two Hashings.

① open Hashing.

② closed Hashing.

① open Hashing :-

→ Under open Hashing, we have a concept separate chaining.

→ By using single linked lists we separates the chain.

② closed Hashing :-

→ Also known as open Addressing.

→ We have 3 Technis under this.

(i) Linear probing

(ii) Quadratic probing.

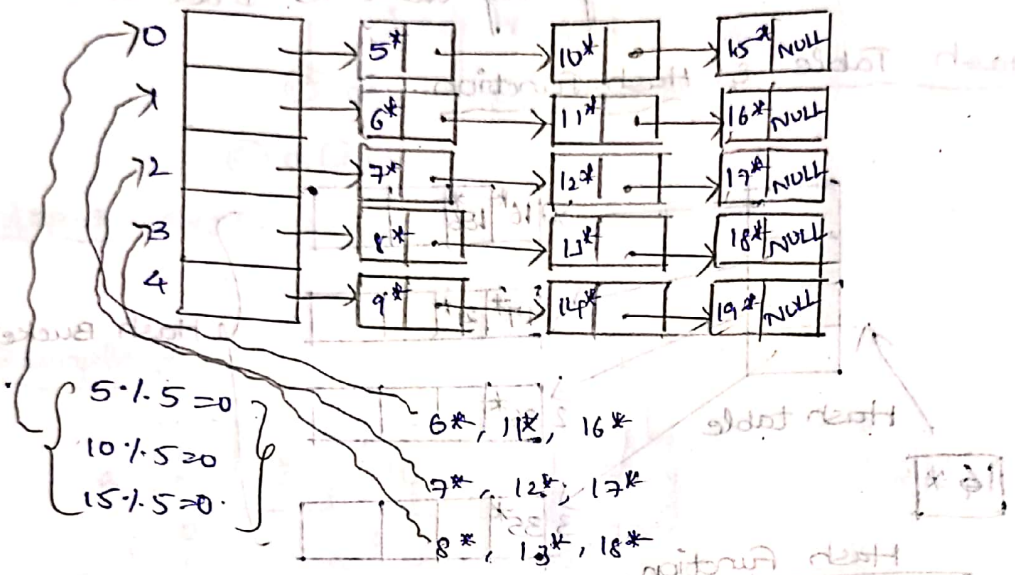
(iii) Double Hashing.

④ Open Hashing :-

→ Set of Records are going to store at same entry, then collision occurs.

→ To resolve collisions, "Open Hashing" is one of the solution.

Example :-



→ We use single Linked List concept to overcome collisions in open Hashing.

Structure :-

SIZE 5.

```
struct node
{
  int data;
  struct node *next;
}*temp, *p;
struct node *a[SIZE];

void Initialize( ); // Initialize array elements.
```

Algorithm

step-1 :- Declare i.

step-2 :- for(i=0; i<SIZE; i++)
  ① Set a[i]=NULL

step-3 :- EXIT.

* Algorithm for Insertion:-

Void insert (int value).

step-1 :- Declare hkey

step-2 :- create a new node ie temp

step-3 :- set temp->data = value, temp->next = NULL.

step-4 :- compute hkey = value % SIZE.

step-5 :- if (a[hkey] == NULL)

(i) set a[hkey] = temp.

step-6 :- else

(i) set P = a[hkey]

(ii) while (P->next != NULL)

(A) set P = P->next

(iii) set P->next = temp

step-7 :- EXIT.

* Algorithm for Deletion:-

void del (int value)

step-1 :- Declare hkey

step-2 :- compute hkey = value % SIZE;

step-3 :- set P = a[hkey].

step-4 :- if (P->next == NULL)

(i) if (P->data = value)

(a) set temp = p.

(b) P = NULL

(c) Free (temp)

step-5 :- else

(i) if (P->data == value)

(a) set temp = P

(b) set a[hkey] = p->next, temp->next = NULL

(c) Free (temp)

(d) EXIT.

(ii) while (P->next != NULL)

(a) if (P->next->data == value)

(A) set temp = p->next

(B) set p->next = temp->next

(C) set temp->next = NULL

(D) Free (temp)

step-6 :- EXIT.

* Algorithm for searching:-

voidsearch (int value)

step-1 :- Declare hkey.

step-2 :- hkey=value % SIZE.

step-3 :- set P=a[hkey]

step-4 :- while (P!=NULL)

(i) if (P->data==value)

(a) return 1

(ii) set P=P->next.

step-5 :- return 0.

step-6 :- EXIT.

* closed Hashing [open Addressing] :-

→ There are three types of technics in closed Hashing.

① Linear probing.

② Quadratic probing

③ Double Hashing.

① Linear probing :-

* Algorithm for Linear probing Insertion :-

#define SIZE 5 ;

int a[SIZE];

void initialize ();

step-1 :- for (i=0; i<SIZE; i++) // Initialization.

(i) set a[i]=-1

Algorithm for Insertion :-

insert (value); // called function.

void insert (int value); // calling function

step-1 :- Declare i, hkey;

step-2 :- compute hkey=value % SIZE.

step-3:- for (i=0; i<SIZE; i++)

(i) if (a[(hkey+i)%.SIZE] == -1)

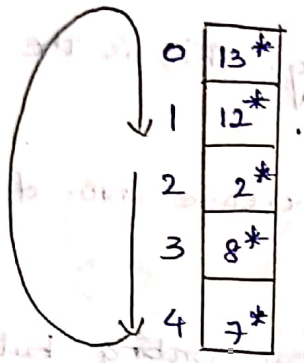
(a) set a[(hkey+i)%SIZE] = value.

(b) break;

step-4:- if (i==SIZE)

(i) Display "No Free slot".

step-5:- EXIT.



2* a[2] is free

8* a[2] is not free, so a[3]

7* a[2], a[3] are not free, so a[4]

13* a[2], a[3], a[4] are not free, so a[0]

12* a[2], a[3], a[4], a[0] are not free.

* Algorithm for deletion:- Deletion:-

step-1:- Declare. i, hkey.

step-2:- Compute hkey = value % SIZE

step-3:- for(i=0; i<SIZE; i++)

(i) if a[(hkey+i)%.SIZE] == value)

(a) Set a[(hkey+i)%.SIZE] = -1

(b) break.

step-4:- if (i==SIZE)

(i) Display "Element to be deleted not found."

(or)

No Input Record found.

step-5:- EXIT

* Algorithm for searching:-

step-1:- Declare i, hkey, flag.

step-2:- compute hkey = value % SIZE.

step-3:- for (i=0; i<SIZE; i++)

(i) if a[(hkey+i)%.SIZE] == value)

(a) set flag = 1

(b) break.

step-4:- if (flag == 1)

(i) Display "Element found"

step-5:- EXIT.

② Quadratic probing :-

0	
1	7*
2	2*
3	8*
4	13*

2*; a[2] is free
8*; a[3] is free
7*; a[2] is not free ; a[1] is free.
13*; a[4] is free.
12*;

→ Here one free slot is skipping. This is the drawback of Quadratic probing.
→ To overcome this we need to increase NO. of iterations.
→ The Insertion is same as Linear probing but replace 'i' by 'ixi'.
→ But Deletion & searching same as Linear probing no need to replace anything.

③ Double Hashing :-

* Algorithm for Insertion :-

Insertion(value);
step-1 :- Declare i, h1key, h2key;
step-2 :- compute h1key = value %. SIZE
step-3 :- compute h2key = PRIME - value %. PRIME.
step-4 :- for (i=0; i<n; i++)
  ⓐ if (a[(h1key + i x h2key) %. SIZE] == -1)
    ⓐ set a[(h1key + i x h2key) %. SIZE] = value
    ⓑ break;
step-5 :- EXIT.

* Note :-
→ To get distinct values take the size of Hashtable as prime numbers.
→ Hashing Applications :- 'DBMS', 'Password storage', 'Data compression', 'cryptography', 'Image processing'.

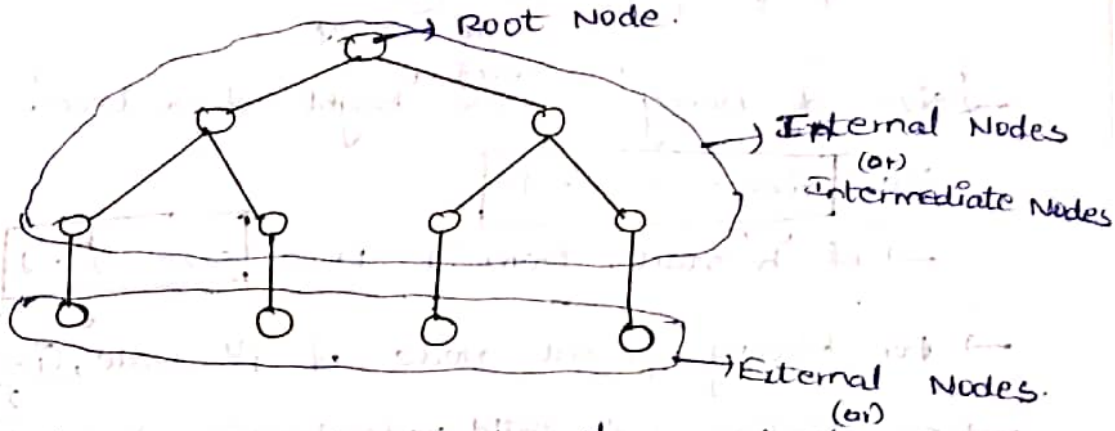
25-04-2023

UNIT-IV · TREES

* Tree :-

→ A 'Tree' is a collection of nodes and there is a one distinct node is called 'Root Node.'

→ A 'Root Node' having no parent Nodes.



Internal Nodes:- Having child Nodes

External Nodes:- Having no child Nodes.

→ If, In a given tree any Node having two child Nodes then it is called 'Binary Tree.'

→ If a tree has any Node containing more than two child nodes then it is called 'M-way Tree.'

* Depth of a Node :-

→ No. of 'Ancestors' is called Depth of a Node.

Ancestors :- Parent (or) grand parent Nodes.

Successors / Descendents :- child (or) grand child Nodes.

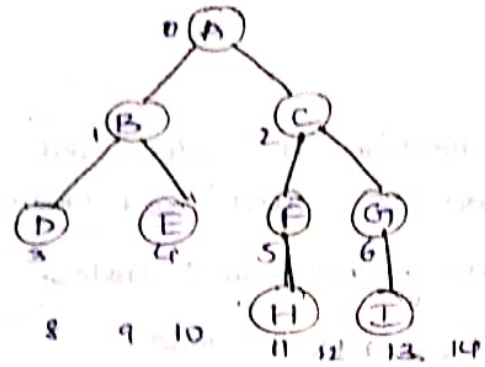
* Height of a Tree :-

→ 'Maximum of depth of a Node' is called the 'Height of a Tree.'

* Sub-Tree :-

→ The 'node' and its Descendents' is known as 'Sub-Tree.'

* Representation of a Binary Tree using Arrays:



→ 'Size of Array' is using the 'height of a tree'.

i.e., $\boxed{\text{size} = 2^{h+1} - 1}$

→ If 'h' starts from '1' then $\boxed{\text{size} = 2^{h} - 1}$

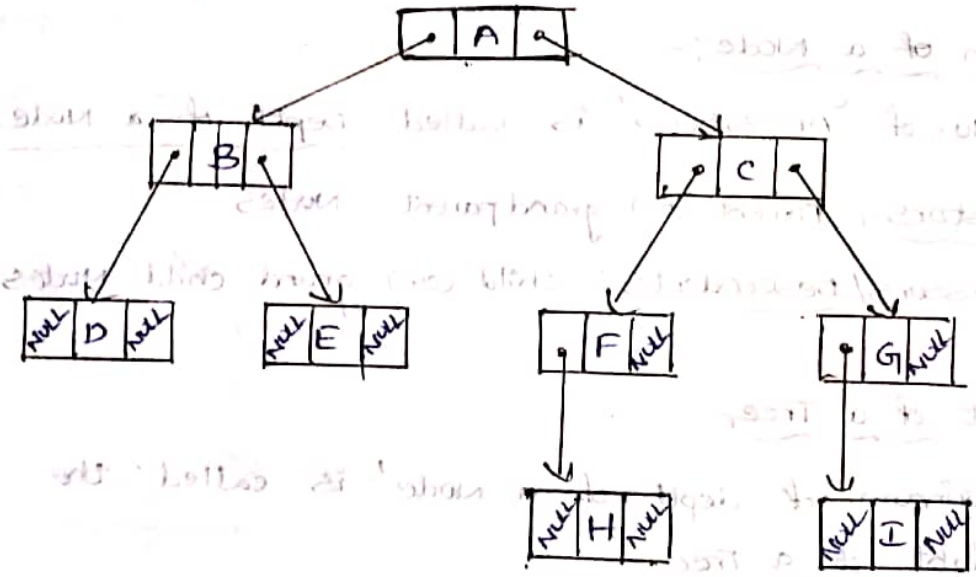
→ For finding parent Node of i^{th} Node, $\frac{(i-1)}{2}$.

→ For finding Left child node location, '2i+1'

→ For finding Right child node location, '2i+2'.

A	B	C	D	E	F	G	-1	-1	-1	-1	H	-1	I	-1
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14

* Representation of a Binary Tree using Linked List:



Full Binary Tree:-

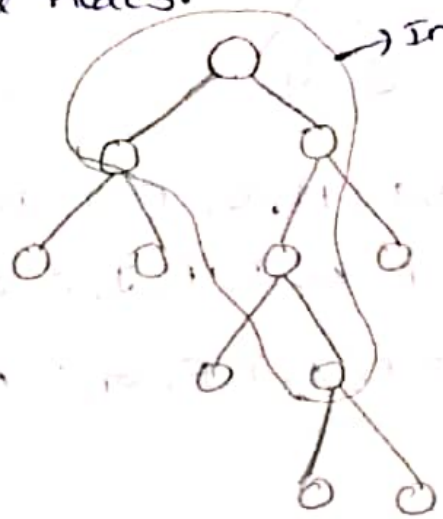
→ A Node containing either '2' or '0' Nodes in the entire tree then it is called 'Full Binary Node'.

Note: For Full Binary tree:

→ If there are 'l' exteral nodes then there are 'l-1' Internal Nodes.

→ If there are 'i' internal nodes then there are 'i+1' external nodes.

Ex:



Internal = i = 5

External = i+1 = 6.

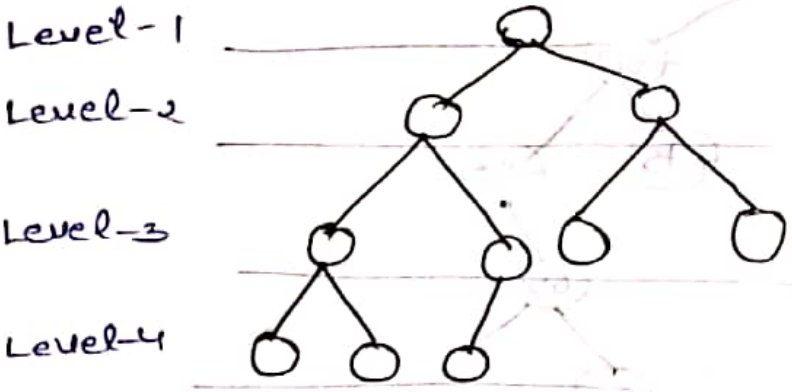
Or)

External = 6 = l

Internal = l-1 = 5

Total = 2(l)-1 = 12-1 = 11

*complete Binary Tree:



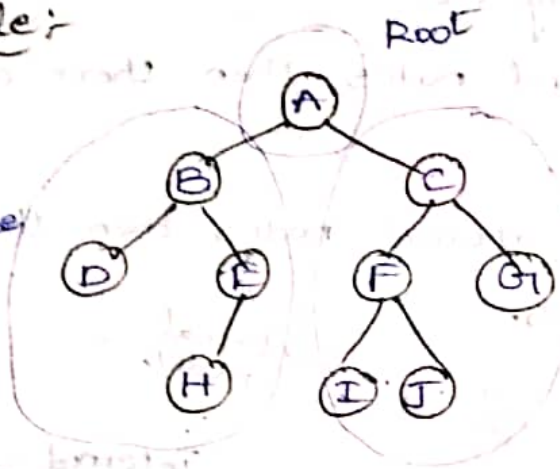
→ Leaf Level will completes by left to right manner.

*Tree Traversals:

→ There are three types.

1. Level order traversal.
2. Depth first traversal.
   - (i) Pre-order traversal [Root, Left subtree, Right stree
   - (ii) In-order traversal. [L.S.T, Root, R.S.T]
   - (iii) Post-order traversal. [L.S.T, R.S.T, Root].

* Example :-

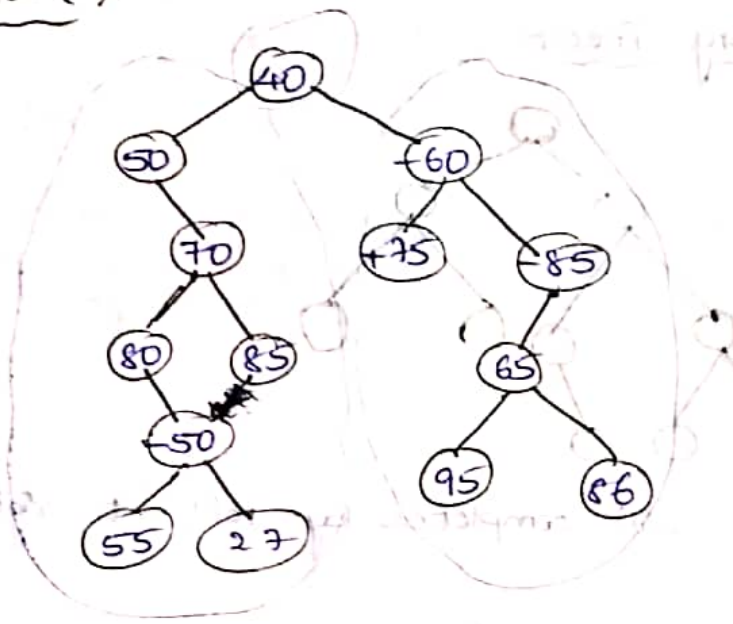


Pre-order :- A, B, D, E, H, C, F, I, J, G.

In-order :- D, B, H, E, A, I, F, J, C, G.

Post-order :- D, H, E, B, I, J, F, G, C, A.

* Example-2 :-



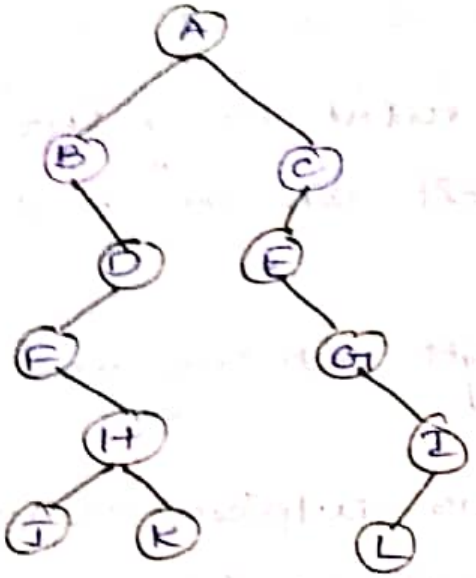
Pre-order :- 40, 50, 70, 80, ~~85~~, -50, 55, 27, 85, -60, 75, ~~+~~

~~In-order :-~~ 65, 95, 86.

Post-order :- ~~55~~ 27, -50, 80, 85, 70, 50, 75, ~~+~~95, 86, 65, -85, -60, 40.

In-order :- 50, 80, ~~70~~ 85, 55, -50, 27, 70, 85, 40, 75, -60, 95, 65, 86, -85.

* Example-3 :-

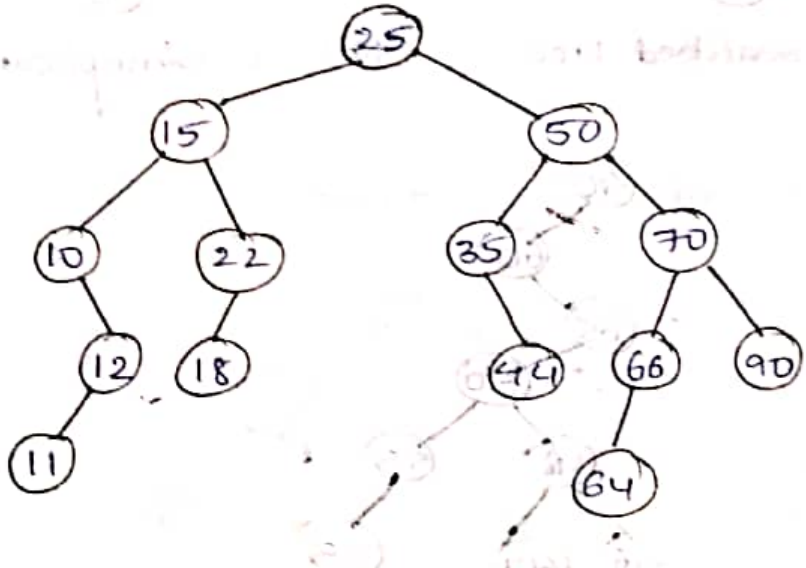


Pre-order :- A, B, D, F, H, J, K, C, E, G, I, L.

In-order :- B, D, J, H, K, D, A, E, G, L, I, C.

Post-order :- J, K, H, F, D, B, L, I, G, E, C, A.

* Example-4 :-



Pre-order :- 25, 15, 10, 12, 11, 22, 18, 50, 35, 44, 70, 66, 64, 90.

In-order :- 10, 11, 12, 15, 18, 22, 25, 35, 44, 50, 64, 66, 70, 90

Post-order :- 11, 12, 10, 18, 22, 15, 44, 35, 64, 66, 90, 70, 50, 25

* Binary Searched Tree [BST]

Properties:-

(1) It should be a rooted Binary tree.

(2) The values of Left sub tree must be less than root Node value.

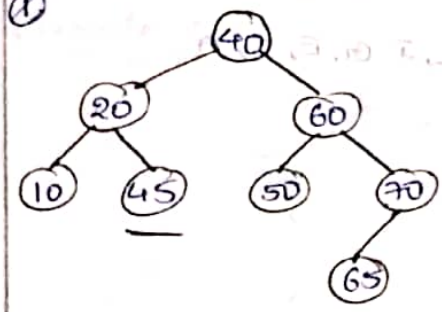
(3) The values of Right sub tree must be greater than root Node value.

(4) It doesnot allows Dulpicate values.

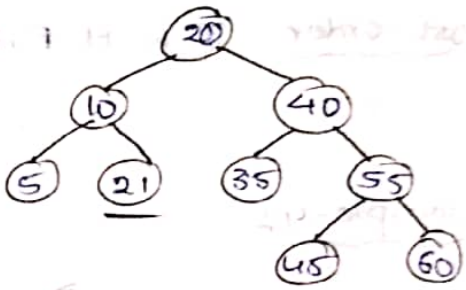
(5) Properties (2), (3) will be followed by every subtree

Examples:-

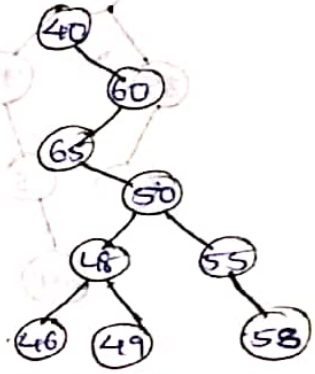
(1)



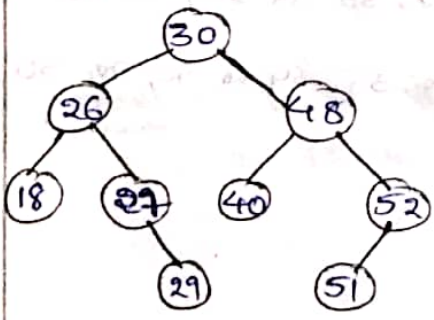
Not a Binary searched tree.



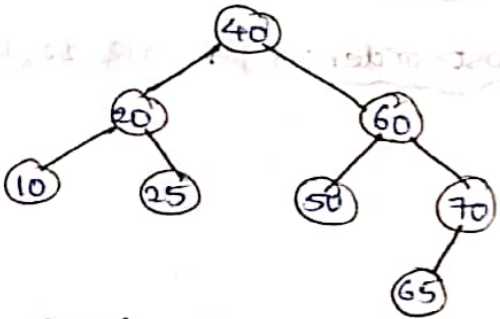
Not a Binary searched tree



It is a Binary searched tree.



It is a Binary Search Tree.



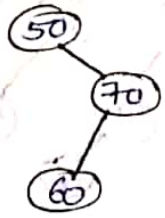
It is a Binary searched tree

* Create a B.S.T using the following sequence of values:

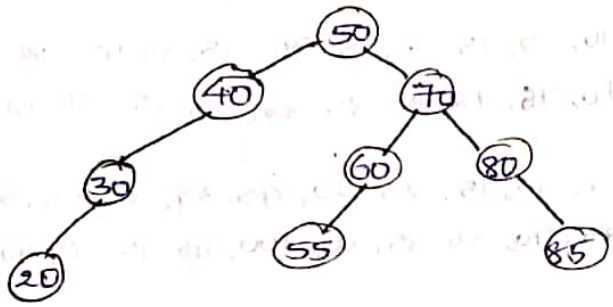
50, 70, 60, 55, 80, 85, 40, 30, 20, 45, 48, 65, 95, 86, 76, 46, 36, 26, 16, 6.

N) → The value inserting at first will be root value.

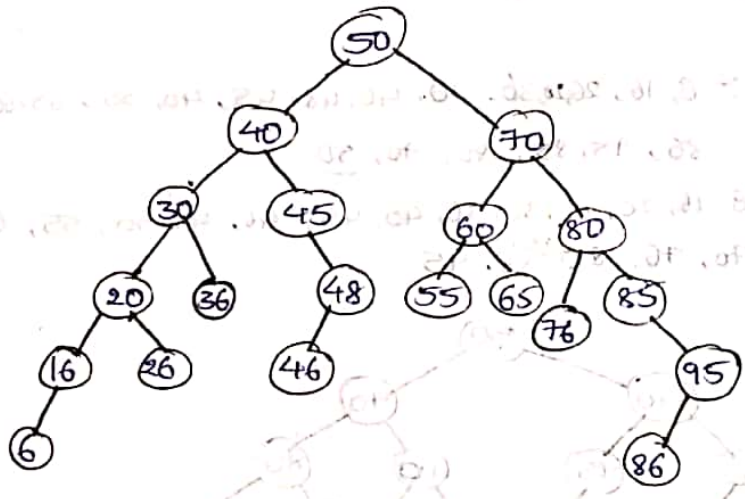
→ compare the inserting value with root value and assign accordingly.



→ If Inserting value is less than root, then traverse Left otherwise traverse Right.



→ Likewise complete the Entire tree step by step.



→ In this way we will create a Binary searched tree satisfying the properties.

Pre-order :- 50, 40, 30, 20, 16, 6, 26, 36, 45, 48, 46, 70, 60, 55, 65, 80, 76, 85, 95, 86.

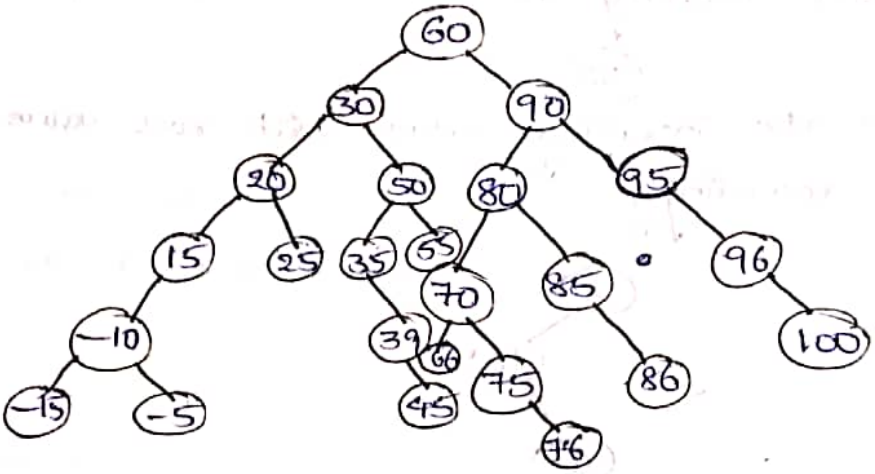
In-order :- 6, 16, 20, 26, 30, 36, 40, 45, 46, 48, 50, 55, 60, 65, 70, 76, 80, 85, 86, 95.

Post-order :- 6, 16, 26, 20, 36, 30, 46, 48, 45, 40, 55, 65, 60, 76, 86, 95, 85, 80, 70, 50.

* Create a B·S·T using the following values:

60, 90, 30, 50, 20, 80, 70, 85, 75, 95, 25, 15, 35, 39, 45, 55, 96, 86, 76, 66, −10, −5, −15, 100.

A)



Pre-order :- 60, 30, 20, 15, −10, −15, −5, 25, 50, 35, 39, 45, 55, 90, 80, 70, 66, 75, 76, 85, 86, 95, 96, 100.

In-order :- −15, −10, −5, 15, 20, 25, 30, 35, 39, 45, 50, 55, 60, 66, 70, 75, 76, 80, 85, 86, 90, 95, 96, 100.

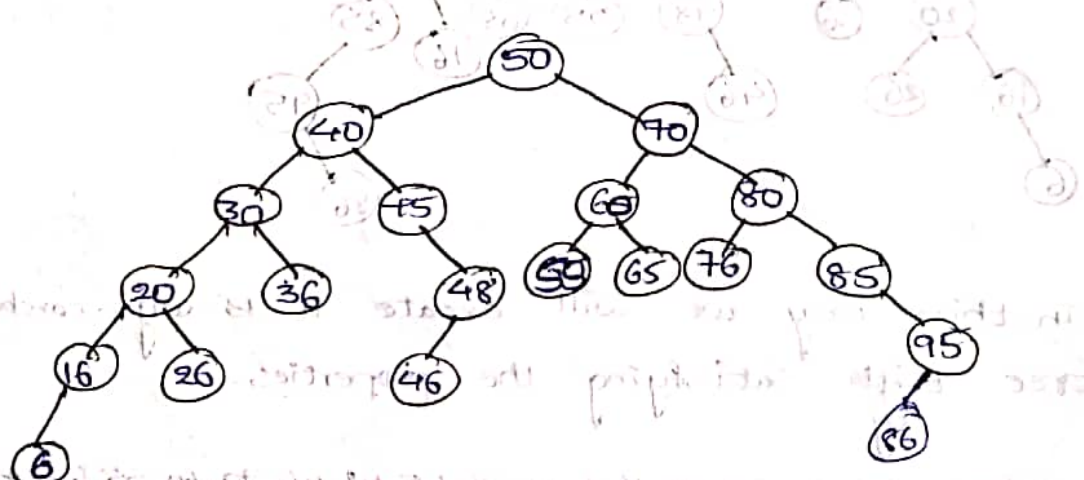
Post-order :- −15, −5, −10, 15, 25, 20, 45, 39, 35, 55, 50, 30, 66, 76, 75, 70, 86, 85, 80, 100, 96, 95, 90, 60.

* Create a B·S·T using its In-order & post-order traversals?

Post-order :- 6, 16, 26, 20, 36, 30, 46, 48, 45, 40, 55, 65, 60, 76, 86, 95, 85, 80, 70, 50.

In-order :- 6, 16, 20, 26, 30, 36, 40, 45, 46, 48, 50, 55, 60, 65, 70, 76, 80, 85, 86, 95.

A)



* Structure of a Binary search Tree

```
struct node
{
  struct node *left;
  int data;
  struct node *right;
} * root = NULL, *p, *temp;
```

root = always points to root / center of a tree.
P = to traverse the tree.
temp = to insert / delete a Node.

* Algorithm for creation :-

```
struct node *newNode (int key);
```

step-1 :- create a new node i.e. temp.

step-2 :- set temp → data = key,
set temp → left = NULL,
set temp → right = NULL.

step-3 :- return temp.

* Algorithm for insertion :-

```
struct node * Insert (struct node *root, int key);
```

step-1 :- if (root == NULL).
(i) return newNode (key)

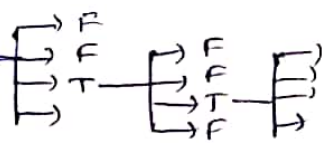
step-2 :- else if (key < root → data).
(ii) root → left = insert (root → left, key).

step-3 :- else if (key > root → data)
(i) root → right = insert (root → right, key).

step-4 :- return root.

Main function :-

root = insert (root, 40).
(or)
root = insert (root, x)



* Algorithm for Traversal :- [In-order Traversal].

void Inorder display (struct node *root);

step-1 :- if (root != NULL)

step-2

(i) Inorder (root → left).

(ii) Display "root → data"

(iii) Inorder (root → right)

step-2 :- EXIT.

* Algorithm for Pre-order Traversal :-

void Preorder (struct node *root);

step-1 ; if (root != NULL)

(i) Display "root → data".

(ii) Preorder (root → left)

(iii) Preorder (root → right)

step-2 :- EXIT.

* Algorithm for Post-order Traversal :-

void Postorder (struct node *root);

step-1 :- if (root != NULL)

(i) Postorder (root → left)

(ii) Postorder (root → right)

(iii) Display "root → data"

step-2 :- EXIT.

* Deletion :-

case-1 :- If deleting node is 'Leaf node' then discard it directly.

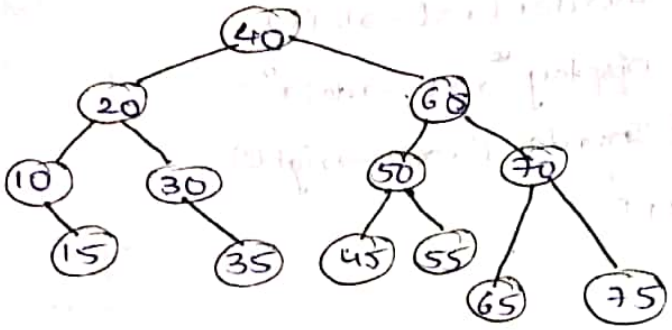
case-2 :- If deleting node contains 'one child node' then replace it with child node [either left or right]

case-3 :- If deleting node contains 'two nodes (or) two sub trees' then substitute with Inorder predecessor or successor. (Maximum value). (or) (min value).

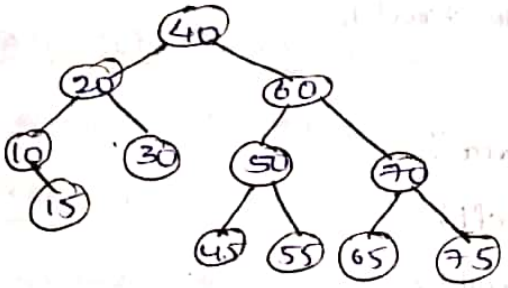
Inorder predecesor :- Maximum value of left sub tree

Inorder successor :- Minimum value of right sub tree

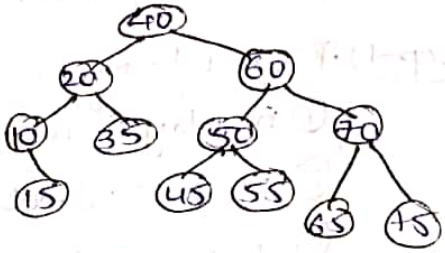
Ex :-



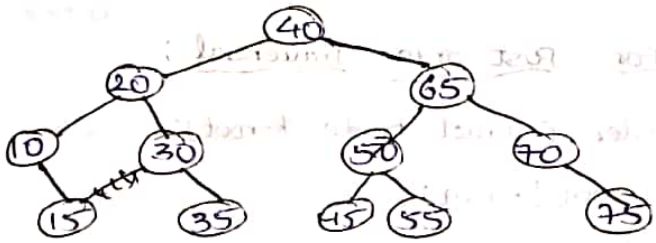
case-1 :- Delete 35



case-2 :- Delete 30



case-3 :- Delete 60



* Algorithm for Deletion :-

struct node * deleteNode (struct node * root, int key);

step-1 :- if (root == NULL)

(i) return NULL

step-2 :- else if (key < root → data)

(i) root → left = deleteNode (root → left, key)

step-3 :- else if (key > root → data)

(i) root → right = deleteNode (root → right, key)

step-4 :- else

(i) if (root → left == NULL)

(a) set temp = root → right

(b) free(root)

(c) return temp.

(ii) else if (root → right == NULL)

(a) set temp = root → left

(b) free (root)

(c) return temp.

(iii) set temp = minValue BST (root → right)

(iv) Set root → data = temp → data.

(v) root → right = deleteNode (root → right, temp → data).

Step-5 :- return root.

* Algorithm for Inorder successor :-

struct node * minValue BST (struct node * root);

step-1 :- set p = root

step-2 :- while (P != NULL && P → left != NULL)

(i) set p = p → left.

step-3 :- return p.

* Algorithm for searching :-

struct node * searchBST (struct node * root, int key);

step-1 :- if (root → data == key OR root == NULL)

(i) return root.

step-2 :- else if (key < root → data)

(i) return searchBST (root → left, key);

step-3 :- else

(i) return searchBST (root → right, key);

* Algorithm for counting total no. of nodes :-

int totalNodesBST (struct node * root)

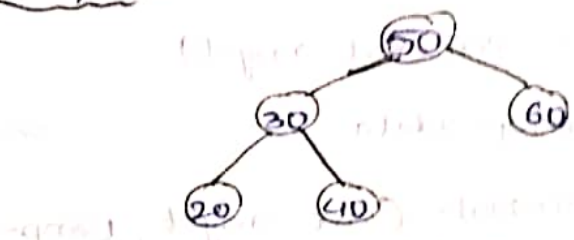
step-1 :- if (root == NULL)

ⓘ return 0.

step-2 :- else

ⓘ return totalNodesBST (root → left) +
totalNodesBST (root → right) + 1.

Example :-



t.N.BST (50) ← 1. False
2. return tNBST (30) + tNBST (60) + 1

1. False | 2. return tNBST (20) + tNBST (40) + 1

1. False | 2. return tNBST (NU + tNBST (Nu + 1

* Algorithm to find internal Nodes in B.S.T :-

int internalnodesBST (struct node * root)

step-1 :- if (root == NULL)

ⓘ return 0

step-2 :- else if (root → left == NULL && root → right == NULL)

ⓘ return 0

step-3 :- else

ⓘ return internalnodesBST (root → left) +
internalnodesBST (root → right) + 1

* Algorithm to find External Nodes:-

int externalBST (struct node * root)

Step-1 :- if (root == NULL)

(i) return 0

Step-2 :- else if (root → left == NULL && root → right == NULL)

(i) return 1

step-3 :- else

(i) return externalBST (root → left) + externalBST (root → right) + 1

* Algorithm for Height of a BST :-

int height BST (struct node *root);

step-1 :- if (root == NULL)

(i) return 0

step-2 :- else

(i) LH = height BST (root → left)

(ii) RH = height BST (root → right)

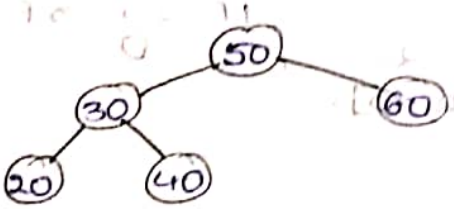
(iii) if (LH > RH)

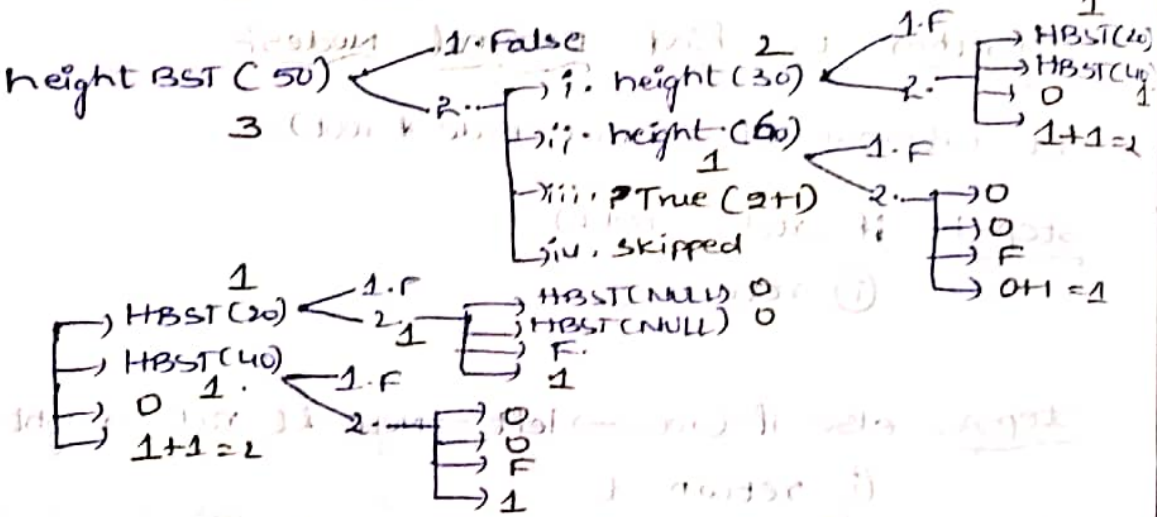
(a) return LH + 1

(iv) else

(a) return RH+1

Example :-





∴ Height of given tree = 3 − 1 = 2. [∵ root node starts from '0'].

Note :-

→ Most of the tree root Node is starts from '0'. so, we need to reduce '1' to get exact height of the Tree [BST].

* Heap Tree :-

→ Heap Tree follows the properties of complete Binary Tree. [Nodes fill from Left to right. by Level by Level]

Types of Heap tree :

① Max Heap tree.

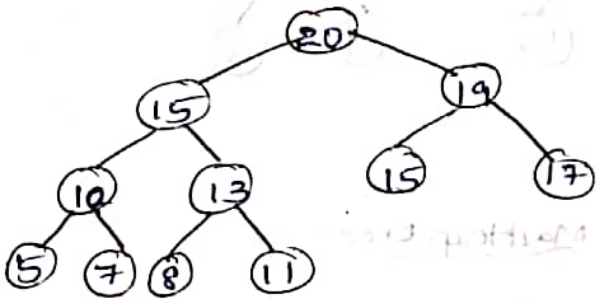
② Min Heap tree.

① Max Heap tree :-

→ The 'root node value' is more than all node values.

→ If we consider any sub tree, the 'parent node value' is must greater than 'child node values'.

Ex :-

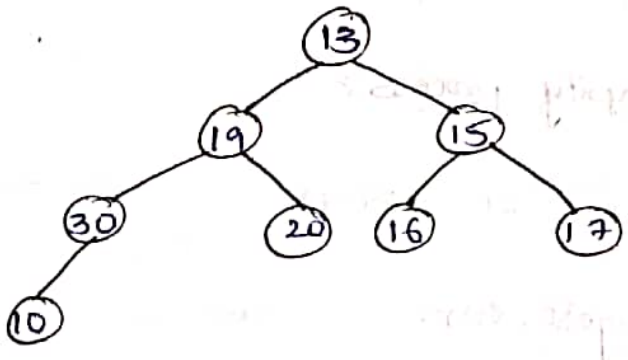


② Min Heap tree :-

→ The root node value is 'smaller' than all node values.

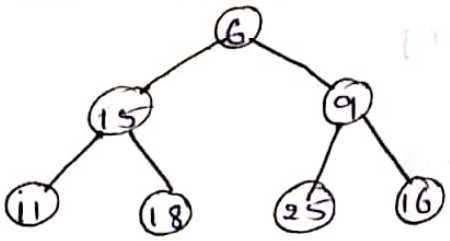
→ If we consider any subtree, the 'parent node value' is must smaller than 'child node values'.

Ex :-



* Ex:- 6, 15, 9, 11, 18, 25, 16.

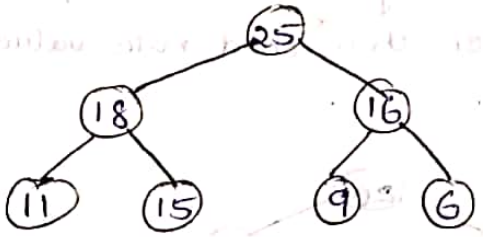
A) Complete Binary Tree :-



Heapify :- heapify(a, n, i).

$i = \frac{n}{2} - 1 = \frac{7}{2} - 1$. & $i >= 0$.

We will build complete max Heap tree by using Heapify procedure:



* Algorithm for MaxHeap tree:-

BuildMaxheap(a, n);

step-1 :- Declare i.

step-2 :- for(i = $\frac{n}{2}$ - 1; i >= 0; i--)

① heapify(a, n, i)

step-3 :- EXIT.

* Algorithm for Heapify process:

heapify(a, n, i);

void heapify(int a[], int n, int i)

step-1 :- Declare largest, l, r.

step-2 :- set largest = i.

step-3 :- compute l = 2xi+1 , r = 2xi+2 .

step-4 :- If (l < n && a[l] > a[largest])

(i) set largest = l ;

step-5 :- if (r < n && a[r] > a[largest])

(i) set largest = r ;

step-6 :- if (largest != i)

(i) swap (&a[i], &a[largest])

(ii) heapify (a, n, largest).

step-7 :- EXIT .

* Algorithm to Build MinHeapTree :-

step-1 :- Declare i.

step-2 :- for(i = $\frac{n}{2}$ - 1; i>=0; i--)

(i) heapify (a, n, i)

step-3 :- EXIT

heapify procedure :-

step-1 :- Declare smallest, l, r.

step-2 :- set smallest = i.

step-3 :- compute l = 2*i+1

(ii) r = 2*i+2

step-4 :- if(l<n && a[l] < a[smallest])

(i) smallest = l.

step-5 :- if(r<n && a[r] < a[smallest])

(i) smallest = r.

step-6 :- if(smallest != i)

(i) swap(&a[i], &a[smallest])

(ii) heapify(a, n, smallest)

step-7 :- EXIT.

* Algorithm for Heapsort :-

step-1 :- Declare i.

step-2 :- for(i = $\frac{n}{2}$ - 1; i>=0; i--)

(i) heapify (a, n, i).

step-3 :- for(i=n-1; i>=0; i--)

(i) swap(&a[0], &a[i])

(ii) heapify(a, i, 0)

step-4 :- EXIT.

* Expression Tree :-

→ It is a special kind of Binary Tree which is used to represent expressions.

→ An Expression is composed of 'variables', 'constants' and 'operators' which are arranged according to rules.

Example :-

→ An 'Expression tree' is a representation of an expression in the form of tree like data structure.

→ Operands are leaves [Leaf Nodes] in the tree.

→ Operators are Internal nodes in the expression tree.

* Example :-

Convert the following Infix expression into Binary expression tree :- a * b/c + e/f * g + K - x * y.

A)

Operator		Precedence & Associativity
^	→	Right to left
*, /	→	Left to right
-, +	→	Left to right.

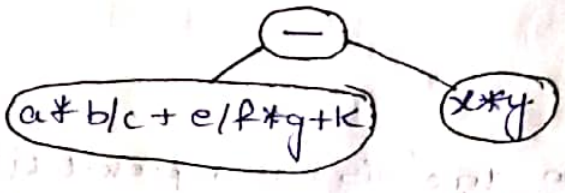
Rules :-

→ Highest preference operators are stored near to the Leaf nodes. Because they are going to be evaluated first.

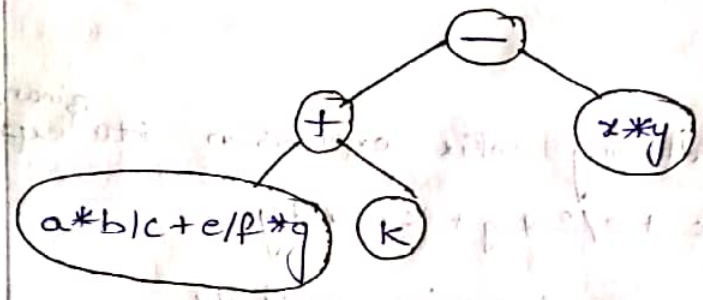
→ The lowest preference operator will be stored in Root Node and it is evaluated at last.

A) Given expression is a * b/c + e/f * g + k - x * y

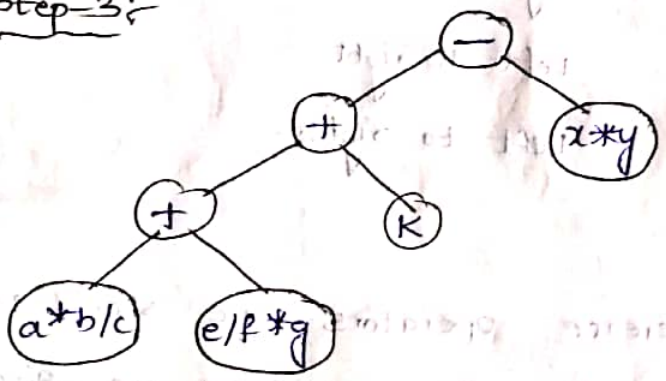
Step-1 :- check out & find the operator which is having least preference. as per associativity and store it in root node.



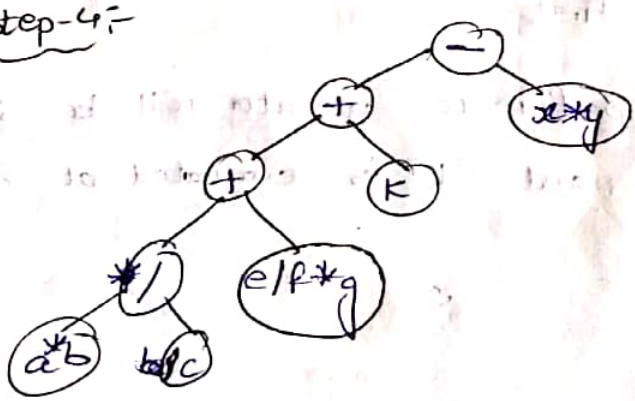
Step-2 :- Repeat the same process [find the operator with minimum precedence]. in either Right subtree (or) left subtree.



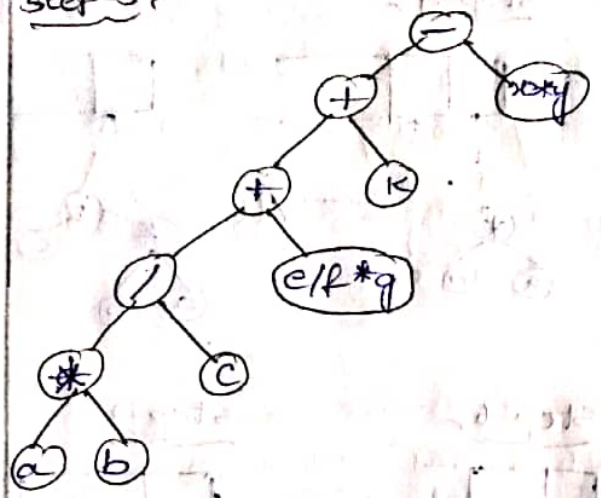
Step-3 :-



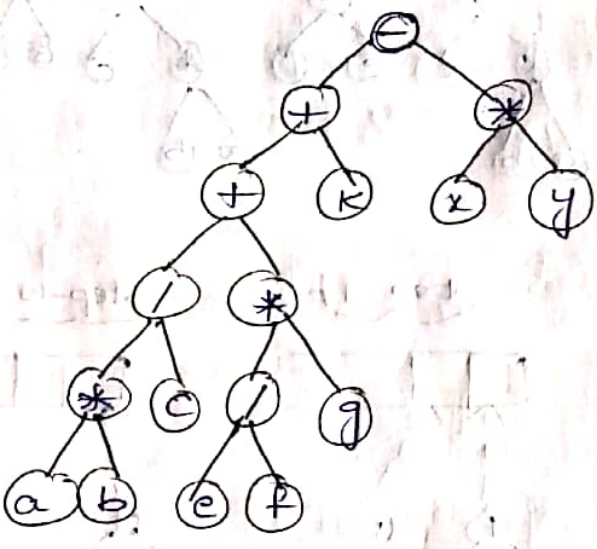
Step-4 :-



step-5:-



step-6:-



This is the required expression Tree.

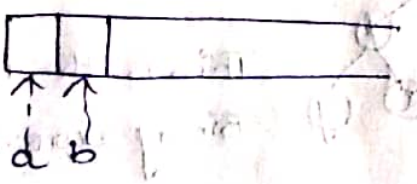
Prefix expression:- -++/*abc*/efgk**xy

Post expression:- ab*c/ef/g*+k+xy*-

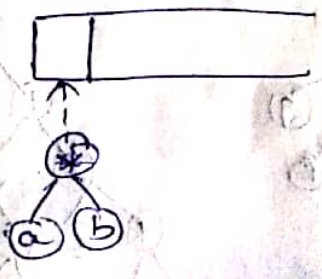
* convert the postfix expression to Binary expression tree?

ab*c/ef/g*+k+xy*-

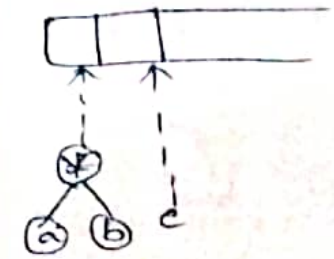
A) step-1:-



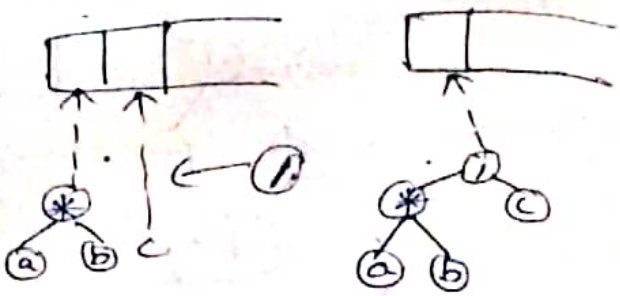
step-2:-



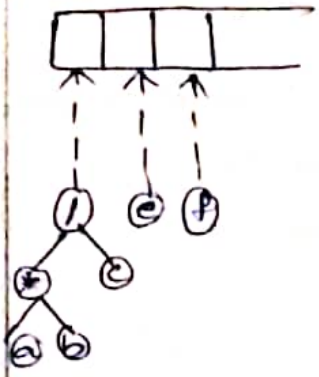
step-3 :-



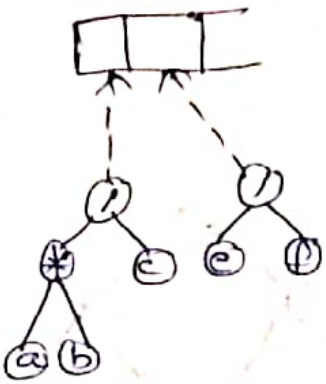
step-4 :-



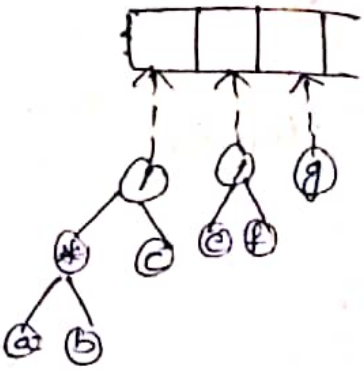
step-5 :-



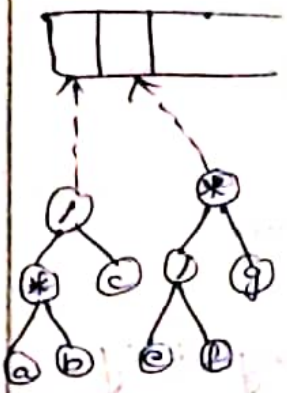
step-6 :-



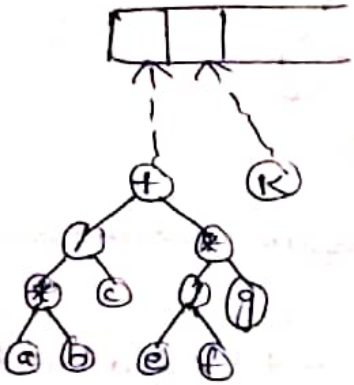
step-7 :-



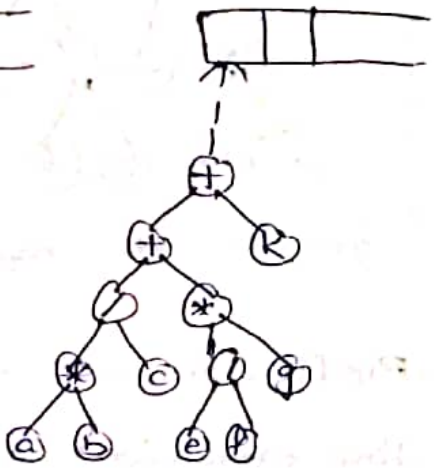
step-8 :-



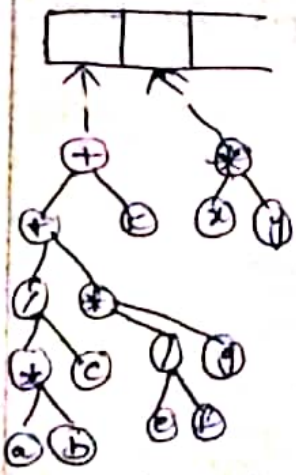
step-9 :-



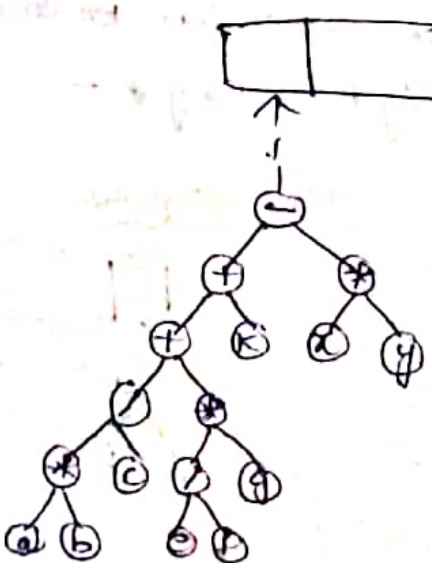
step-10 :-



step-11 :-



step-12 :-

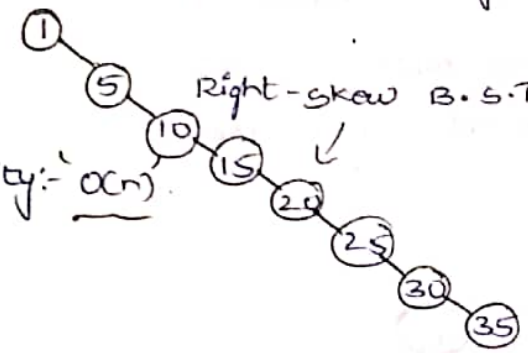


This is the required Binary expression tree

* AVL Tree(s) :- [Self Balanced B.S.T]

→ The main drawback of B.S.T is, when we want to create a B.S.T using sequence of values which are in increasing order / Decreasing order then we will get Linear List. [skew tree].

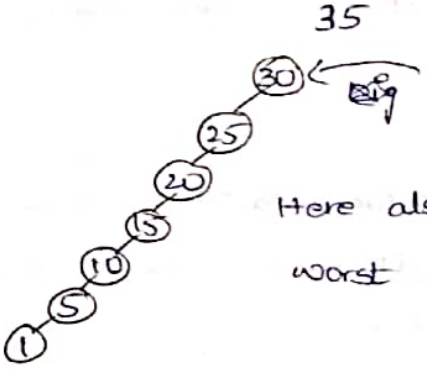
Ex:- 1, 5, 10, 15, 20, 25, 30, 35 [Increasing order].



Right-skew B.S.T

Time complexity:- O(n).

Ex:- 35, 30, 25, 20, 15, 10, 5, 1 [Decreasing order]



Left skew B.S.T

Here also, the time complexity for worst case is 'O(n)'. [For searching]

→ To overcome this Limitation, 'Adelson', 'Velski' and 'Landis' these three persons gave a concept with their name called 'AVL Tree.'

→ It is also known as "Balanced Binary search Tree."

Properties of AVL Tree :-

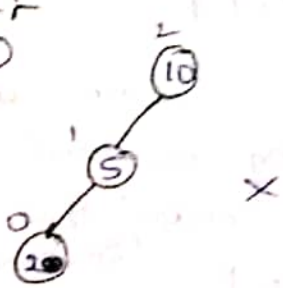
1. It should follow the all properties of 'Binary Tree' [Having atleast two child nodes].
2. It should follow the all properties of 'Binary search Tree'.
3. **Balance factor = Left subTree's Height − Right subTree's height**

→ At any point of time for a given tree, the height of tree will be '0, −1, 1'.

→ The difference b/w heights of Leftsubtree and Rightsubtree should not be more than '1'.

Ex:-

(1)



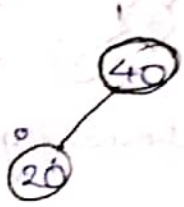
Height of Leftsubtree = 2
Height of Right subtree = 0
∴ Balance factor = 2 - 0 = 2
∴ It is not a AVL Tree.

(2)

Balance factor = 1 - 0
= 1
∴ It is an AVL Tree.

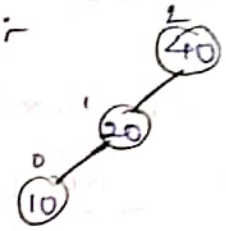
Example:-

step-1:-



Balance factor = 1 - 0 = 1
∴ It is an AVL Tree.

step-2:-



Balance factor = 2 - 0 = 2
∴ It is not an AVL Tree.

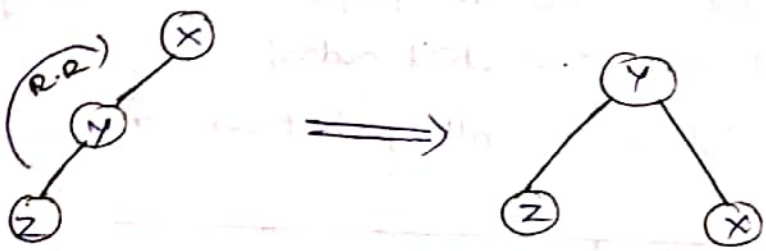
* Insertion:-

Case-1:- "L-L Insertion"

→ 'L-L Insertion' means inserting a node for at 'Left subtree of Left subtree'.

→ Here we will apply 'Right Rotation' [RR]

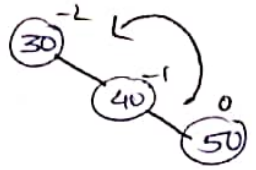
Ex:-



Case-2 :- R·R Insertion"

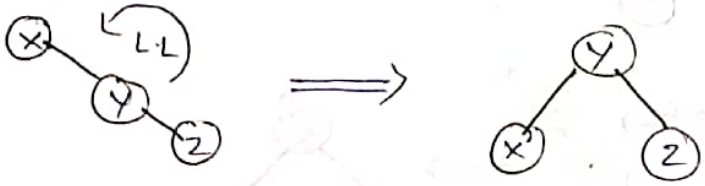
→ Insertion at Right subtree of Right subtree.

Ex :-



Balance factor = -2.
Not AVL Tree.

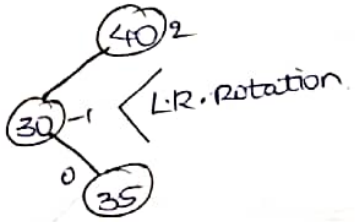
→ So, we'll use L·L [Left Rotation] here.



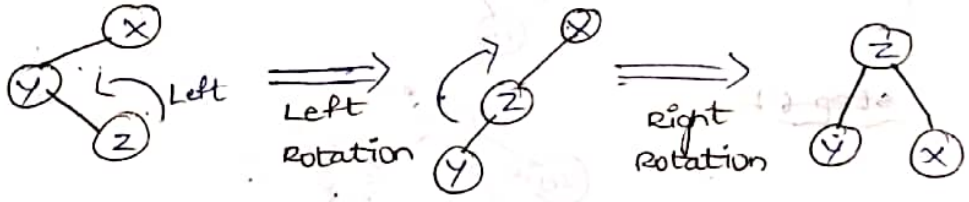
Case-3 :- "R·L Insertion"

→ Insertion at Right subtree of Leftsubtree.

Ex :-



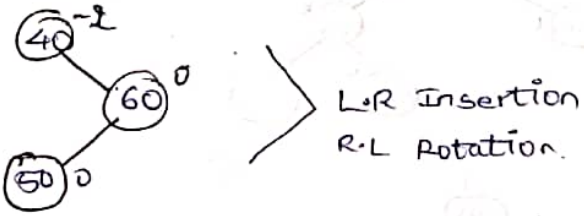
→ So we'll apply L·R Rotation here.



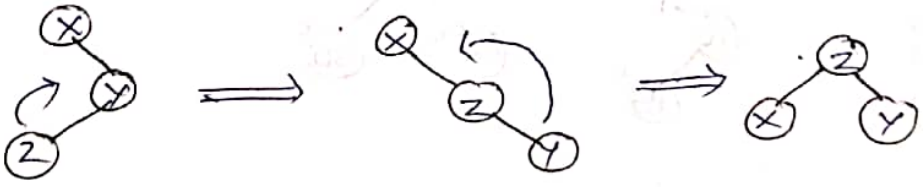
Case-4 :- "L·R Insertion"

→ Insertion at Left subtree of Right subtree.

Ex :-



→ We need to apply Right Rotation first then Left rotation.

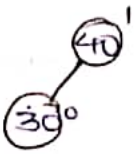


* Create an AVL Tree by using the following values

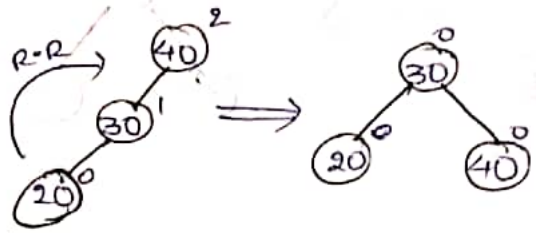
40, 30, 20, 50, 45, 10, 15, 5, 55, 60, 65, 70, 68, 80, 75, 85, 90.

A) step-1 :-

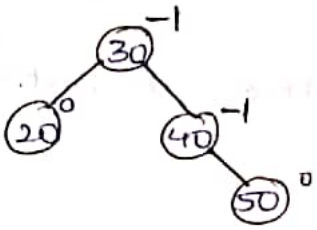
step-2 :-



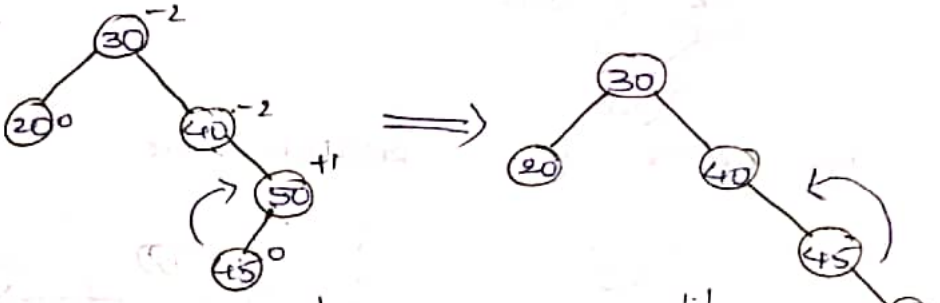
step-3 :-



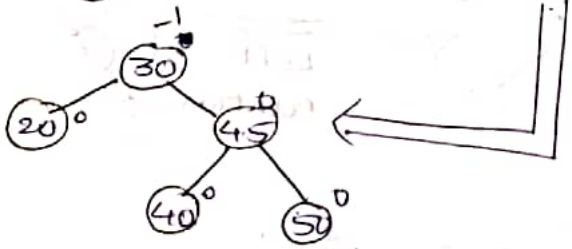
step-4 :-



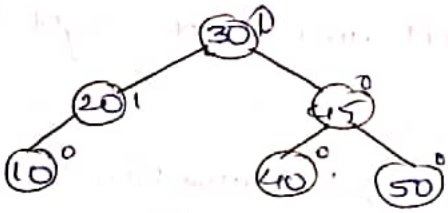
step-5 :-



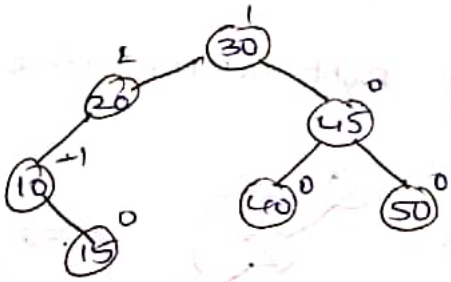
step-6 :-



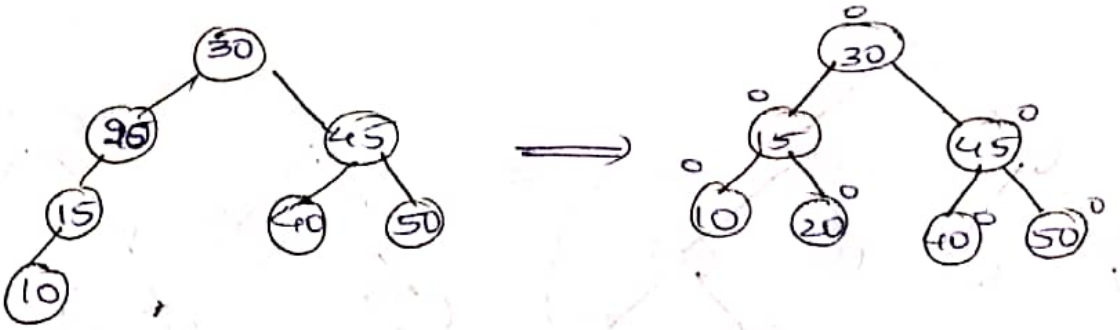
step-7 :-



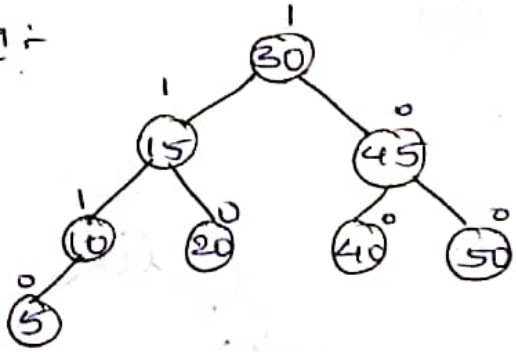
step-8 :-



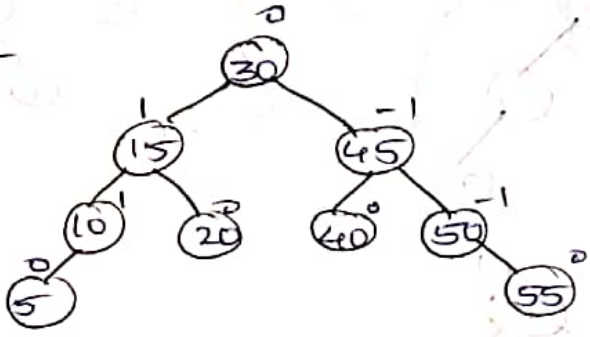
→ Then we need to apply L-R Rotation



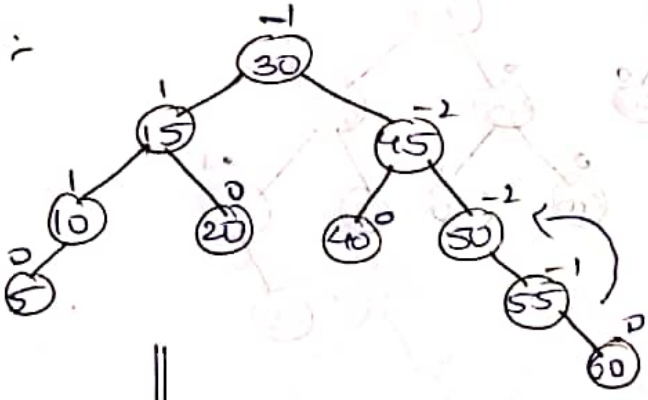
step-9 :-

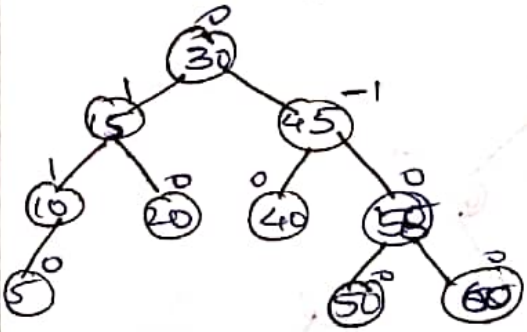


step-10 :-

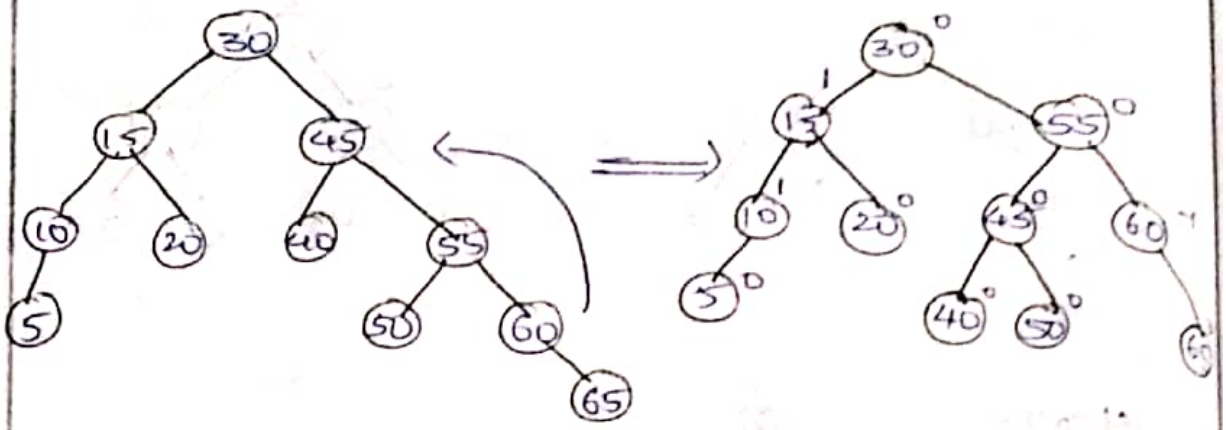


step-11 :-

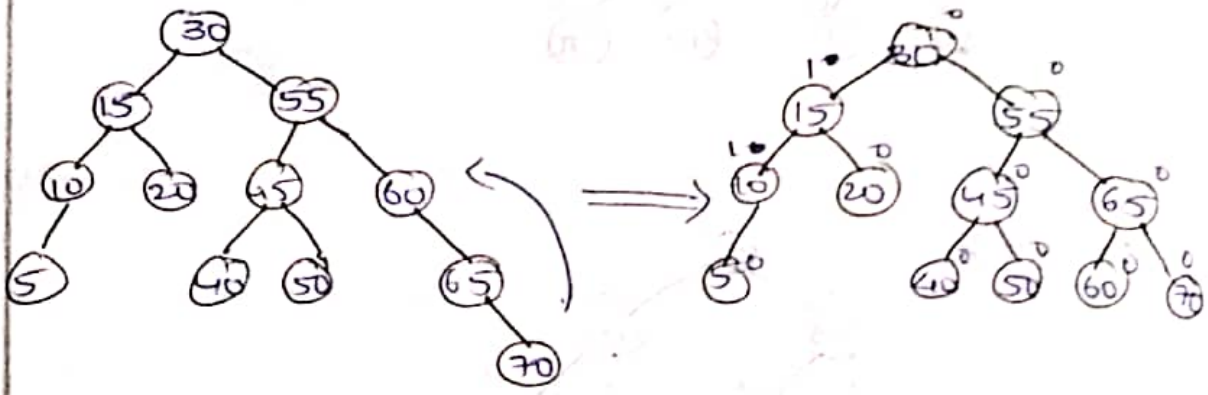




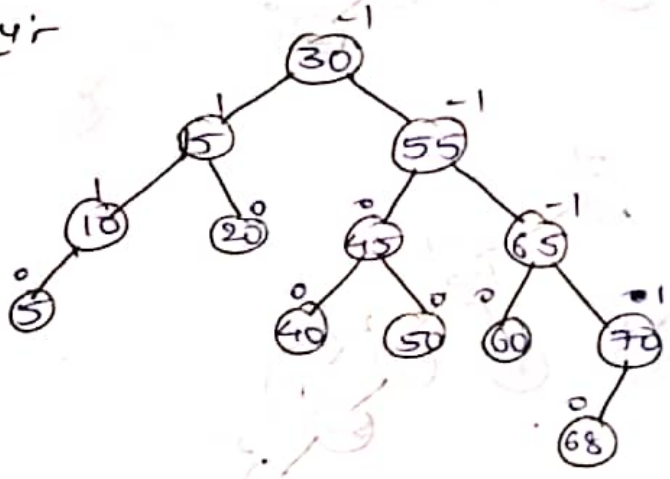
step-12 :-



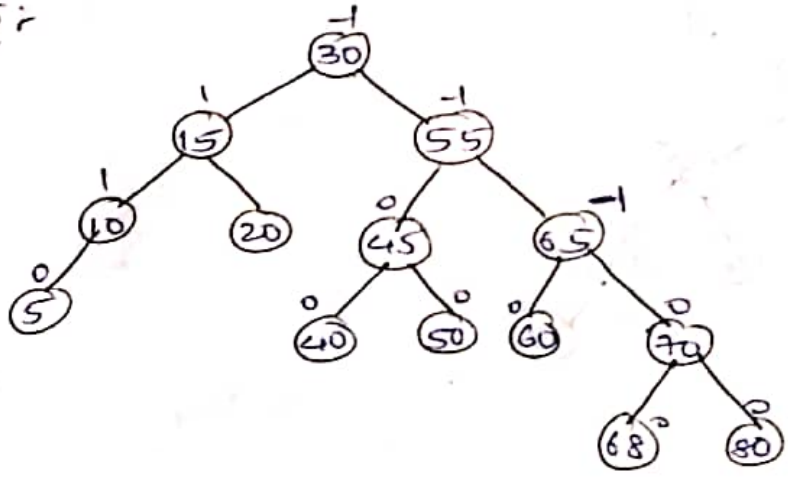
step-13 :-



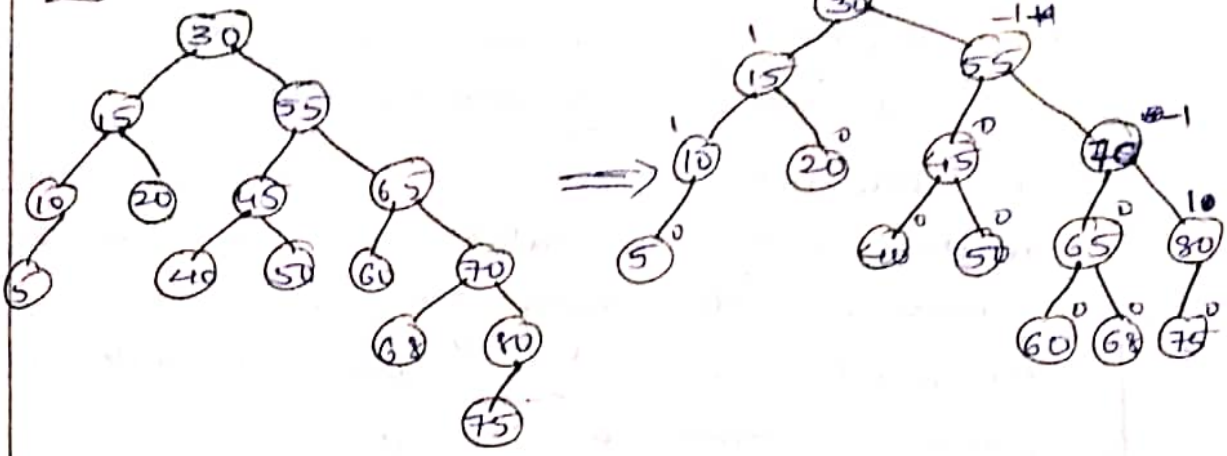
step-14 :-



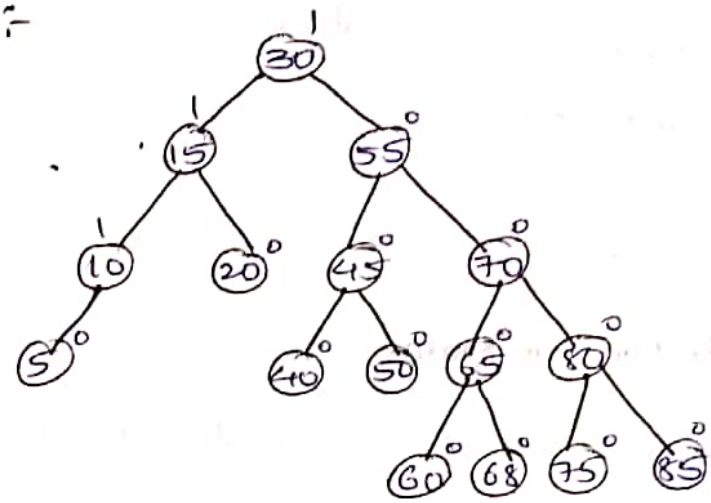
step-15 :-



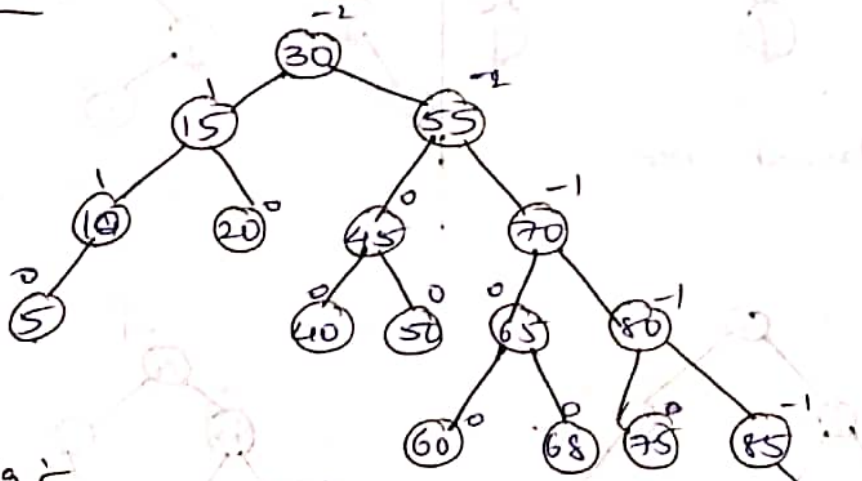
step-16 :-



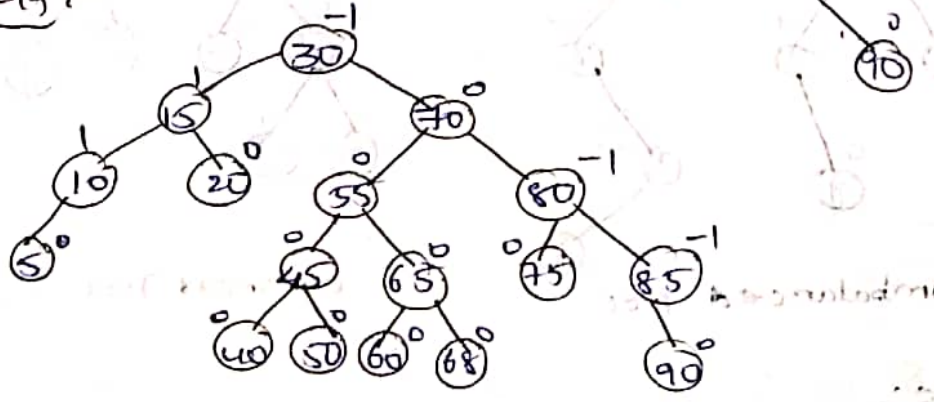
step-17 :-



step-18 :-



Step-19 :-

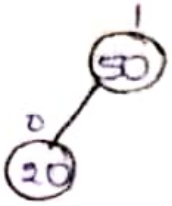


* Create an AVL Tree with the following values by using suitable Rotations

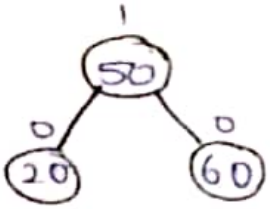
50, 20, 60, 10, 8, 15, 32, 46, 11, 48.

A) step-1 :-

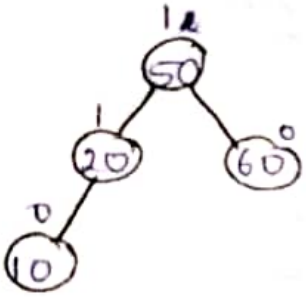
step-2 :-



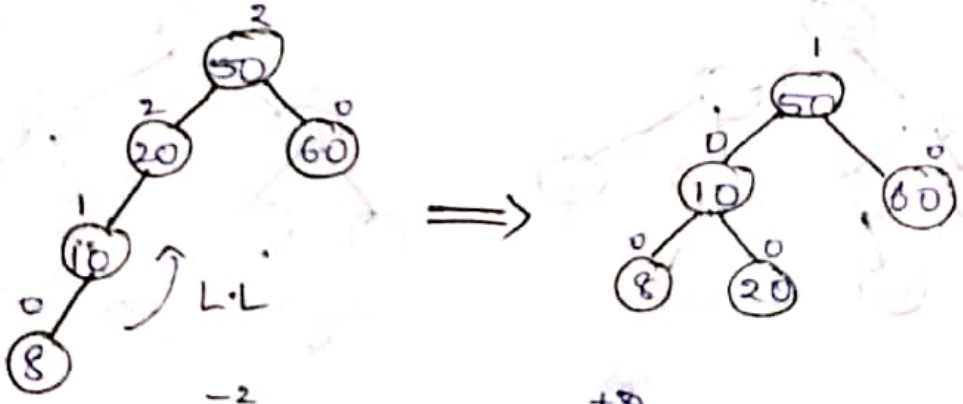
step-3 :-



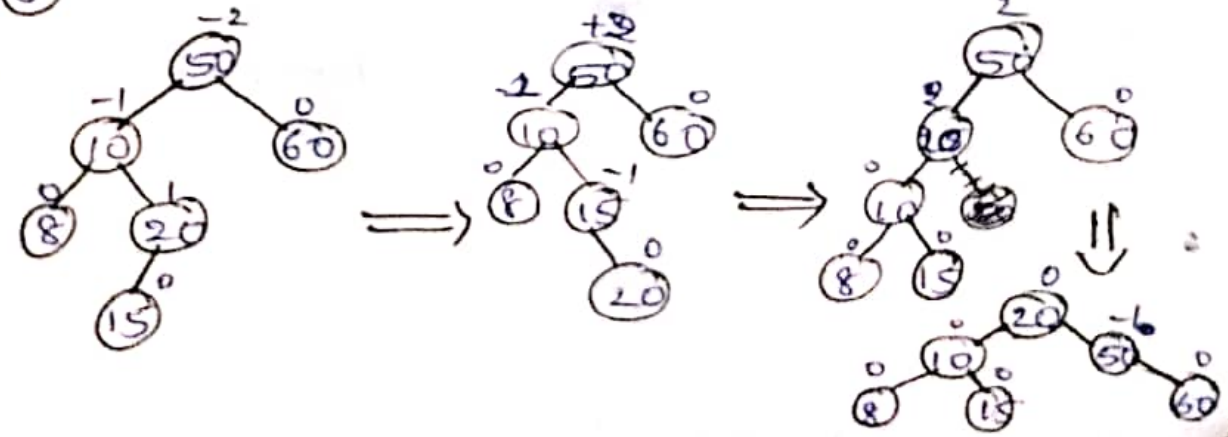
step-4 :-



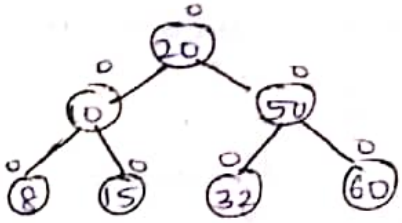
step-5 :-



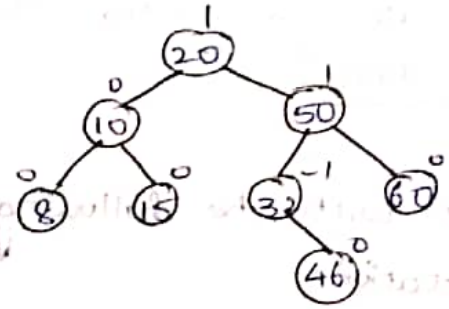
step-6 :-



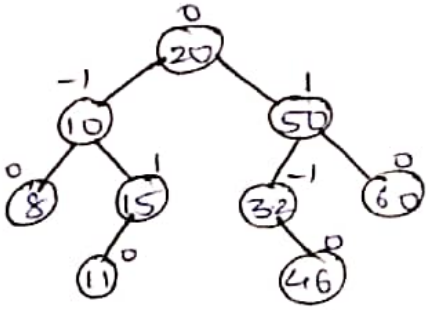
Step-7 :-



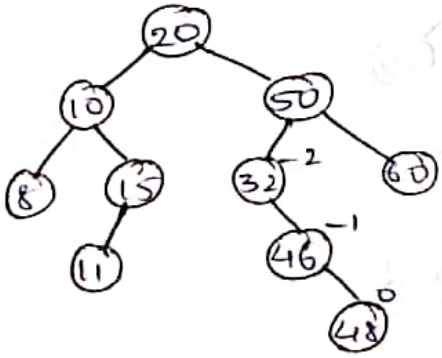
Step-8 :-

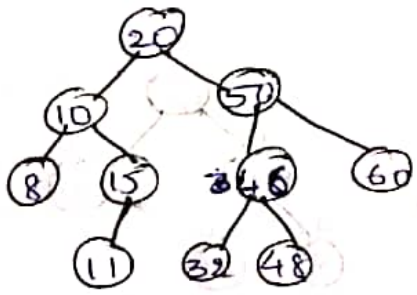


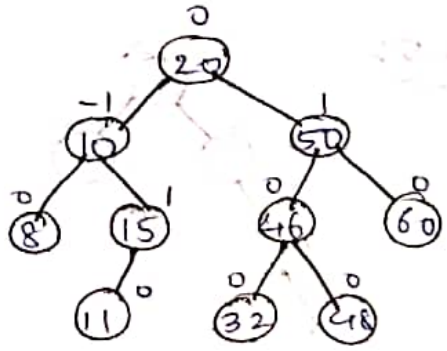
Step-9 :-



step-10 :-



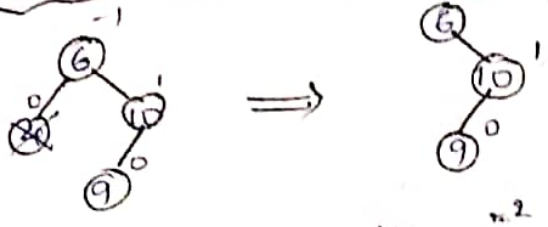




Final Tree. [AVL Tree]

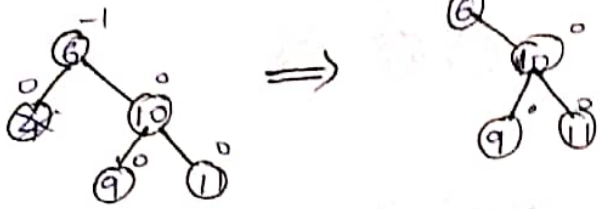
* Delete Operation in AVL Tree:-

(1)



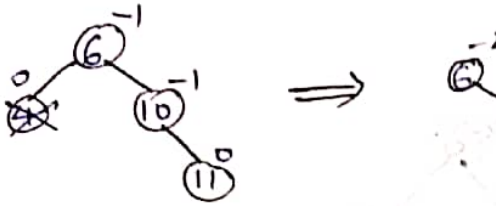
L_{-1} Notation
[R-L rotation]

(2)



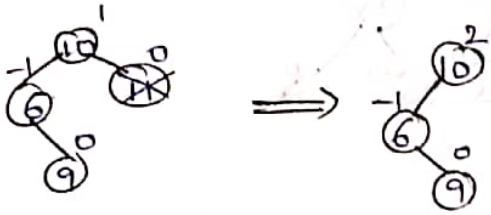
L_0 [R-R].

(3)



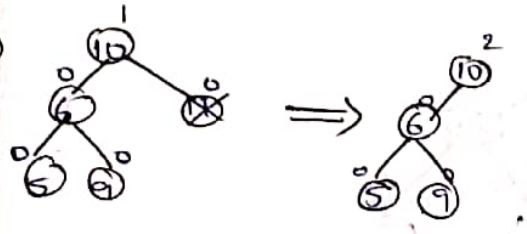
$L_{-1} \rightarrow$ R-R rotation

(4)



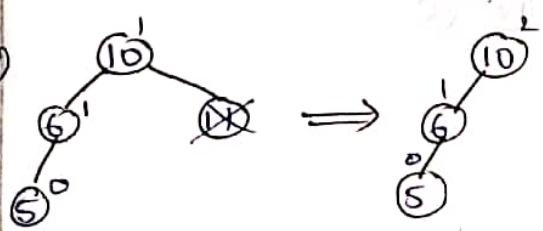
$R_{-1} \rightarrow$ L-R rotation.

(5)



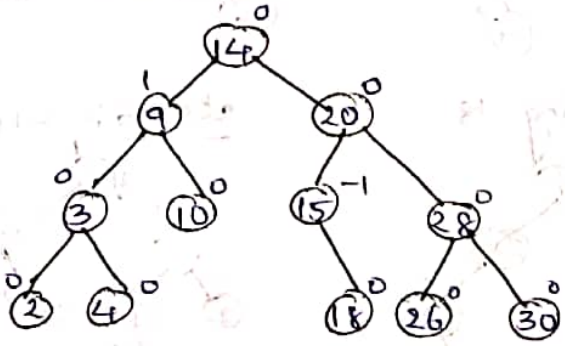
$R_0 \rightarrow$ L-L rotation.

(6)

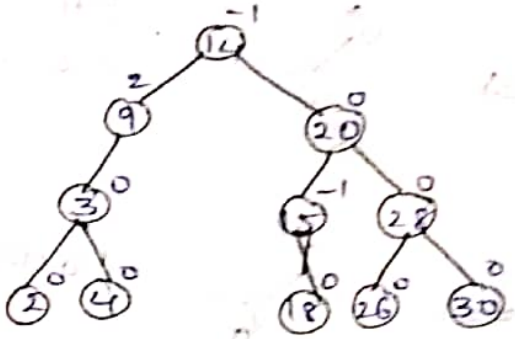


$R_{+1} \rightarrow$ L-L rotation

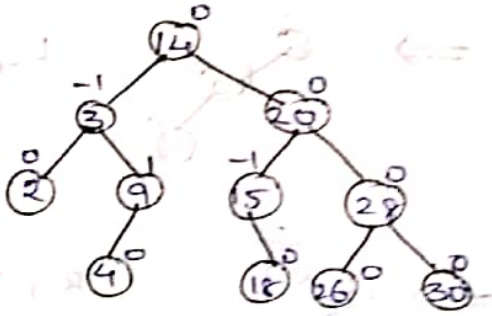
* Delete 10, 30, 15, 18 from the following tree.



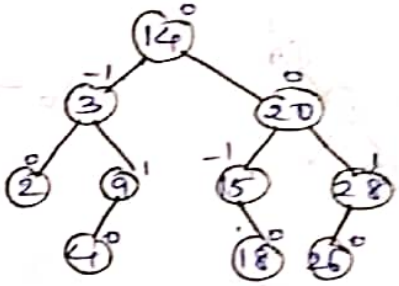
A) Step-1 :- Deleting 10.



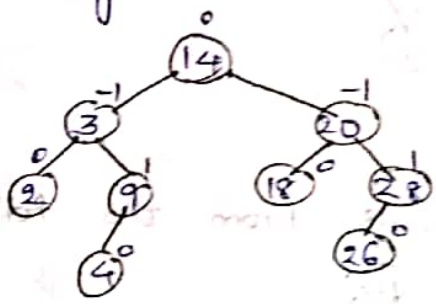
Step-2 :- R_0 [L-L Rotation]



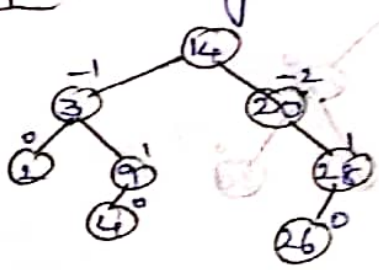
Step-3 :- Deleting 30. [No problem].



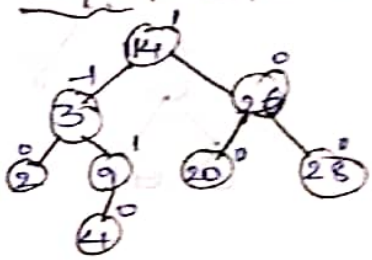
Step-4 :- Deleting 15.



Step-5 :- Deleting 18.



step-6 :- L+1 [R-L rotation]



* B-Tree :-

→ It is used to fetch the Records from 'Disk' to 'Main memory' very fastly and easily.

→ In a 'm-way tree', a node can stores 'm-1 records'

* Properties of B-Tree :-

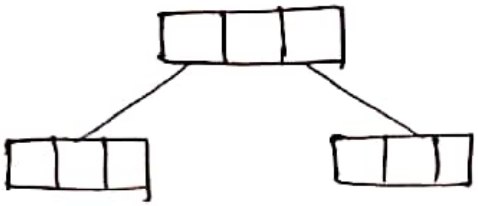
① Each Node, if order of B-Tree is 'm' then each node have 'm children' and 'm-1 ways'

② Except root node, all other nodes can have atleast 'm/2 children'. $\lceil m/2 \rceil$.

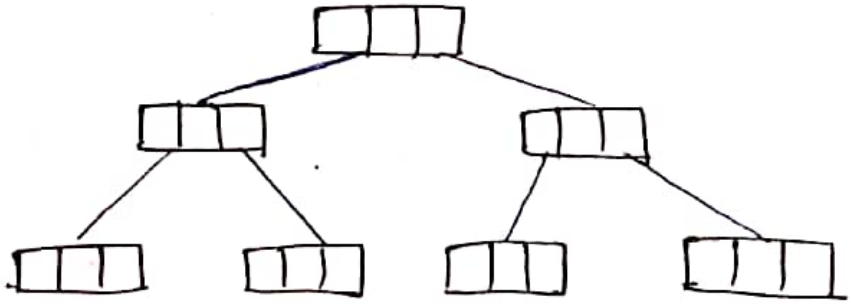
(or)

Except root node, all other nodes should have atleast 'm/2 keys' $\lceil m/2 \rceil$

③ If a root is non-terminal node, then it should have atleast '2 children'.



④ All Leaf nodes must be at same level.



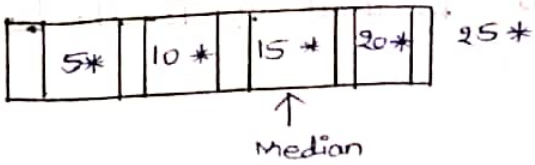
⑤ It should follow the properties of B.S.T.

* Create a B-Tree with the following values.

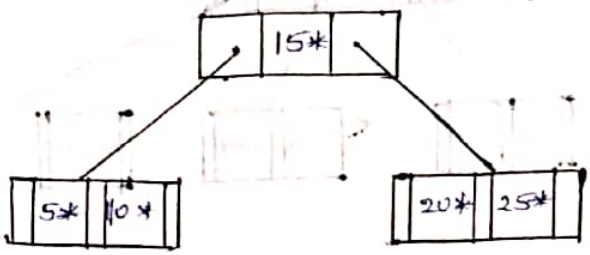
5, 10, 15, 20, 25, 30, 35, 40, 45, 50, 95, 90, 85, 80, 75, 70, 65, 60, 55, 42, 52, 62, 72, 82, 92, 12, 22, 32, 102.

A) Let us assume the order of B-Tree as 5.

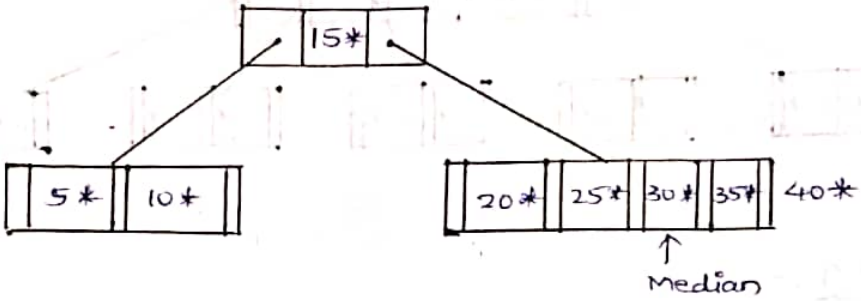
step-1 :-



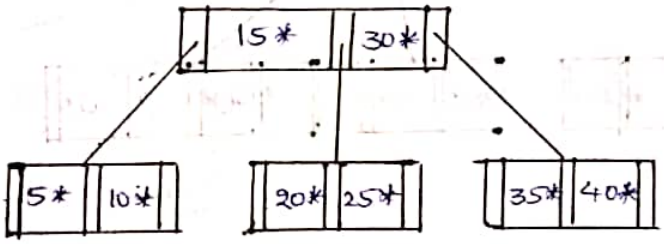
step-2 :-



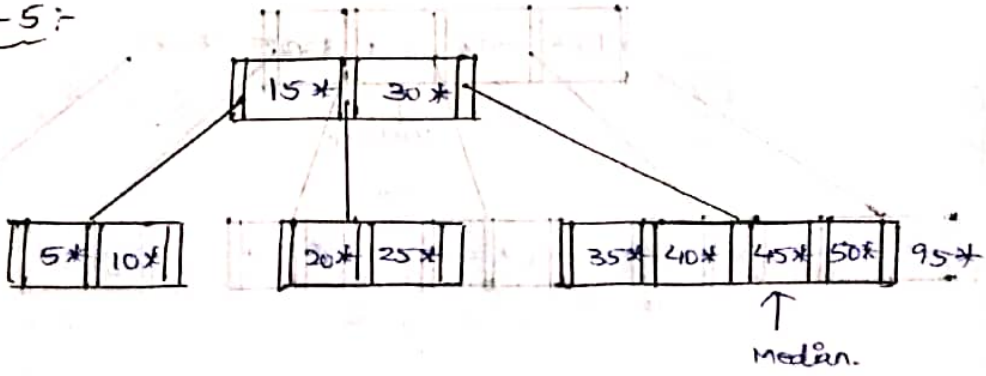
step-3 :-



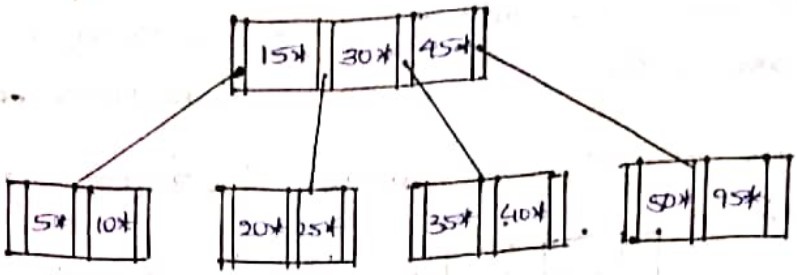
step-4 :-



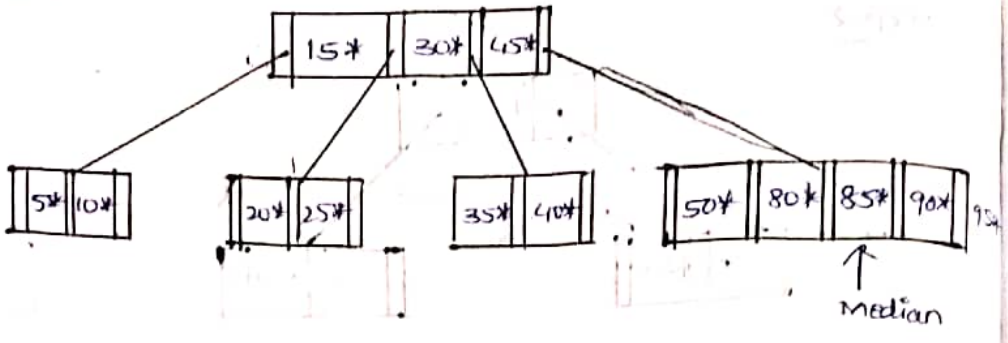
step-5 :-



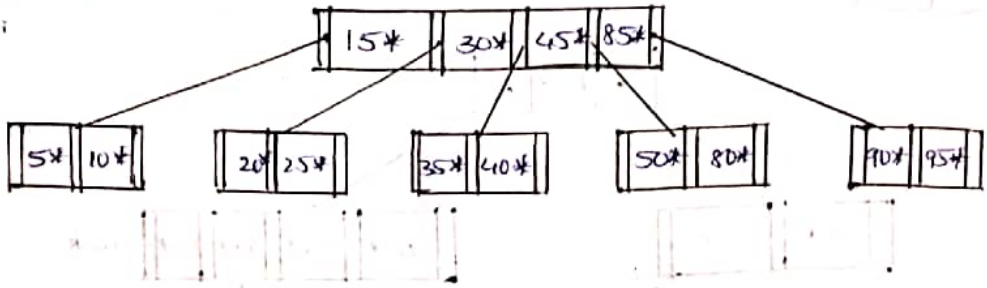
step-6 :-



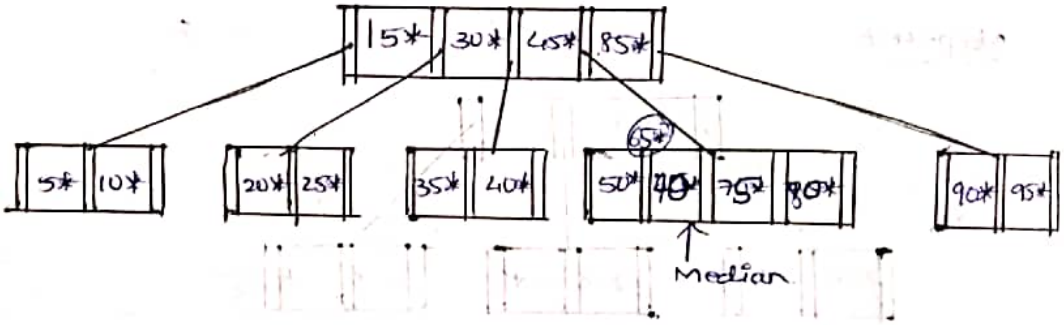
step-7 :-



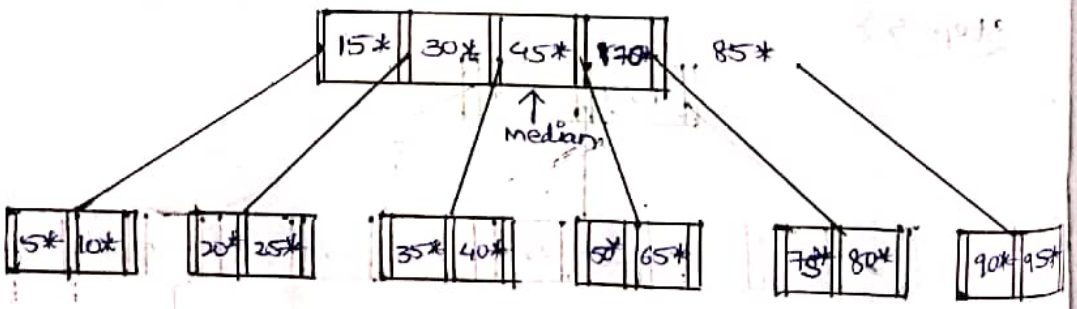
step-8 :-



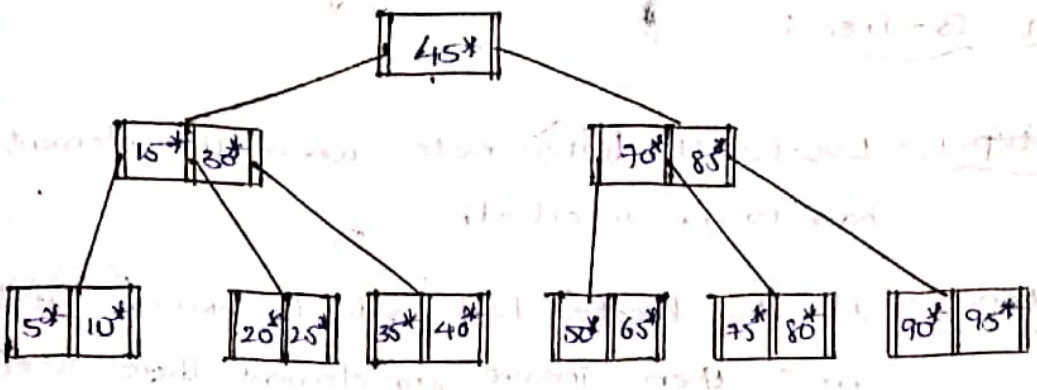
step-9 :-



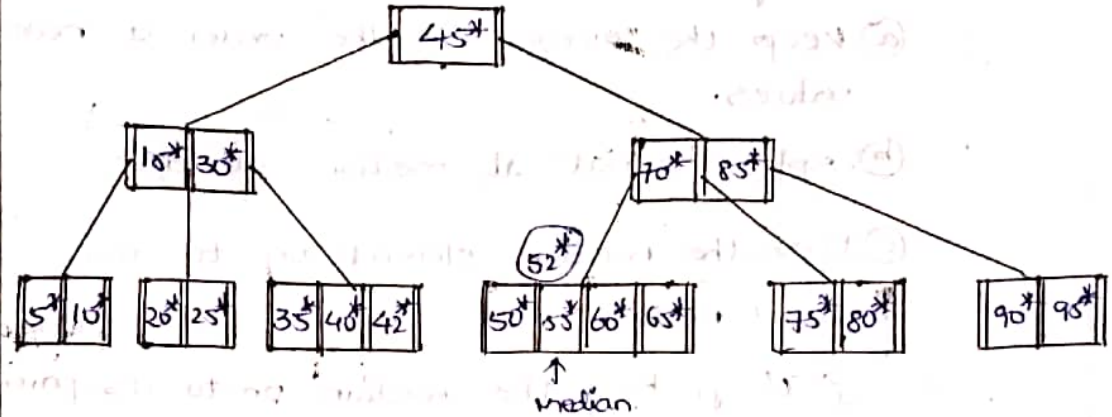
step-10 :-



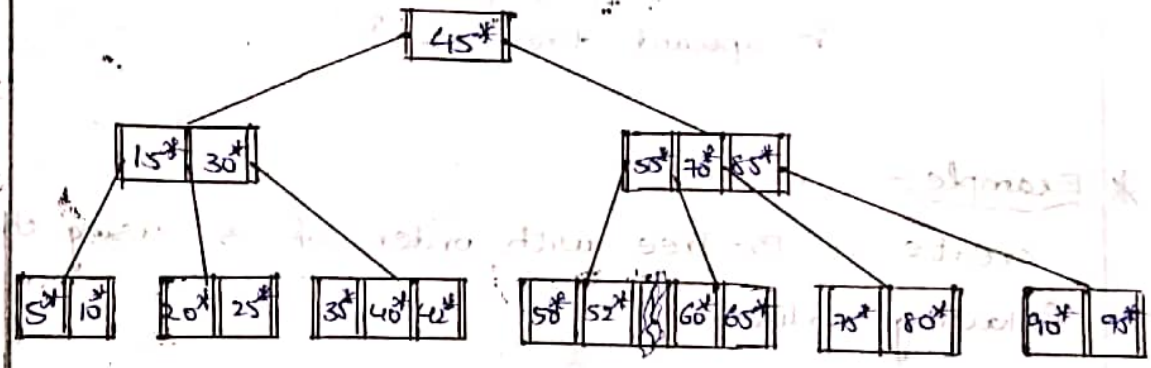
Step-11 :-



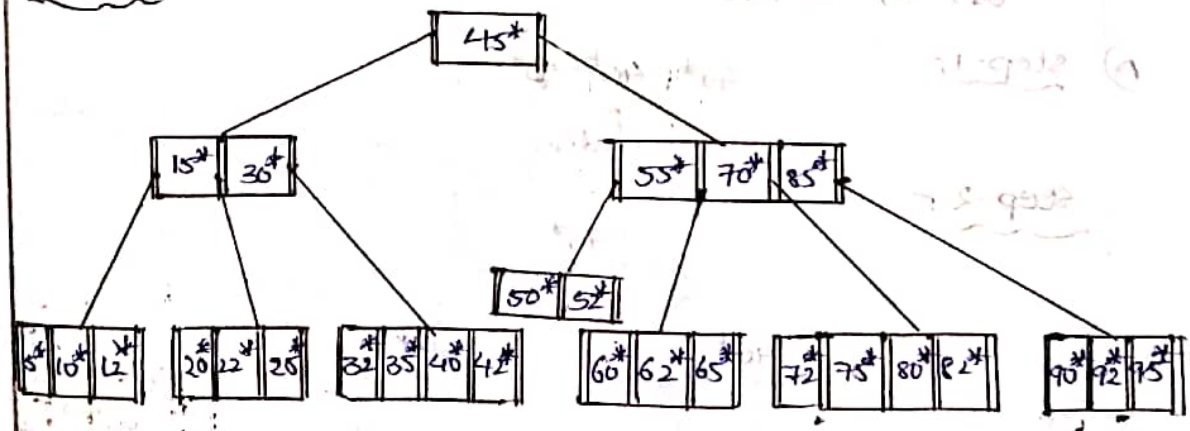
Step-12 :-



Step-13 :-



Step-14 :-



* Algorithm for creation / Inserting an element into a B-Tree :-

Step-1 :- Locate the leave node where the element has to be inserted.

Step-2 :- If the located leaf node is having sufficient space then insert an element there itself.

Step-3 :- If the located leaf node become full then following the below steps.

(a) keep the element in the order of node values.

(b) split the node at median element.

(c) Push the median element on to its parent node.

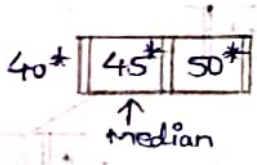
(i) If pushing the median on to its parent node, if the parent node already became full then propagate the above process in upward direction.

* Example :-

create a B-Tree with order of 3 using the following values.

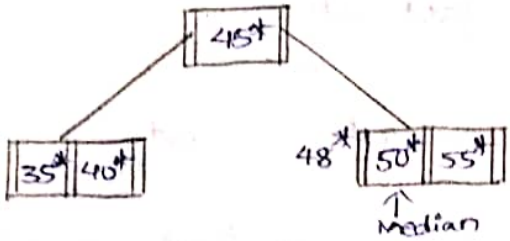
50, 45, 40, 35, 55, 48, 60, 65, 25, 30, 15, 10, 5, 68, 70, 75, 80

A) Step-1 :-

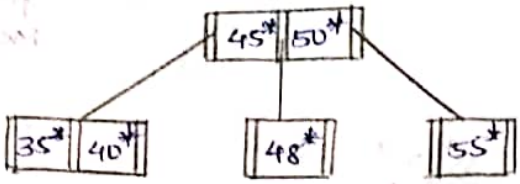


Step-2 :-

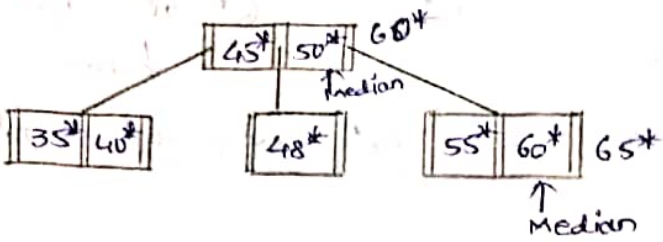
step-3 :-



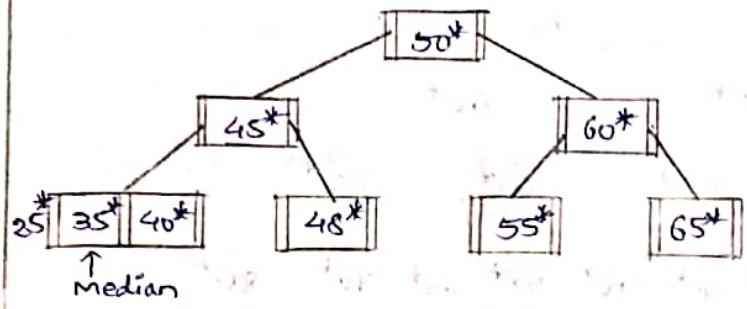
step-4 :-



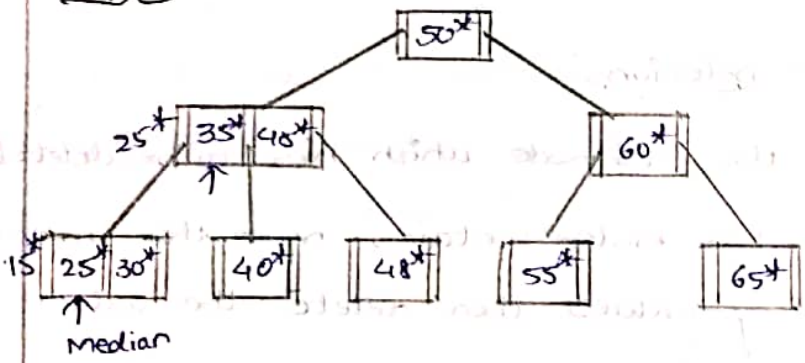
step-5 :-



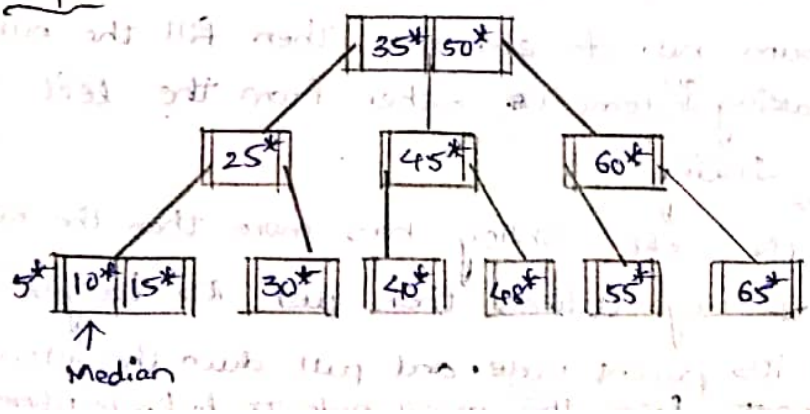
Step-6 :-



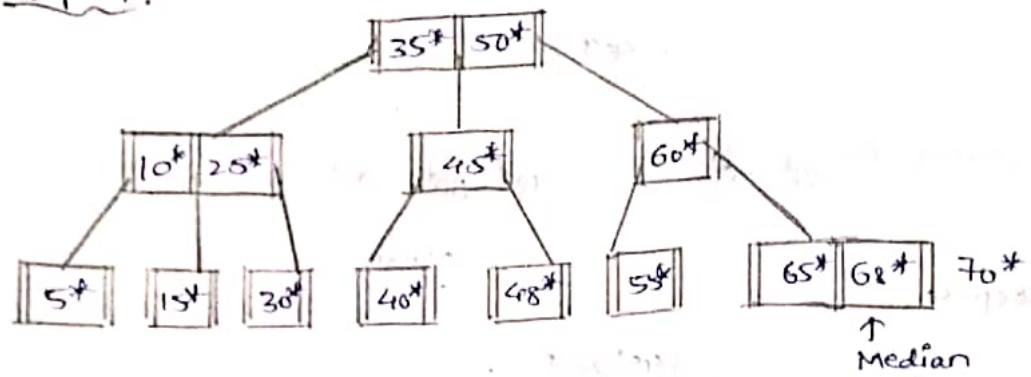
Step-7 :-



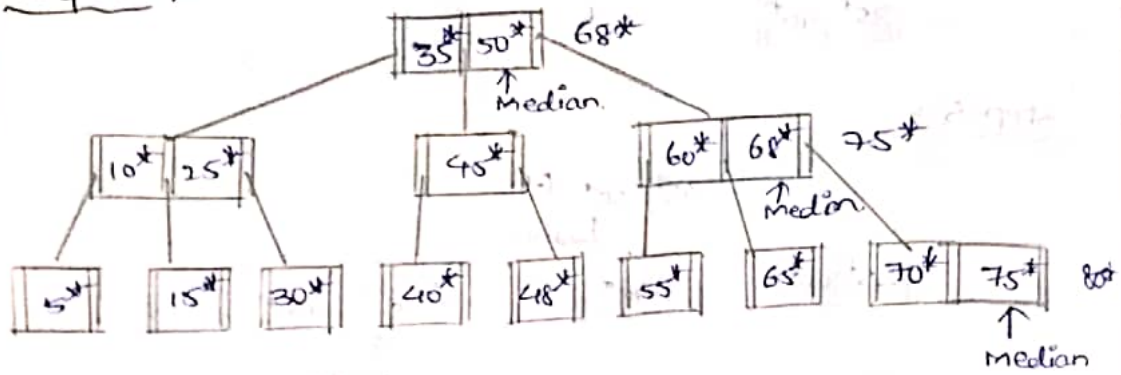
Step-8 :-



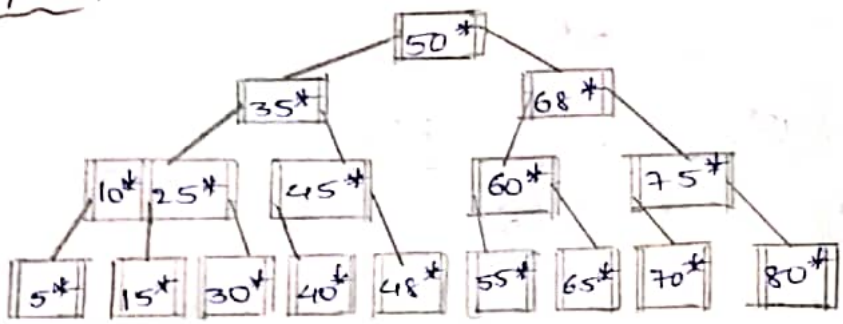
Step-9 :-



Step-10 :-



Step-11 :-



* Algorithm for Deletion :-

step-1 :- Locate the leaf node which has to be deleted.

step-2 :- If the Leaf Node contains more than minimum no. of key values then delete the value.

step-3 :- else if the leaf node doesnot contains even minimum no. of elements then fill the node by taking an element either from the left or right sibling.

(a) If the left sibling has more than the minimum no. of key values then push its Largest key into its parent node. and pull down the intervening element from the parent node to leaf node where key is deleted.

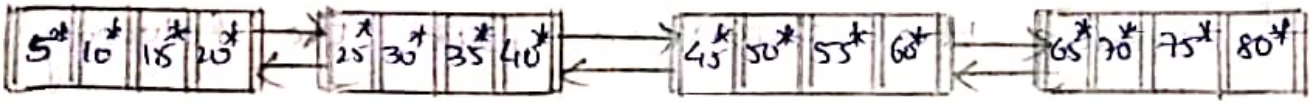
(b) else if the right sibling has more than minimum no. of key values then push its smallest key into its parent node & pull down the intervening element from the parent node to leaf node where the key is deleted.

Step-4 (c) else if both left & right siblings contains only the minimum no. of elements then create a new leaf node by combining the two leaf nodes and the intervening element of the parent node.

If the pulling the intervening element from the parent node leaves it with less than the minimum no. of keys in the node then propagate the process in upward direction. Thereby Reducing the height of the B-Tree.

* B+ Tree :-

→ Let us assume the leaf nodes of B-Tree with order-5 as follows.



→ The above is called as B+ Tree.

Advantages :

→ It is used to retrieve the range of records easily.

Ex:- ① Employees details from ID 5000 to 10,000.

② Accessing a value from 20 to 60.

* Red-Black Tree :-

→ It is used to minimize the no. of Rotations in an AVL Tree.

Properties :-

(1) Any node in the tree should be coloured either with red or with black.

(2) Root node must be in Black colour.

(3) Red-Black Tree shouldnot contains Red-Red sequence in any path from root node to leaf node.

(4) The no. of black nodes from any path from root node to leaf node must be equal.

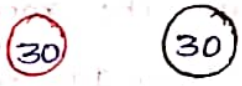
(5) It should follow the properties of B.S.T.

* creation of R.B.Tree :-

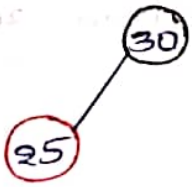
create the R-B Tree with the following values

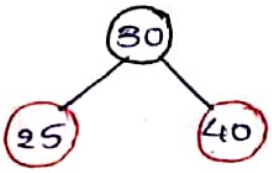
30, 25, 40, 28, 45, 50, 48, 55, 60, 42, 35, 20, 15, 10, 65

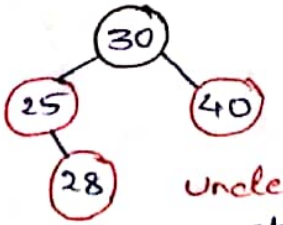
A).-In Insertion, always we are going to insert a value in red colour.

Step-1 :- 

But Root node is Black.

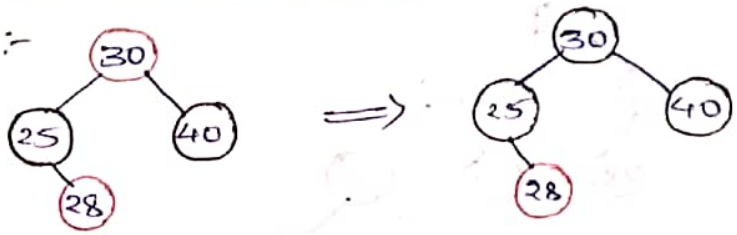
step-2 :- 

step-3 :- 

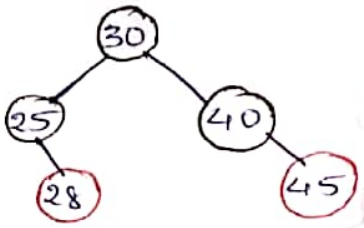
step-4 :- 

Uncle Node case - We need to change Parent node color.

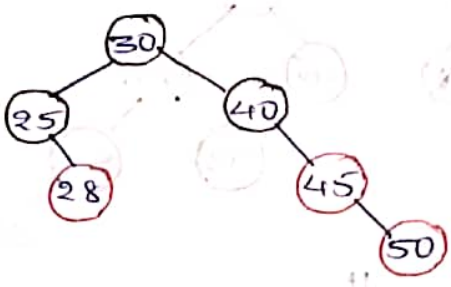
step-5 :-



step-6 :-

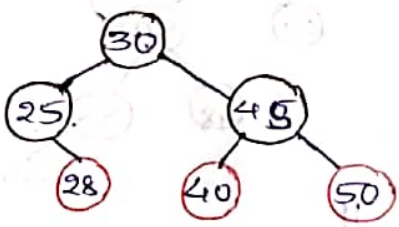


step-7 :-

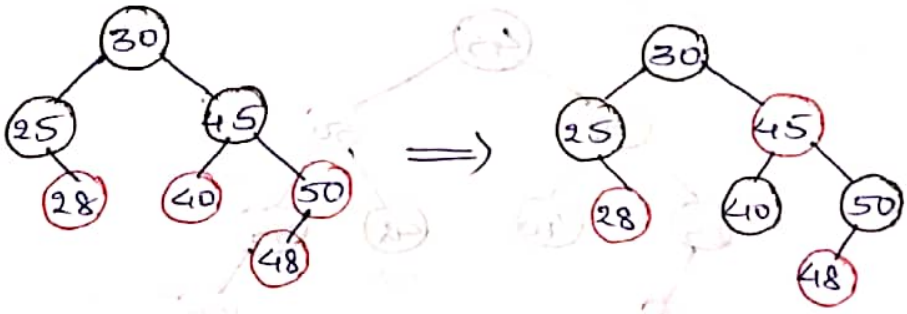


→ Red-Red sequence occured & uncle-Node is also absent.

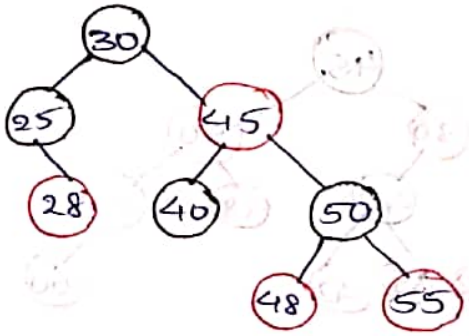
→ so we need to apply rotation.



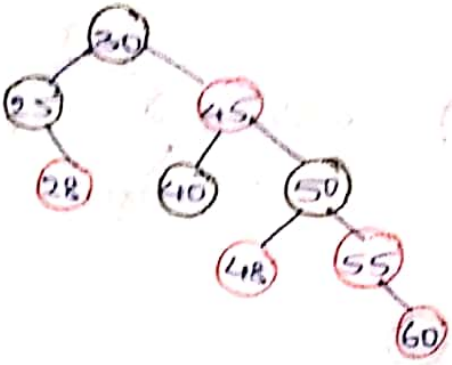
step-8 :-

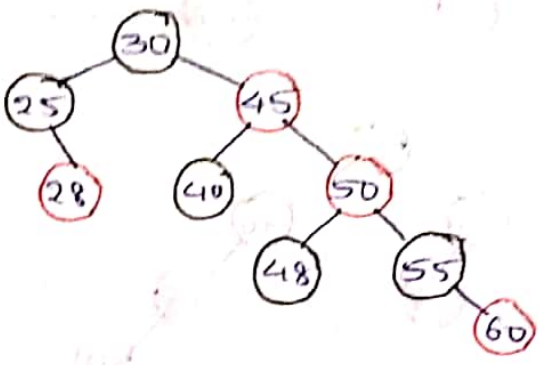


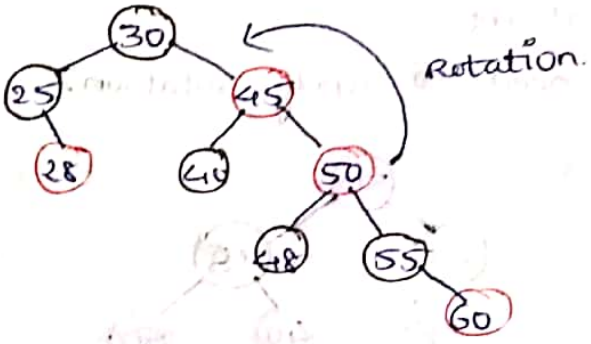
step-9 :-

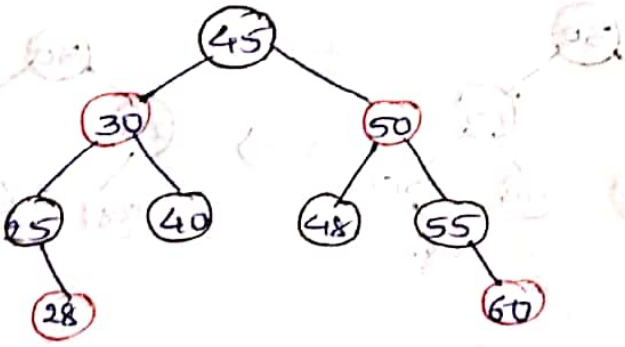


Step-10 :-

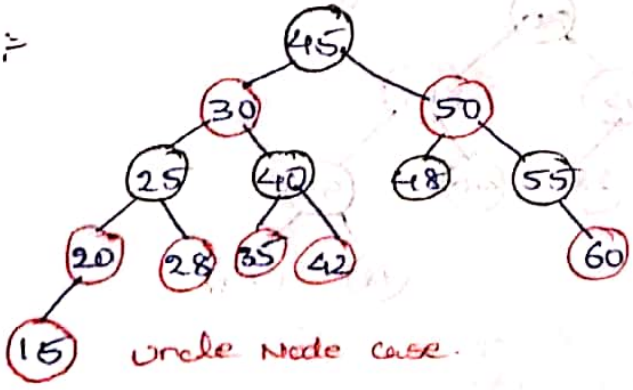






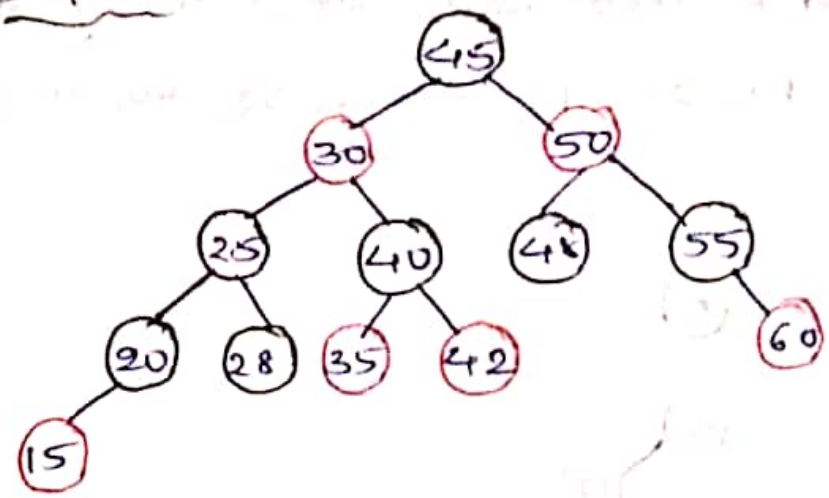


Step-11 :-

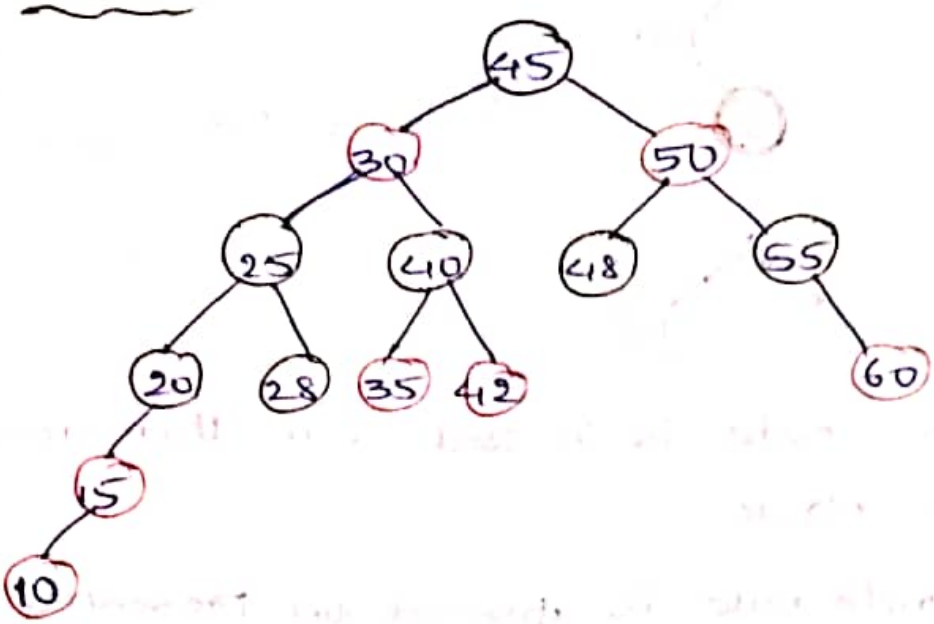


Uncle Node case.

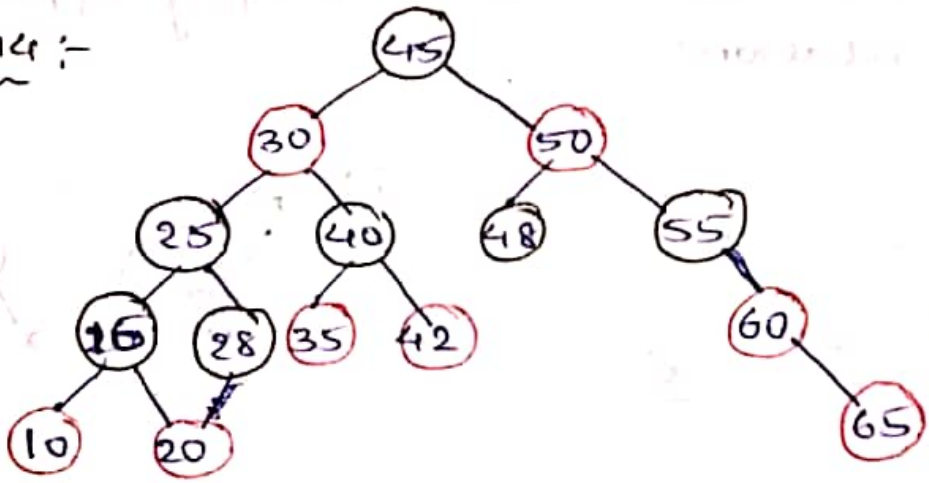
step-12 :-



step-13 :-



step-14 :-



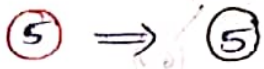
Rotation.

* Create a Red-Black Tree with the following values

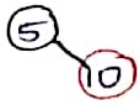
5, 10, 8, 9, 12, 1, 15, 25, 18, 30, 35, 32, 40, 50, 45, 46, 65, 60.

A)

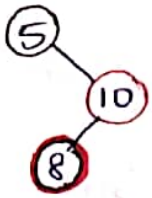
step-1 :-



step-2 :-

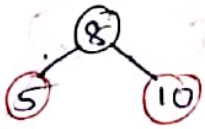


step-3 :-



Red-Red sequence

step-4 :-

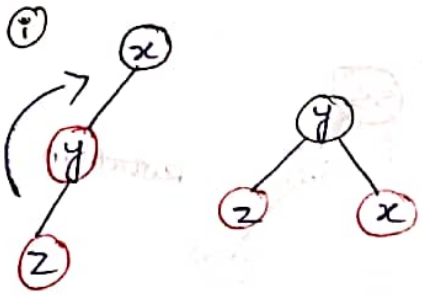


cases :-

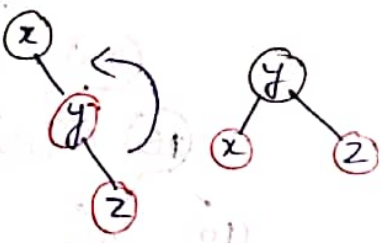
case-1 :- If root node is in Red color then recolor it as black.

case-3 :- If Uncle node is absented (or) presented with black color, then apply appropriate rotation.

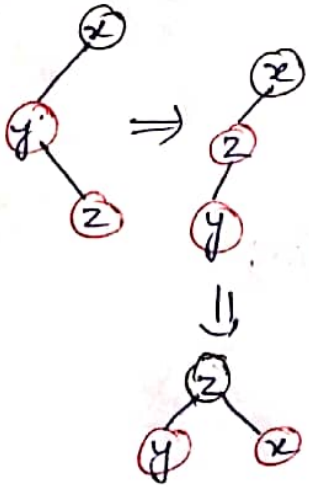
(i)



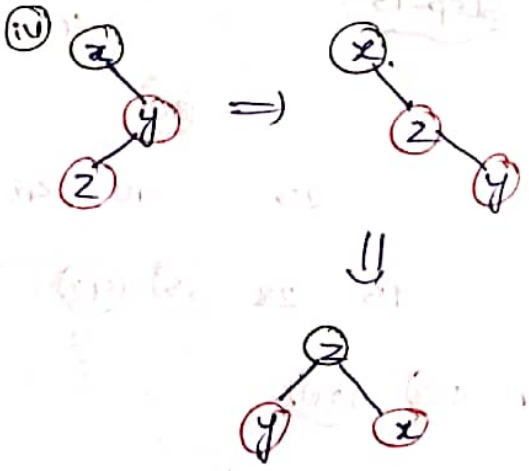
(ii)



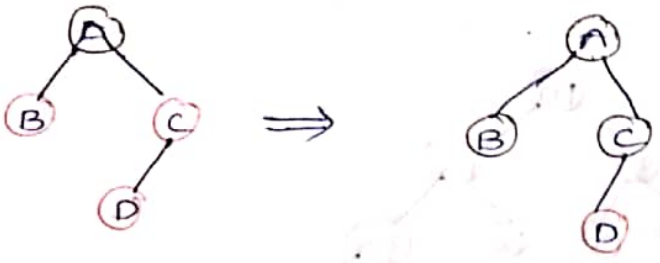
(iii)



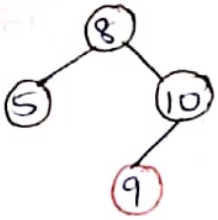
(iv)



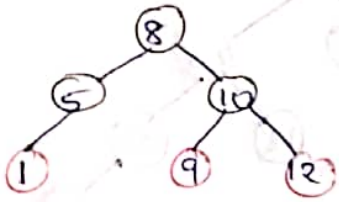
Case-2:- If the uncle node is presented with red color, then recolor its parent node, recolor parent sibling node [uncle node] & recolor their parent node.



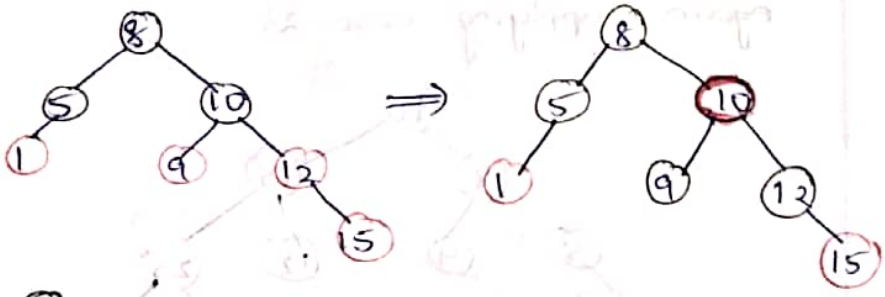
step-5:-



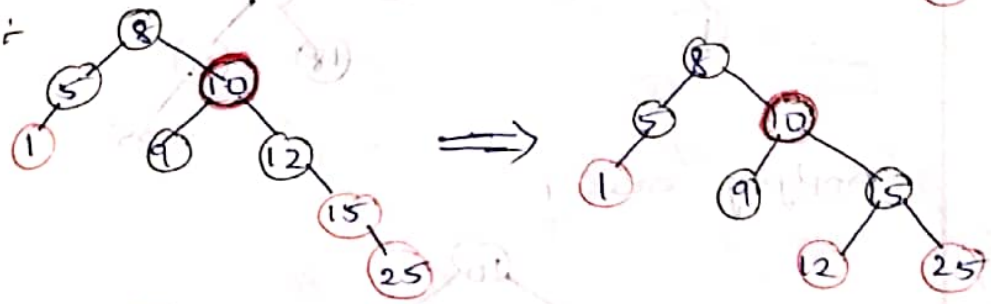
step-6:-



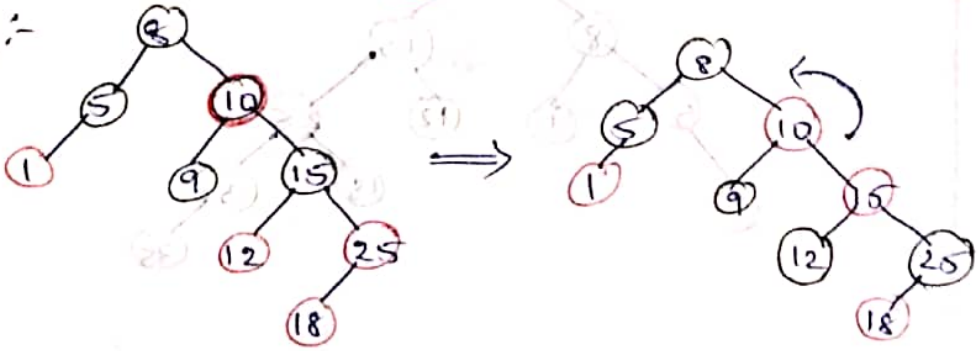
step-7:-

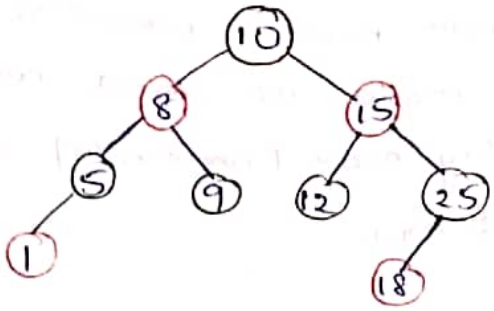


step-8:-

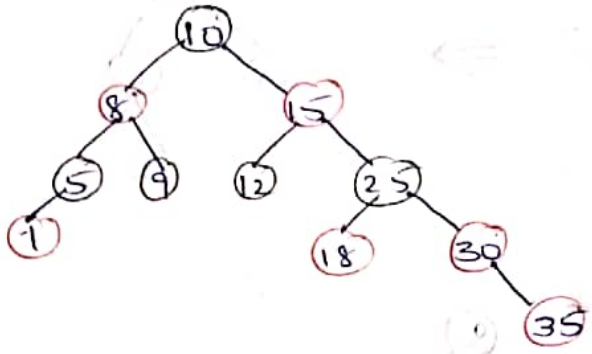


step-9:-

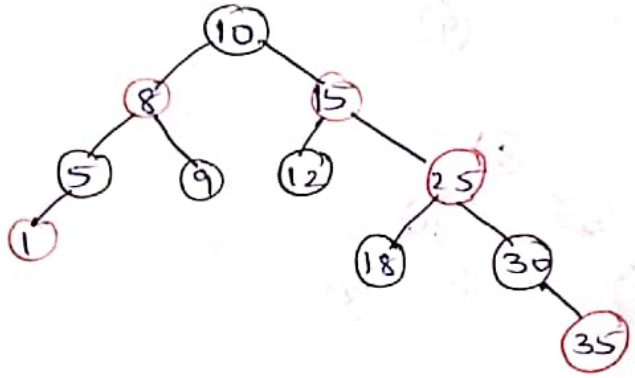




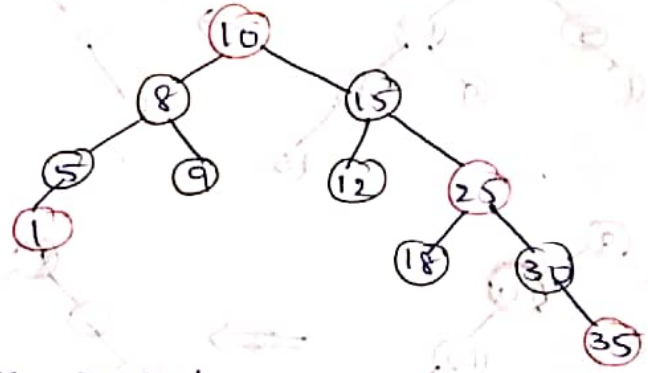
step-10 :-



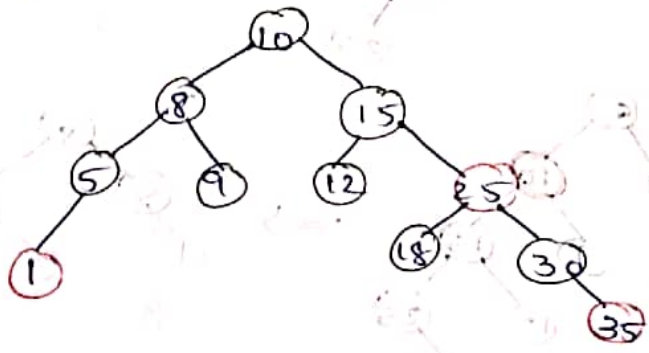
Applying case-2.



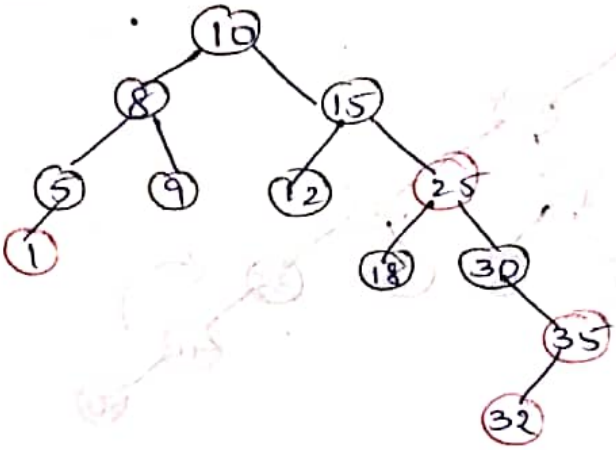
Again applying case-2.



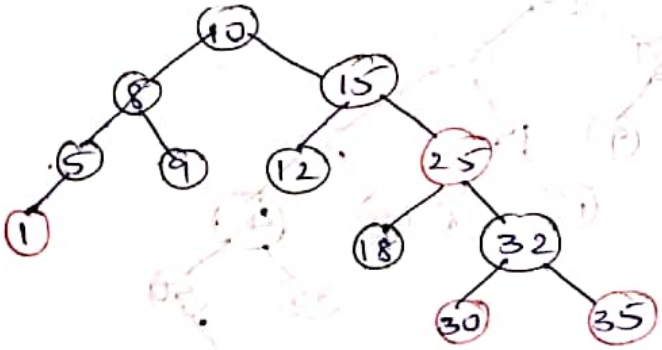
Applying case-1



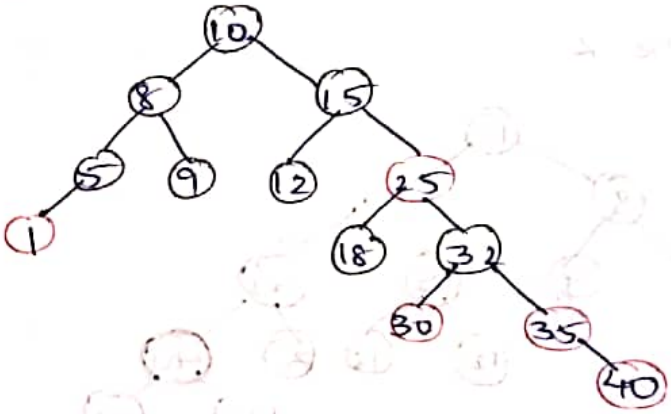
step-11 :-



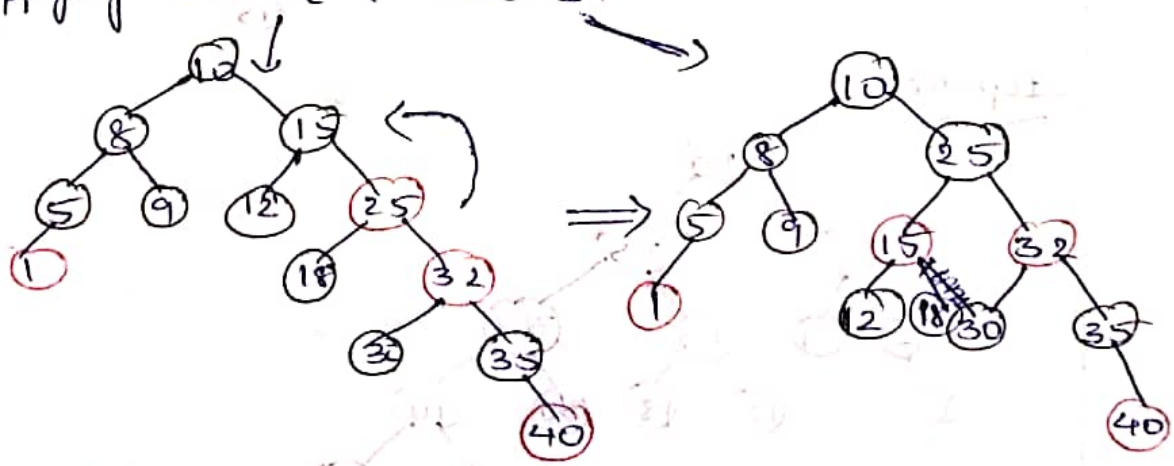
Applying case-3, [2 - Rotations].



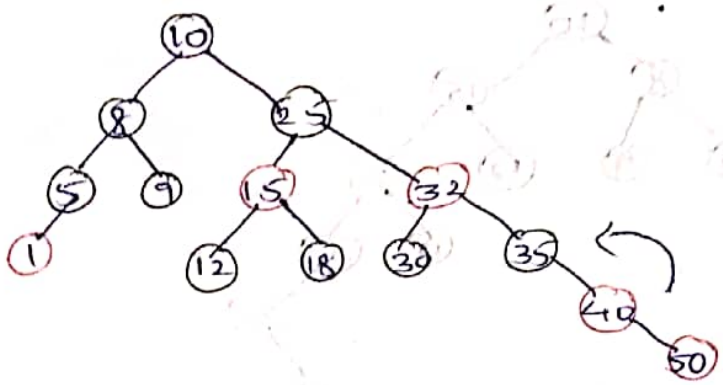
step-12 :-



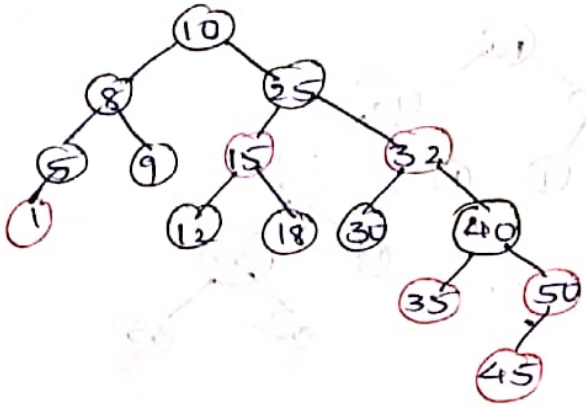
Applying case-2 & case-3



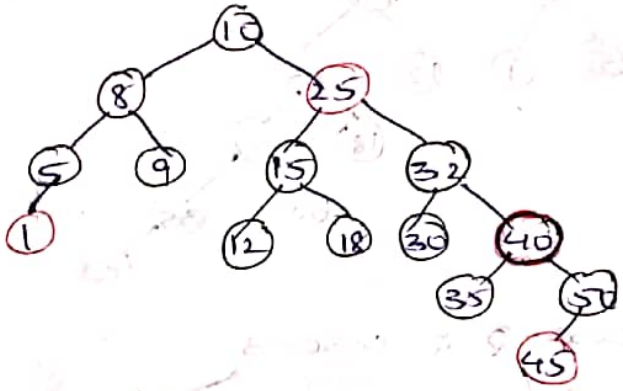
step-13:



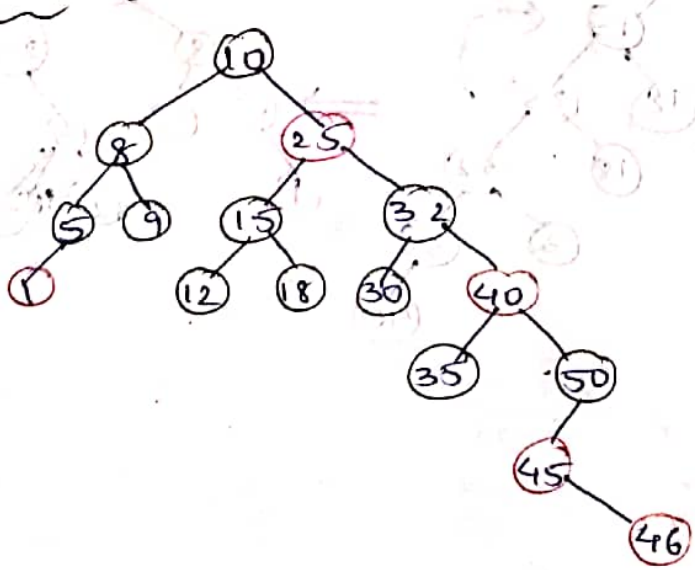
Applying case-3:



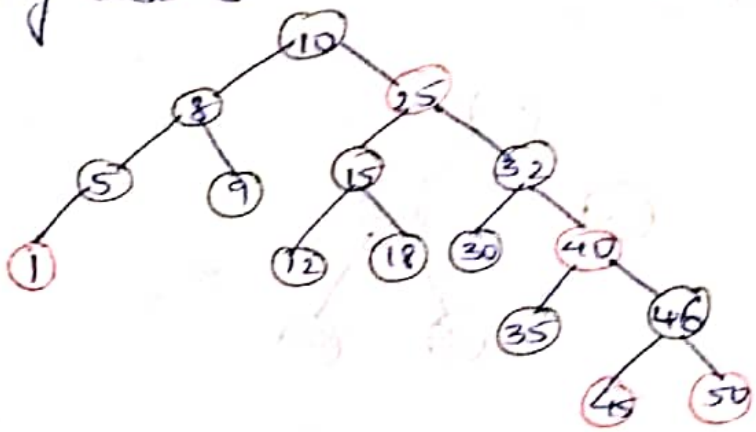
Applying case-2.



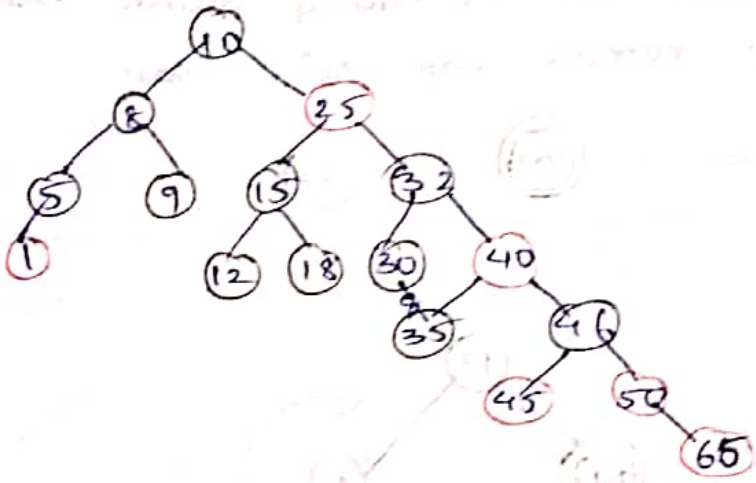
step-14:



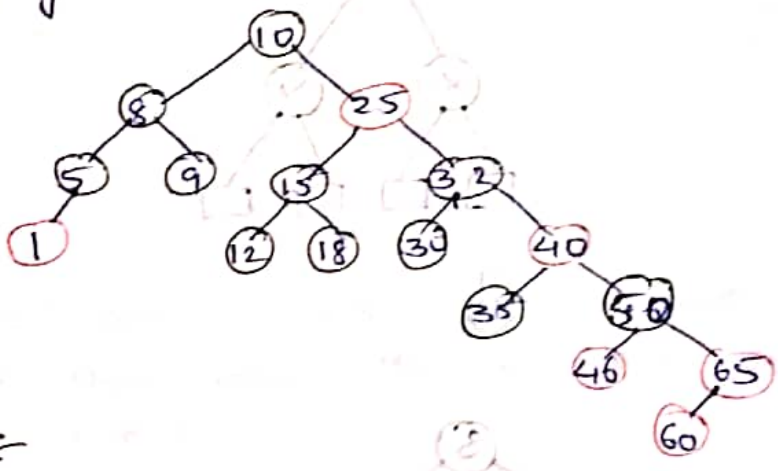
Applying case-3 :



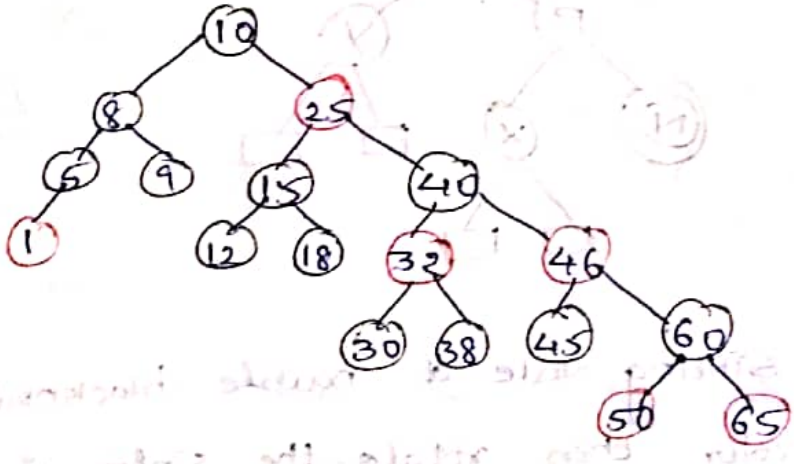
step-15 :-



Applying case-3 :

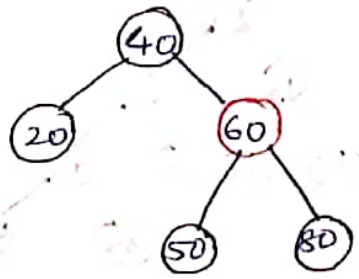


step-16 :-



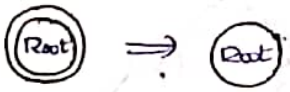
* Deletion operation :-

Ex:-

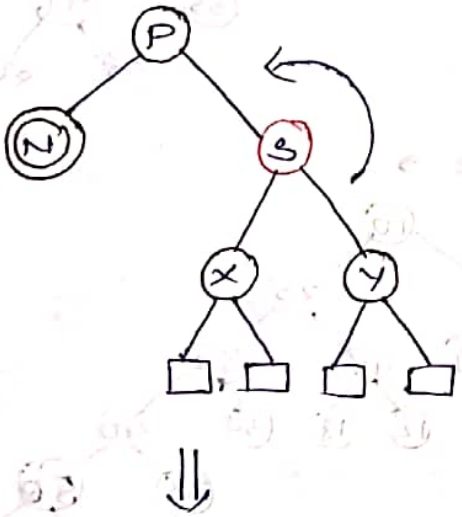


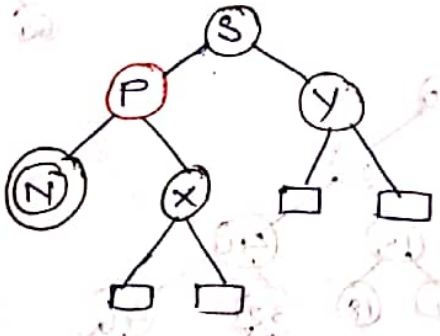
case-1 :-

→ If root node is having double black colour then remove one Black colour.



case-2 :-

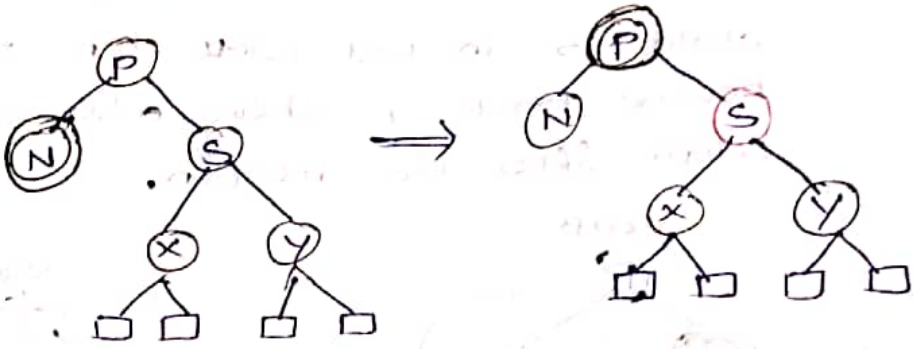




→ If sibling Node of Double blacknode is in Red color then rotate the sibling in L.L direction and append Red color to parent Node.

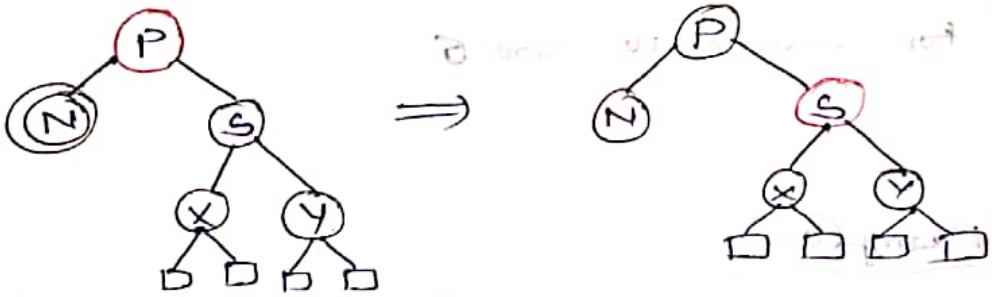
case-3 :-

→ If all are in black color, then make parent node as double black node & append red color to sibling node.



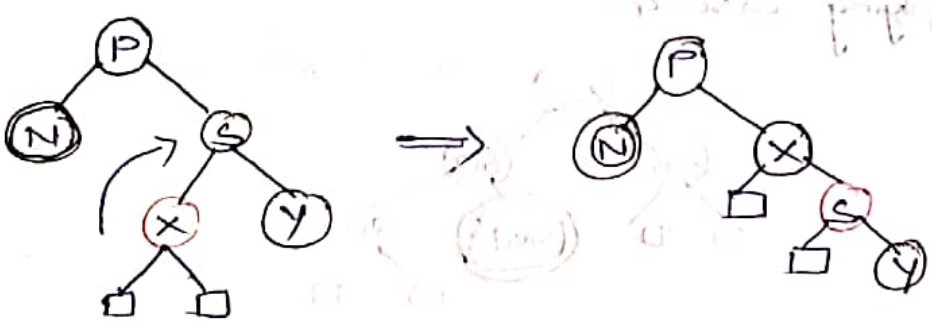
case-4 :-

→ If all are in black color except parent node then make parent color as black [by moving one black color from N] and sibling as red.



case-5 :-

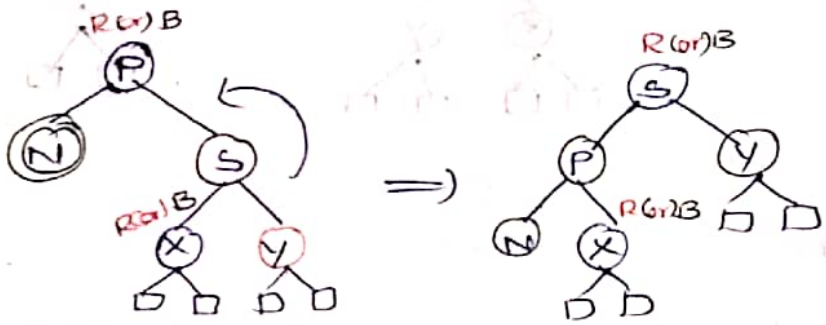
→ If all are in black colour except sibling's left child then rotate the x and s nodes and give red color to s.



→ This is not Terminating case since Double Black Node is still present.

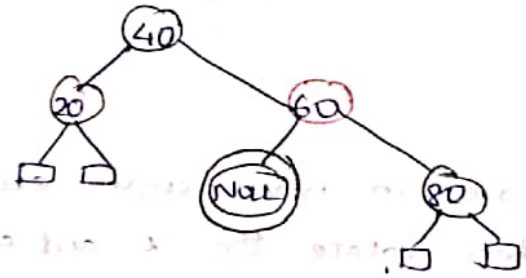
Case-6 :-

→ If parent and sibling's Left childs are in either red or Black colour and sibling's right child is in Red colour then parent node colour become black & sibling colour is taken parent's colour after the rotation.

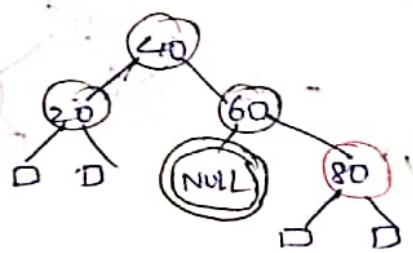


Case-7 to 11 :- These are mirror cases [symmetric] for case-2 to case-6.

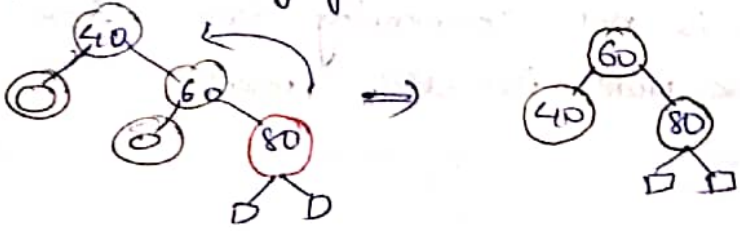
* Example :-



A) Applying case-4 :-

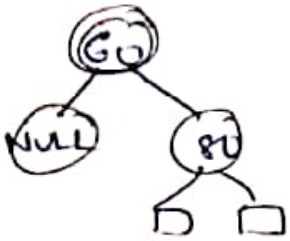


Deleting 20 & Applying case-6.

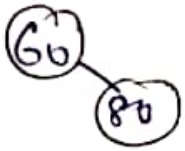


Deleting 40

Applying case-3.



Applying case-1



Deleting 80.

(or)

Deleting 60

* Applications of Trees :

(1) Used to sorti the elements.

(2) storing different/unique records [No duplicates].

(3) To store the data in Hierarchial structure.

(4) Less Accessing Time.

19-06-2023

UNIT - 6. GRAPHS

* Graph :-

→ 'Graph' is a Non-linear Data structure.

→ The difference b/w 'Tree' & 'Graph' is tree doesnot contain cycles but 'graph' may contains 'cycles'.

→ Graph contains 'vertices' & 'Edges'; G(V, E).

→ Every 'Tree' is a 'Graph' but every 'graph' need not be a 'tree'.

→ Graph is classified as
  ① 'Directed graph'
  ② 'Undirected graph'.

* Types of Graphs :-

① Regular Graph :-

→ The degree of every vertex is same in the graph is called 'Regular graph'.

② cyclic Graph :-

→ The graph which contains cycle is called 'cyclic graph'.

③ complete Graph :-

→ If every vertex of graph connected to other vertex then the graph is called 'complete graph'

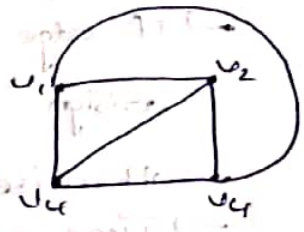
④ Bipartite Graph :-

→ set-1 vertices connected to the vertices of set-2 is called 'Bipartite Graph.'

→ If all the vertices are connected in both sets the it is called 'complete Bipartite Graph'.

* Note :-

→ If there are 'n' vertices then the complete graph contains $\frac{n \times (n-1)}{2}$ edges.



Ex :- 4 vertices

Edges = $\frac{4 \times 3}{2} = 6$.

* Degree :-

→ No. of edges connected to a vertex is called 'Degree of vertex'.

* simple graph :-

→ If they are no parallel edges in a graph then it is called 'simple graph'.

* Multigraph :-

→ Having parallel edges in a graph.

* In-Degree & out-Degree :-

→ The Incoming edges towards a vertex in a Directed graph is called 'In-Degree'.

→ The No. of outgoing edges from a vertex in a Undi Directed graph is called out-degree.

* connected graph :-

→ If a graph contains path which is passing through all vertices then it is a connected graph

* Isolated Node / Vertex :-

→ A vertex having 'degree-0' then it is called Isolated vertex.

* Graph Representations :-

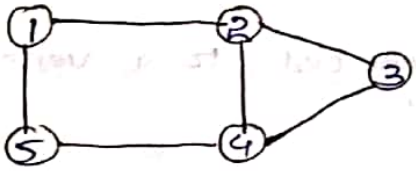
① Adjacency Matrix :-

→ If edge is present b/w two vertices then we will assign '1' to respective position in a matrix

→ otherwise we will assign '0'.

→ Time complexity is 'O(v²)'.

Ex :-



Adjacency Matrix :

	1	2	3	4	5
1	0	1	0	0	1
2	1	0	1	1	0
3	0	1	0	1	0
4	0	1	1	0	1
5	1	0	0	1	0

DrawBack :-

→ If the no. of vertices are more and then no. of edges will be less. Then Adjacency matrix is not suitable.

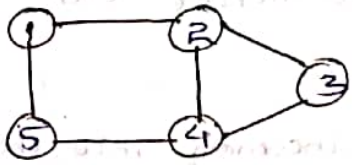
Ex :- 100 x 100 matrix, needs 10,000 memory locations to store the values.

→ If given matrix is sparce [More no. of '0's] then don't consider Adjacency matrix; it is difficult.

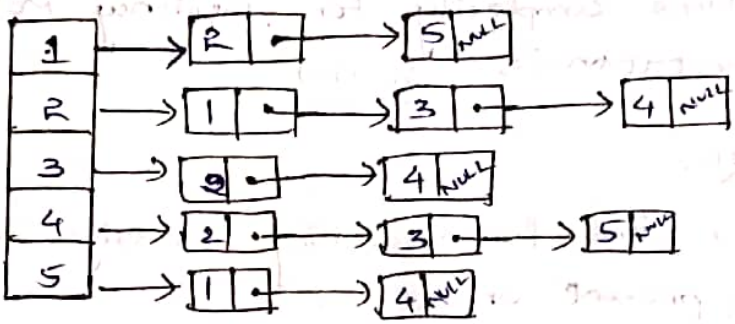
→ If given matrix is Dense [having more edges → '1'] then we can choose Adjacency matrix representation

(2) Adjacency List Representation :-

Ex:-



Adjacency List :-



→ The Time complexity is 'O(V+E)'

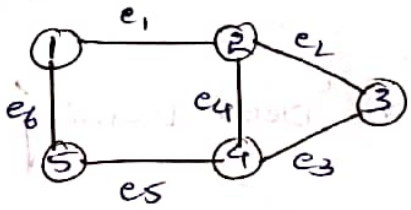
Drawback :-

→ The time complexity in worst case for searching an edge is present or not is 'θ(n)'.

Advantage :-

(3) Incidency Matrix Representation :-

Ex:-



Incidency Matrix :-

	e_1	e_2	e_3	e_4	e_5	e_6
1	1	0	0	0	0	1
2	1	1	0	1	0	0
3	0	1	1	0	0	0
4	0	0	1	1	1	0
5	0	0	0	0	1	1

→ If the graph is Directed graph, if an edge is outgoing from a vertex then the value is '1'.

→ If the edge is Incoming into the vertex then the value is '-1'.

→ The Time complexity for Incidency Matrix representation is O(VE).

Draw Back:-

→ Takes time for searching an edge whether it is present or not.

* Graph Traversals:-

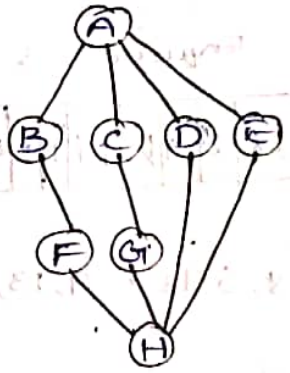
→ There are two methods to traverse the Graph.

① "Breadth First search [BFS]" → Queue concept.

② "Depth First search [DFS]" → By stacks concept.

① Breadth First search [B.F.S] :- [Level-order Traversal]

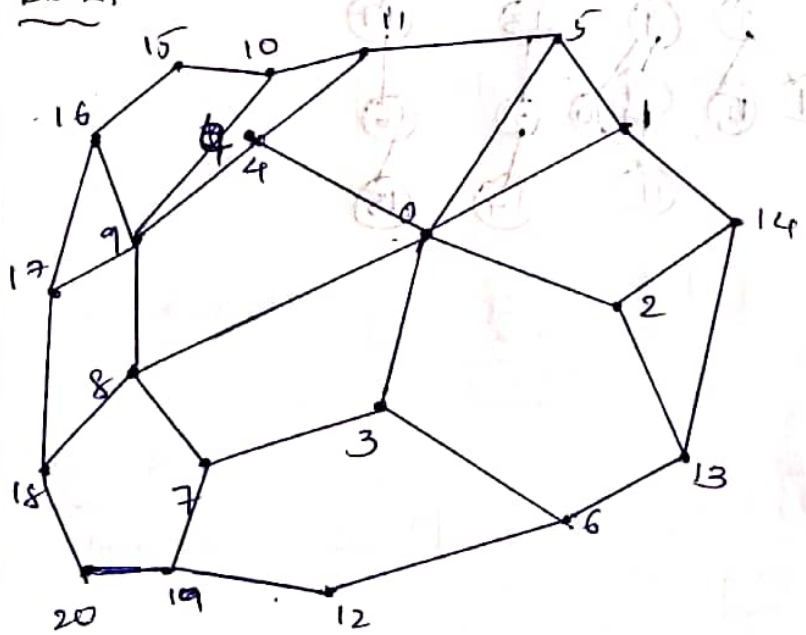
Ex-1



A	B	C	D	E	F	G	H

o/p :- A, B, C, D, E, F, G, H

Ex-2 :-

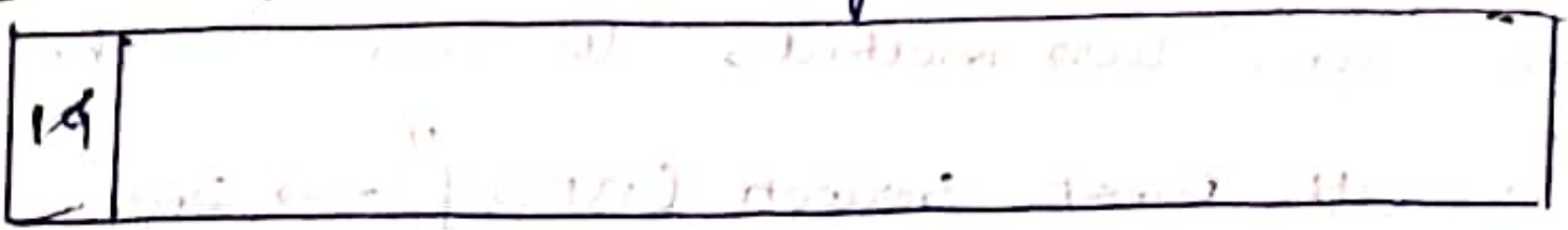


Let,

starting vertex is 19. — for ~~B.F.S~~ B.F.S

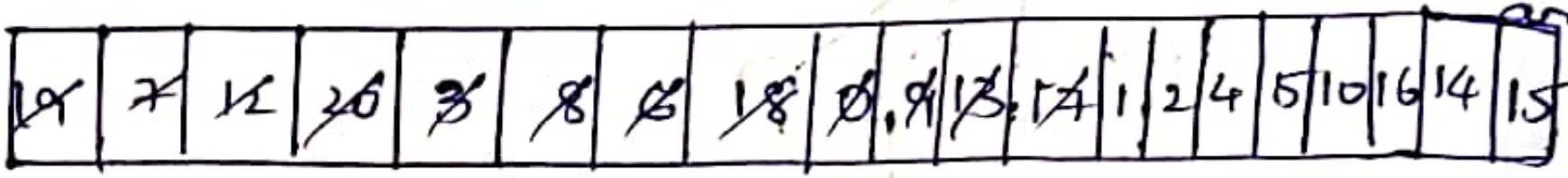
starting vertex is 0. — for D.F.S

Step-1 :- Enqueue the starting vertex



O/p :- 19

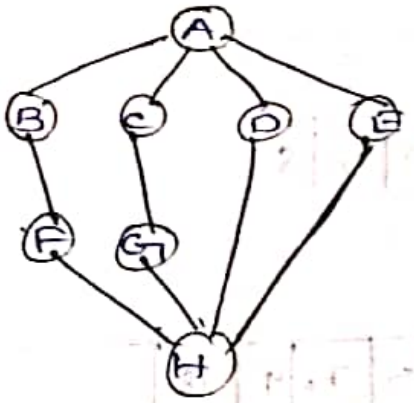
Step-2 :- Dequeue 7 and Enqueue its adjacent vertices.



O/p :- 19, 7, 12, 20, 3, 8, 6, 18, 0, 9, 13, 17, 1, 2, 4, 5, 10, 16, 14, 15.

(2) Depth First Traversal :-

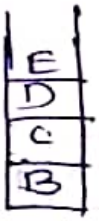
Ex :-



step-1 :-

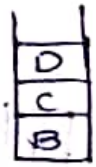
step-2 :-

o/p : A



step-3 :-

o/p : A, E



step-4 :-

o/p :- A, E, H

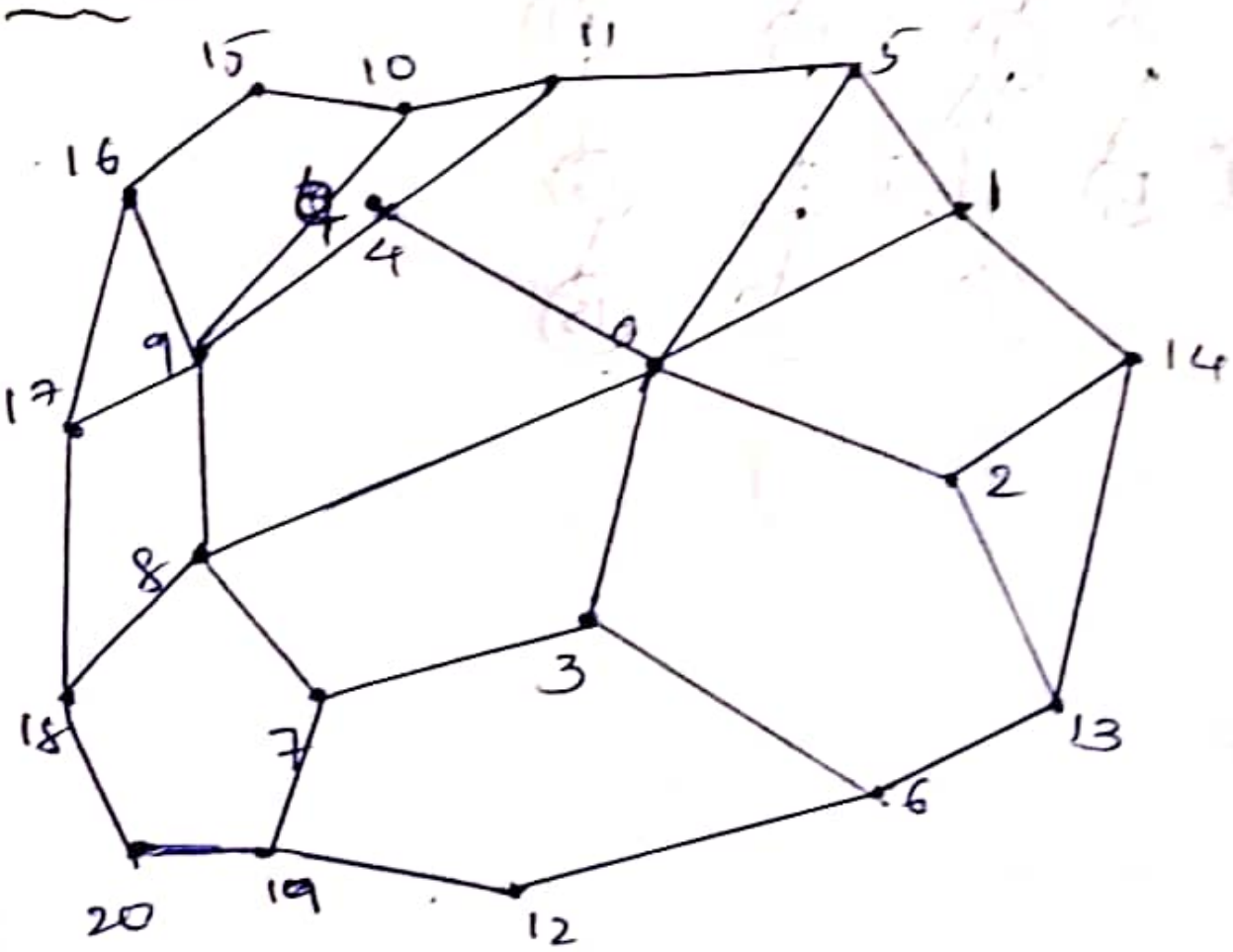
step-5 :-

o/p : A, E, H

step-6 :-

o/p : A, E, H, G, F, D, C, B

Ex-2:-



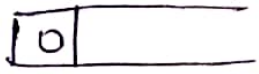
Let,

Starting vertex is 19. — for ~~B.F.S~~ B.F.S

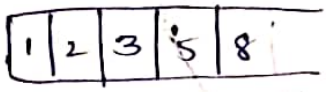
Starting vertex is 0. — for D.F.S

Ex-2:

step-1:

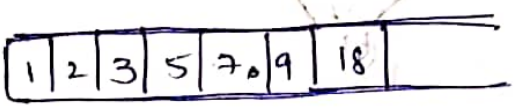


step-2:



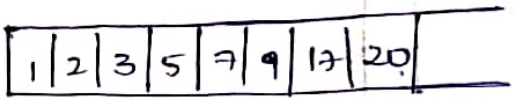
o/p:-0

step-3:



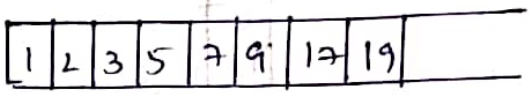
o/p:- 0, 8

step-4:



o/p:- 0, 8, 18

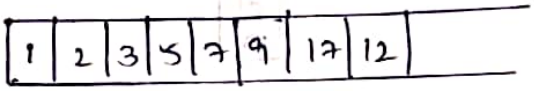
step-5:



o/p:- 0, 8, 18, 20

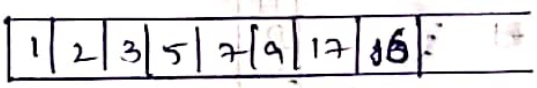
step-6:

o/p



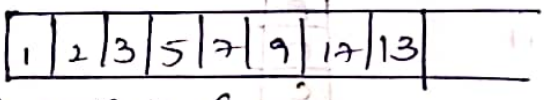
o/p:- 0, 8, 18, 20, 19

step-7:



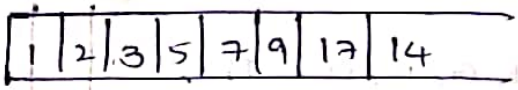
o/p:- 0, 8, 18, 20, 19, 12

step-8:



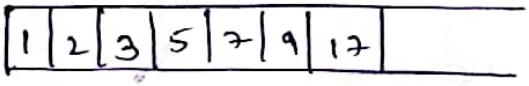
o/p:- 0, 8, 18, 20, 19, 12, 6

step-9:



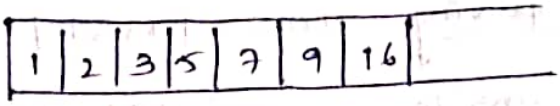
o/p:- 0, 8, 18, 20, 19, 12, 6, 13.

step-10:



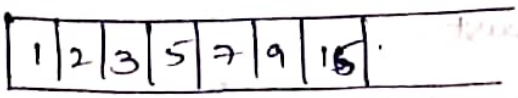
o/p:- 0, 8, 18, 20, 19, 12, 6, 13, 14

step-11 :-



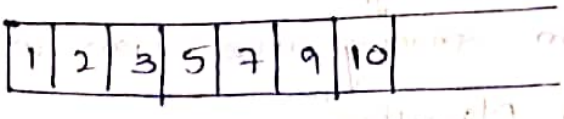
o/p :- 0, 8, 18, 20, 19, 12, 6, 13, 14, 17

step-12 :-



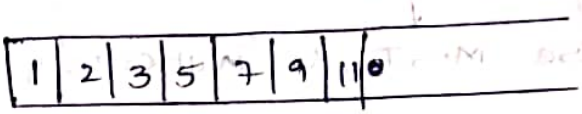
o/p :- 0, 8, 18, 20, 19, 12, 6, 13, 14, 17, 16

step-13 :-



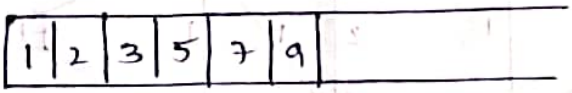
o/p :- 0, 8, 18, 20, 19, 12, 6, 13, 14, 17, 16, 15

step-14 :-



o/p :- 0, 8, 18, 20, 19, 12, 6, 13, 14, 17, 16, 15, 10

step-15 :-

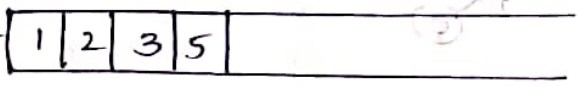


o/p :- 0, 8, 18, 20, 19, 12, 6, 13, 14, 17, 16, 15, 10, 11

step-16 :-

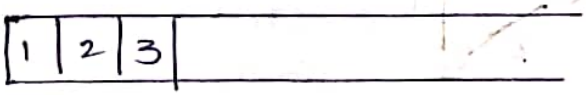
o/p :- 0, 8, 18, 20, 19, 12, 6, 13, 14, 17, 16, 15, 10, 11, 9

step-17 :-



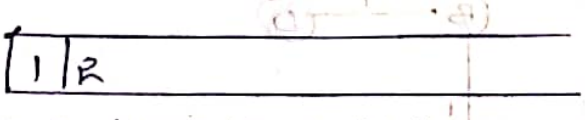
o/p :- 0, 8, 18, 20, 19, 12, 6, 13, 14, 17, 16, 15, 10, 11, 9, 7

step-18 :-



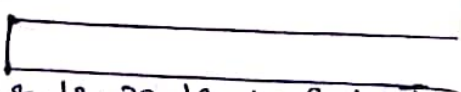
o/p :- 0, 8, 18, 20, 19, 12, 6, 13, 14, 17, 16, 15, 10, 11, 9, 7, 5

step-19 :-



o/p :- 0, 8, 18, 20, 19, 12, 6, 13, 14, 17, 16, 15, 10, 11, 9, 7, 5, 3

step-20 :-



o/p :- 0, 8, 18, 20, 19, 12, 6, 13, 14, 17, 16, 15, 10, 11, 9, 7, 5, 3, 2, 1

* Minimum Spanning Trees :- [MST]

→ Let us consider a "weighted graph", we have to create a Tree from that graph such that it should cover all the 'n' no of vertices with minimum coust.

→ The tree obtained have (n-1) edges.

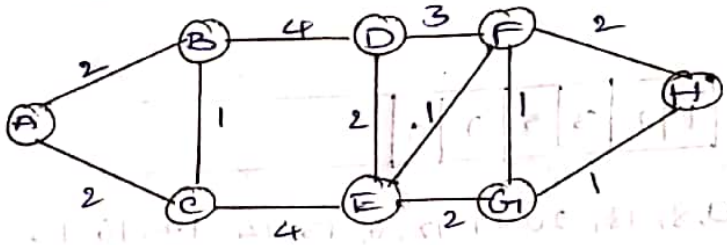
→ There are two types of methods to create a Minimum spanning Tree from weighted graph

① "Prim's Algorithm."

② "Kruskal's Algorithm."

→ We will use M.S.T. for "Multicasts" & "Broad casts" etc.

Ex :-

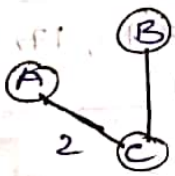


① Prim's Algorithm :-

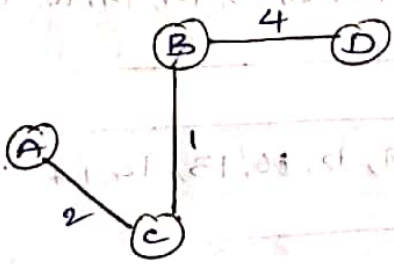
step-1 :- Take any arbitary vertex.

step-2 :-

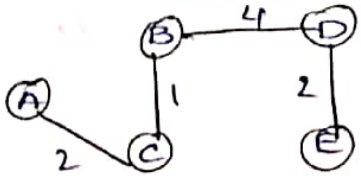
step-3 :-



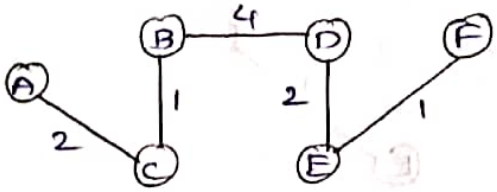
step-4 :-



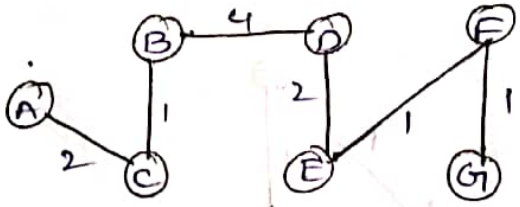
step-5 :-



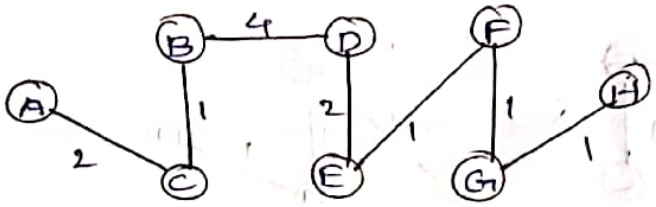
step-6 :-



step-7 :-



step-8 :-



We covered all the vertices without cycles and with minimum costs.

∴ The cost of Minimum Spanning Tree = 2+1+4+2+1+1+1

Edges = n-1 = 7. = 12

② Kruskal's Algorithm :-

Costs :-

(B, C) = 1

(E, F) = 1

(F, G) = 1

(G, H) = 1

(A, B) = 2

(A, C) = 2

(D, E) = 2

(E, G) = 2

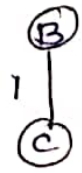
(F, H) = 2

(D, F) = 3

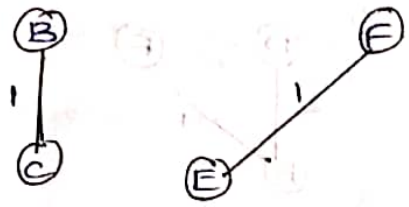
(B, D) = 4

(C, E) = 4

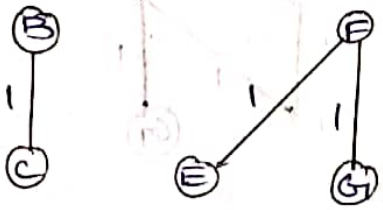
step-1 :-



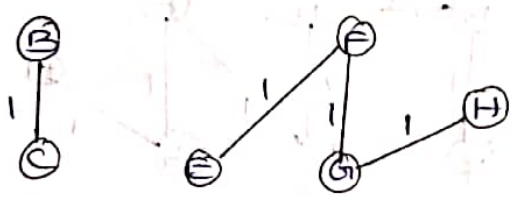
step-2 :-



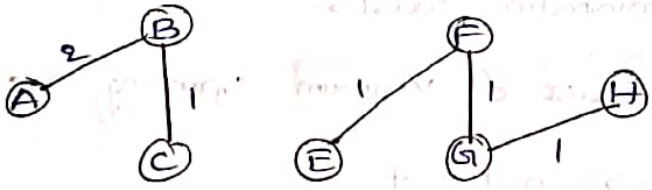
step-3 :-



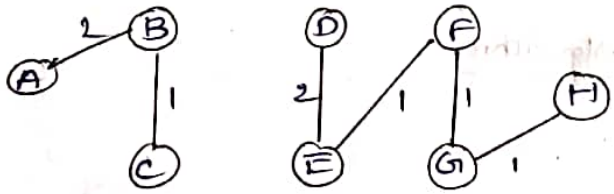
step-4 :-



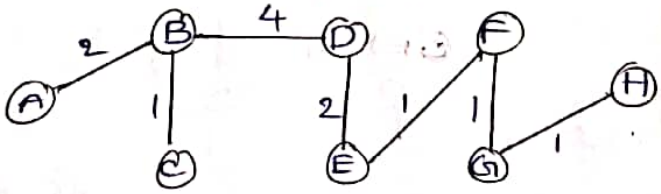
step-5 :-



step-6 :-



step-7 :-



Minimum cost = 12.

→ We created a MST and discarded all the edges which makes a cycle.

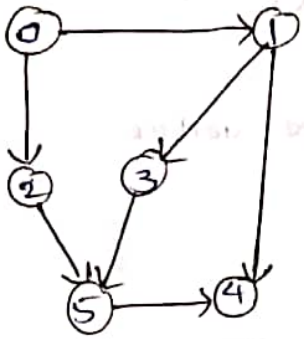
* Topological sorting:

→ It is used to sort the elements by using graphs

→ It is applied only on Directed Asyclic Graphs [DAG]

→ DAG :- A graph contains directions but not contains cycle is called "Directed Asyclic Graphs".

Ex:-

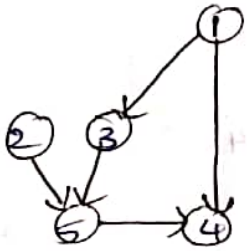


Step-1 :-

→ consider a vertex which contains In-degree as '0'.

O/p :- 0

Step-2 :- Remove the edges connected with vertex-'0'.



Step-3 :- Again select the '0'- indegree vertex.

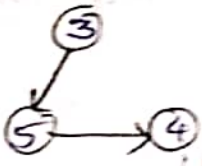
O/p :- 0, 1

Step-4 :- Remove '1' and its corresponding edges.

step-5 :- Select '0-Indegree' vertex

o/p :- 0, 1, 2

step-6 :- Remove '2' and its corresponding edges.



step-7 :- Select '0-degree' vertex

o/p :- 0, 1, 2, 3

step-8 :- Remove '3' and its corresponding edges.

step-9 :- Select '0-degree' vertex

o/p :- 0, 1, 2, 3, 5

step-10 :- 

step-11 :- 0, 1, 2, 3, 5, 4.

* Uses :-

→ It is used for project completion.

* Trie- Data structure :-

→ It is basically 'Non-linear Data structure'.

→ It is also known as 'Digital Tree' (or) 'Prefix Tree'

→ It is used to represent the strings in the form of Tree.

→ It contains tree fields 'Data field', 'pointer field' and 'Flag field'.

Ex:- a b c

a b c d e

a b c g h

a b c g l m

l m n

l m n o

l m n x y