Data Structures Notes

Data Structures & Algorithms

Data Structure :-

* Data structure is a particular way of storing and organizing data in a computer so that it can be used efficiently.

classification of data structure :

* There are two major classification of data structure :-
  1. Linear data structure (link list, stack, Queue)
  2. Non-linear data structure (Tree, graph)

Algorithms :-

* Sequence of steps to solve any given problem.
* Algorithms have a definite beginning and a definite end.
* A finite no of steps.

Characteristic of data structure :

Correctness : Data structure implementation should implement its interface correctly

Time Complexity :- Running time or the execution time of operations of data structure must be as small as possible.

Space complexity :

Need for data structure :

Searching data

Need to manage processor speed.

Execution time cases :-

Worst Case

Average Case

Best Case

Arrays

* An array is a finite collection of similar elements stored in adjacent memory locations
* By finite, we mean that there are specific no. of elements in an array.
* By similar, we mean that all the elements in an array are of same type.
e.g., any array may contain all integers or all characters but not both
* an array consisting of n number of elements is referenced using an index that varies from 0 to (n-1).
* The elements of A[n] containing n elements are denoted by A[0], A[1] --- A[n-1],
where 0 is the lower bound.
n-1 is the upper bound.
* In general, the lowest index of an array is called its lower bound and highest index of an array is called its upper bound.
* The number of elements in the array is called its range.

* Thus, an array is a set of pairs of an index & a value. For each index, there is a value associated with it.

a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]
34	1	5	-6	12	9	2

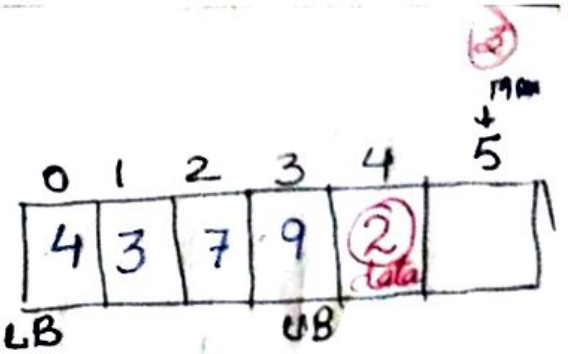
* An array is further categorized as a one-dimensional array & multi-dimensional array.
* A multi-dimensional array can be a 2-D array, 3-D array, 4-D array etc. Whether an array is a 1-D or 2-D can be judged by the syntax used to declare the array.

A[5] :- A 1-D array holding 5 elements.
A[2][5] :- A 2-D array with 2 rows & 5 columns holding 10 elements.
A[2][5][3] :- A 3-D array with two 2-D array each of which is having 5 rows and 3 columns., thus holding totally 30 elements.

Operations on Array :-

* Insertion
* Deletion
* traversal
* search

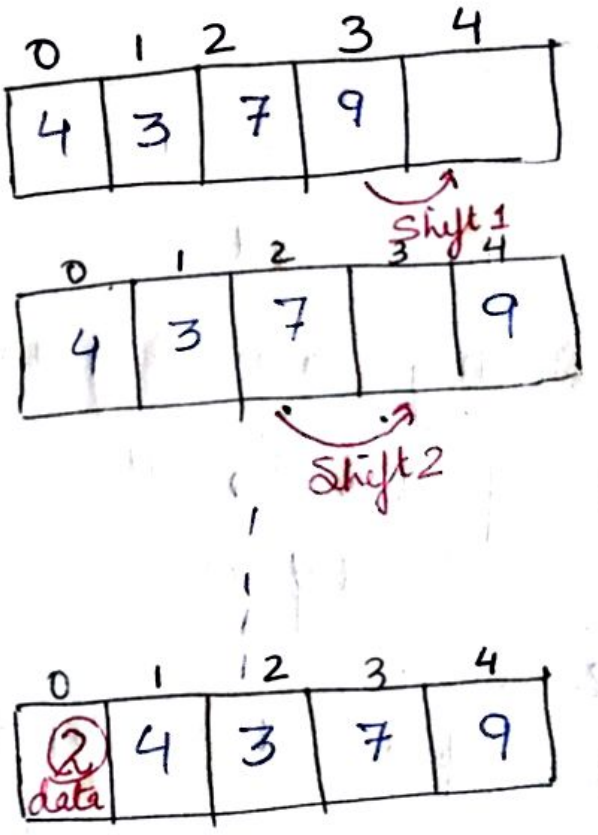
Insertion at End :



Let A be an array of size N.

1. If UB = ~~MAX~~ N-1
   then "overflow"
2. Read data or value to be inserted
3. Set UB = UB + 1
4. A[UB] = data
5. Stop or Exit

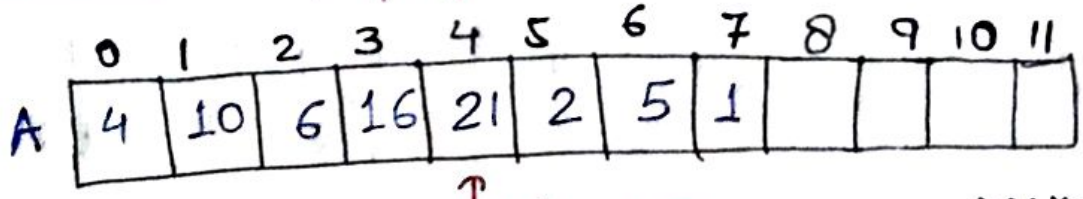
Insertion at Beginning :-



Let A be an array of size N.

1. If UB = ~~MAX-1~~ N-1
   then "overflow"
2. Else Read data
3. ~~LB~~ K = UB
4. Repeat step (5)
   while K ≥ LB
5. A[K+1] = A[K]
6. K = K-1
7. A[LB] = data
8. Stop

Insertion at specific location :-



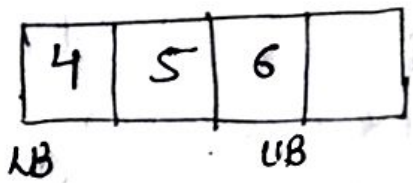
MAX = 11
UB = 7
loc = 4
data = 100

7 to 4 Shifted, 8 to 5
UB to loc (UB+1) to (loc+1)

1. Initialize a counter I = UB
2. Repeat step 4 while (I >= loc)
3. A[I+1] = A[I]
4. I = I - 1.
5. UB = UB+1, A[loc] = data
6. Exit

Deleting from beginning

1. If UB = -1, then "Underflow"



2. K = LB.
3. Repeat till Step 5 while K < UB
4. A[K] = A[K+1]
5. K = K+1
6. A[UB] = NULL
7. UB = UB - 1
8. Exit

Deletion of a particular element :-

A[0]	A[1]	A[2]	A[3]	A[4]	
4	6	10	2	12	

LB UB

1. If UB = -1, "Underflow"
2. Read data
3. K = LB
4. Repeat step (5) while A[K] ≠ data
5. K = K + 1
6. Repeat step (8) while K < UB
7. A[K] = A[K+1]
8. K = K+1

Representation of 2-D arrays in memory :-

* A 2-D array is a collection of elements placed in m rows & n columns.
* The syntax used to declare a 2-D array includes two subscripts, of which one specifies the no. of rows & the other specifies the no. of columns of an array.
* e.g; A[3][4] is a 2-D array containing 3 rows & 4 columns. & A[0][2] is an element placed at 0th row, & 2nd column.

* 2-D array is also called a matrix

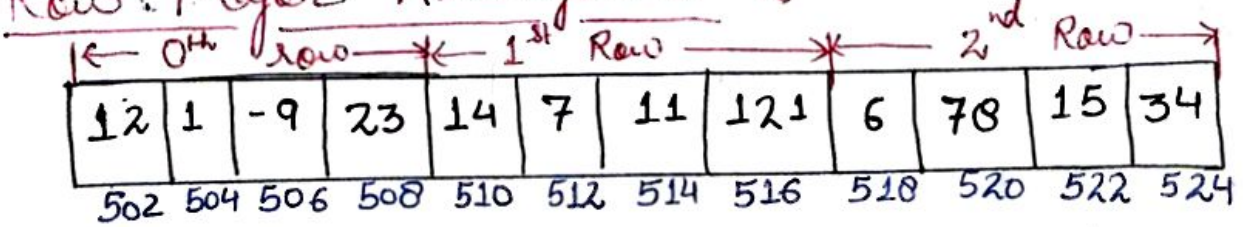
	0	1	2	3
0	12	1	-9	23
1	14	7	11	121
2	6	78	15	34

Row Major & column Major Arrangement:

* Row & columns of a matrix are only a matter of imagination. when a matrix gets stored in memory all elements of it are stored linearly.
* Since computer's memory can only be viewed as consecutive units of memory.
* This leads to two possible arrangements
  - → Row-Major arrangements
  - → column-Major arrangements

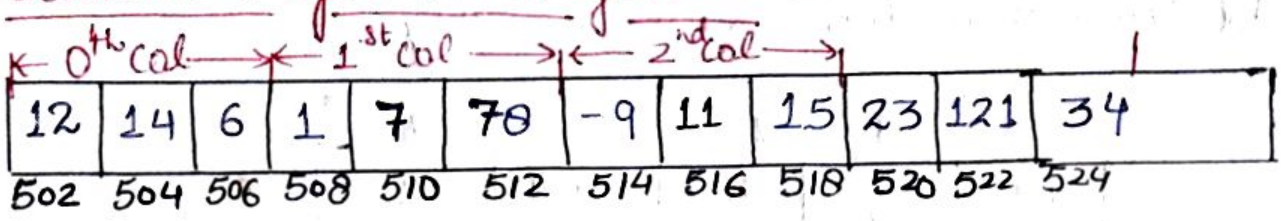
```
int A[3][4] = {
                {12, 1, -9, 23},
                {14, 7, 11, 121},
                {6, 78, 15, 34},
              };
```

Row-Major Arrangement :-



+ UB = UB - 1

Column-Major Arrangement :-



* Since the array elements are stored in adjacent memory locations, we can access any element of the array once we know the base address of the array & no. of rows & columns present in the array.
* e.g., if base address of the array is 502 & we wish to refer the element 121, then the calculation involved as follows:-

Row-Major Arrangement :-

* Element 121 is present at A[1][3].

Hence location of 121 would be

$= 502 + 1 * 4 + 3 = 516$

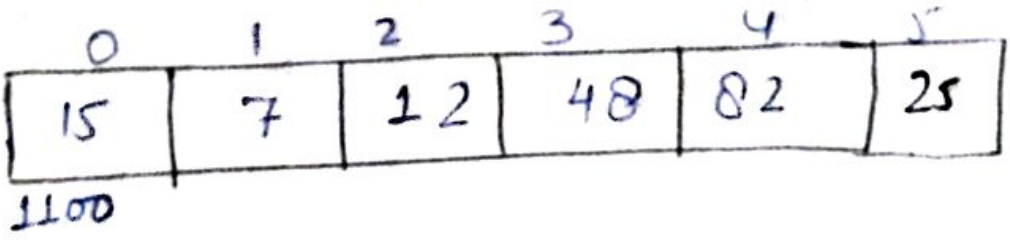
In general, for an array A[m][n] the address of element A[i][j] would be

Baseaddress + i * n + j

Illy. for column major arrangement

Base address + j * m + i

Address Calculation in 1-D array :-



address of $A[i] = B + W * (i - LB)$

B → Base address of array
W → Storage size of one element
I → Subscript of element whose address is to be found
LB → Lower Bound of subscript, if not specified assume 0 (zero)

Question :- Given base address of an array B[1300 .... 1900] as 1020 & size of each element is 2 bytes. Find address of B[1700]

the given values are $B = 1020$, $W = 2$, $LB = 1300$, $I = 1700$

Ans :- 1020

Address of $A[i][j] = B + W * [N * (I - Lr) + (J - Lc)]$
{Row Major order}

Address of $A[i][j] = B + W * [(I - Lr) + M * (J - Lc)]$
{column Major order}

where,

B → Base Address

I → Row Subscript of element whose add. is to be found.

J → Column " " " " " " "

W → Storage Size of one element

Lr → Lower limit / start row index, if not given assume zero.

Lc → Lower limit / start column index, if not given assume zero

M → No. of rows of given matrix

N → No. of columns " " "

Q.11:-

Note:- Usually no. of rows & columns of a matrix are given like A[20][30] etc. but if not given as A[Lr ---- Ur][Lc ... Uc]. then the no. of rows & columns are calculated using :

No. of rows (M) = (Ur - Lr) + 1

No. of cols (N) = (Uc - Lc) + 1

Examples :- X[-15 --- 10, 15 --- 40] requires one byte of storage. If beginning location is 1500, Determine the location of X[15][20].

Solution :- M = [10 - (-15)] + 1 = 26

N = [40 - 15] + 1 = 26

(i) Column Major Calculation :-

B = 1500, w = 1, I = 15, J = 20, Lr = -15, Lc = 15, M = 26

Address of A[I][J] = B + w * [(I - Lr) + M * (J - Lc)]

= 1500 + 1 * [(15 - (-15)) + 26 * (20 - 15)]

= 1500 + 1 * [30 + 26 * 5]

= 1500 + 1 * [160]

= 1660

Stacks :-

* The linear data structures as array and a linked list allow us to insert & delete an element at any place in the list.
* However, sometimes it is required to permit the addition or deletion of elements only at one end that is either beginning or end.
* Stack and Queues are two types of data structures in which addition or deletion of an element is done at end, rather than middle.
* A stack is a data structure in which addition of a new element or deletion of an existing element always takes place at the same end.
* This end is often known as top of stack.
* e.g., A stack of plates in cafeteria in where every new plate is added to the stack is added at top.
* similarly, every new plate taken off the stack is also from top.

→ When an item is added to the stack, the operation is called push and when an item is removed from the stack, the operation is called pop.

→ Stack is also called as last-in-first-out (LIFO) list.

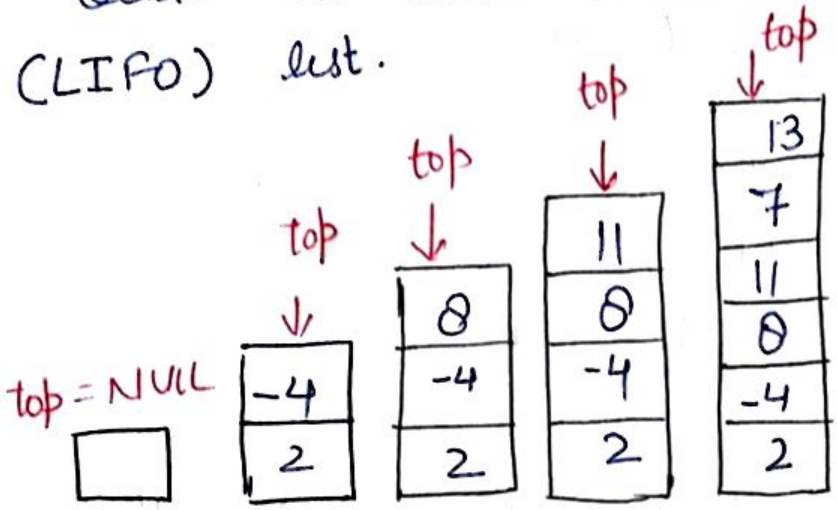


fig :- Representation of Stack after inserting elements

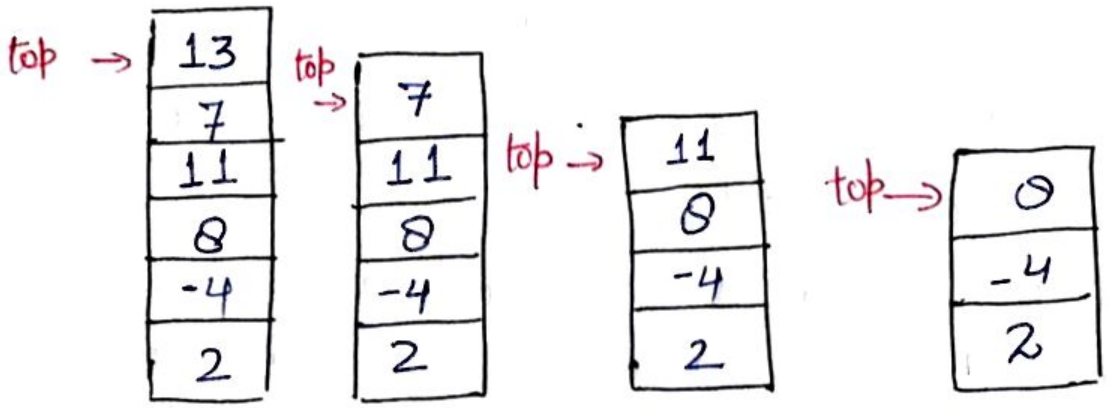


fig :- Representation of Stack after deletion

Operations on Stack :-

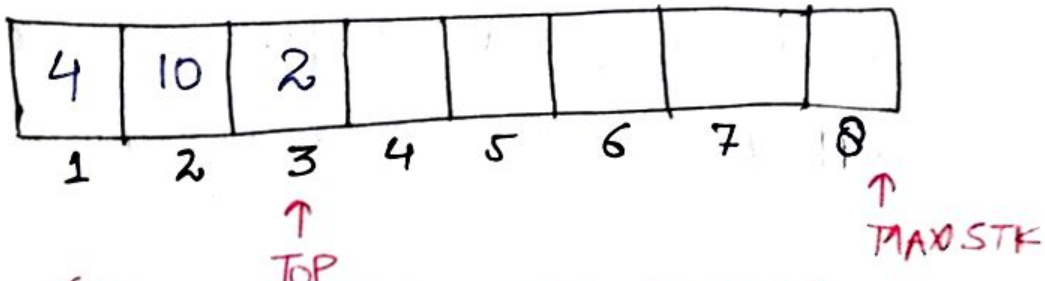
Stack is generally implemented with two basic operations

* Push
* Pop

Representation of Stack as Array :-

* Stack contain an ordered collection of elements. An array is used to store ordered list of elements., Hence it would be very easy to manage a stack if we represent it using an array.

* Each of our Stacks will be maintained by
→ a linear array STACK → a pointer variable TOP, which contains the location of the top element of the stack.
→ A variable MAXSTK which gives the maximum no. of elements that can be held by the stack.
→ The condition TOP = 0 or TOP = NULL will indicate that the stack is empty.



PUSH (STACK, TOP, MAXSTK, ITEM)
(This algo. pushes an item onto a stack)

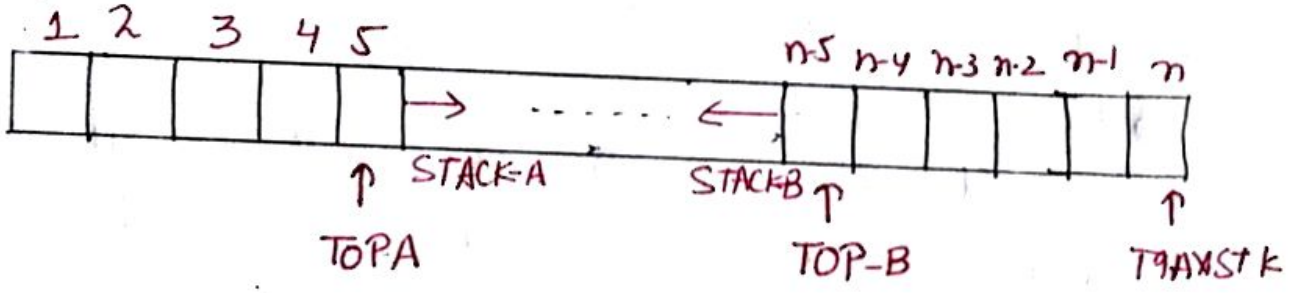
1. If TOP = MAXSTK then Print Overflow & Return
   [stack already filled]
2. Set TOP = TOP + 1 [Increase Top by 1]
3. Set STACK [TOP] = ITEM [Insert item]
4. Return.

POP (STACK, TOP, ITEM)

This algo. deletes the top element of STACK & assigns it to the variable ITEM.

1. If TOP = 0 then Print UNDERFLOW & Return [Stack has an item to be removed?]
2. Set ITEM = STACK [TOP]. [Assigns TOP element to ITEM]
3. Set TOP = TOP - 1 [decreases TOP by 1]
4. Return

Multiple Stack Implementation using Single Array :-



Algo. Push-A (STACK, TOPA, TOPB, Item, MAXSTK)

1. If (TOPA + 1 = TOPB OR TOPB = = 1 OR TOPA = MAXSTK then Print "Overflow" & Exit.
2. TOPA = TOPA + 1
3. STACK [TOPA] = Item
4. Exit

Algo PushB (STACK, TOPA, TOPB, item, MAXSTK)

1. if (TOPA+1 = TOPB OR TopA = MAXSTK OR TOPB = 1)
   Print overflow & exit
2. if (TOPB = 0) then TOPB = MAXSTK
3. Else TOPB = TOPB - 1
4. Set STACK[TOPB] = Item
5. Exit

Algo POPA (STACK, TOPA, Item)

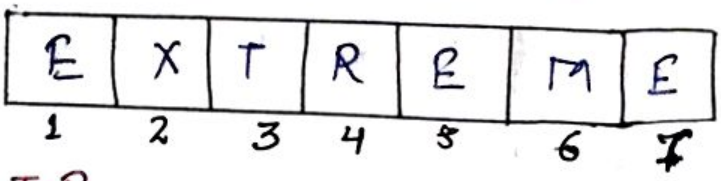
1. If (TOPA = 0) then Print "Underflow" Exit.
2. Else item = STACK[TOPA]
3. TopA = TOPA - 1
4. Exit

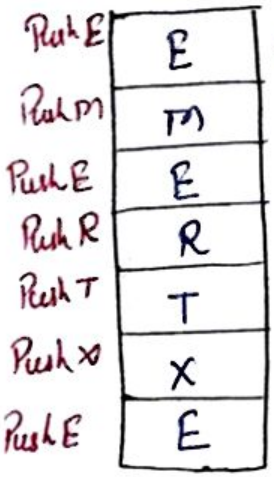
Algo POPB (STACK, TOPB, Item, MAXSTK)

1. If TOPB = 0 Print "Underflow" & Exit
2. Else { Item = STACK[TOPB]
3. if (TOPB = MAXSTK) then TOPB = 0 }
4. Else TOPB = TOPB + 1
5. Exit

Stack Applications

① Reversing a String





Algo :- 1. Traverse the list and push all its values onto a stack.

2. Traverse the list from TOP element & POP a value & connect them in reverse order.

② Factorial Calculation

Recursion :- When a function calls itself.

```
main(){
    fun();
}
fun(){
   fun()
}
```

direct recursion

```
main() {
    a();
}
a() {
   b();
}
b(){
   a();
}
```

Indirect recursion

e.g. factorial of a number

$4! = 4 * 3!$ } Recursive condition $[n * (n-1)!]$

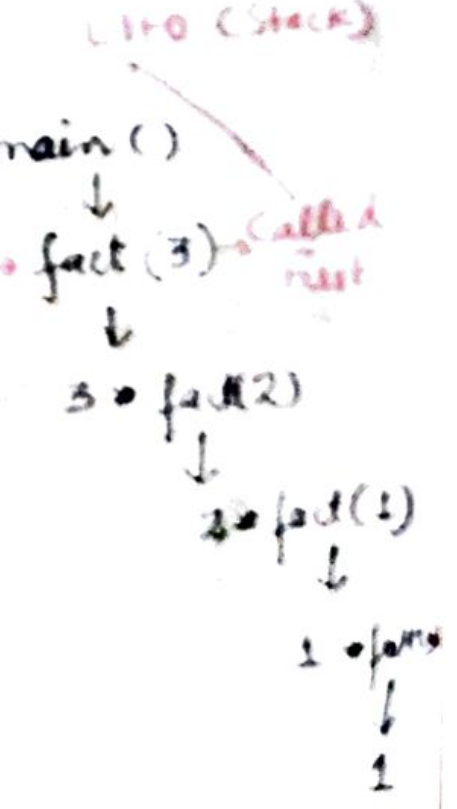
$3! * 2!$

$2 * 1!$

$1 * 0!$

1 } Base Condition / Stop condition

```
# include <stdio.h>
int fact (int n) {
    if (n == 0)
        return 1;
    else
        return (n * fact (n-1));
}

void main()
{
    int num, result;
    printf("Enter a no");
    scanf("%d", &num);
    result = fact(num);
    printf("%d", result);
}
```



Polish Notation

* The process of writing the operators of an expression either before their operands or after them is called Polish Notation.

* The fundamental property is that the order in which the operations are to be performed is completely determined by the positions of the operators and operands in the expressions.

* One never needs parenthesis when writing expression in Polish Notation. These are classified in three categories :-
  (i) Infix
  (ii) Prefix
  (iii) Postfix

* Infix :- When operators exist between two operands, the expression is called Infix.

* Prefix :- When operators are written before their operands, the expression is called prefix polish notation.

* Postfix :- When the operators come after their operands the resulting expression is called the reverse polish notation or Postfix notation.

e.g., $z + (y * x - (w / v \uparrow u) * t) * s$

By Inspection & Hand :-

$\Rightarrow z + (y * x - (w / v\ u \uparrow) * t) * s$ ①

$\Rightarrow z + (y * x - w\ v\ u \uparrow / * t) * s$ ②

$\Rightarrow z + (y\ x * - w\ v\ u \uparrow / t *) * s$ ③

$\Rightarrow z + y\ x * w\ v\ u \uparrow / t * - * s$ ④

$\Rightarrow z + y\ x * w\ v\ u \uparrow / t * - s *$ ⑤

$\Rightarrow z\ y\ x * w\ v\ u \uparrow / t * - s * +$ ⑥

Stack Implementation of Infix to Postfix

Label No.	Symbol Scanned	Stack	Expression
1.	z	(	z
2.	+	(+	z
3.	(	(+ (	z
4	y	(+ (	z y
5.	*	(+ (*	z y
6.	x	(+ (*	z y x
7.	−	(+ (−	z y x *
8.	(	(+ (− (	z y x *
9.	ω	(+ (− (	z y x * ω
10.	/	(+ (− (/	z y x * ω
11.	v	(+ (− (/	z y x * ω v
12.	↑	(+ (− (/ ↑	z y x * ω v
13.	u	(+ (− (/ ↑	z y x * ω v u
14.	)	(+ (− [redacted]	z y x * ω v u ↑ /
15.	*	(+ (− [redacted] *	z y x * ω v u ↑ /
16.	t	(+ (− [redacted] *	z y x * ω v u ↑ / t
17	)	(+ [redacted]	z y x * ω v u ↑ / t * −
18.	*	(+ *	z y x * ω v u ↑ / t * −
19.	s	(+ *	z y x * ω v u ↑ / t * − s
20.	)		z y x * ω v u ↑ / t * − s * +

Postfix Expression

Algo. for converting Infix Expression To Postfix

Postfix (Q, P) :-

Q is an arithmetic expression within Infix

P is an equivalent postfix

1. Push "(" onto stack & add ")" to the end of Q.
2. Scan Q from L to R & repeat till (6) for each element of Q until the stack is empty.
3. If an operand is encountered, push it onto P stack.
4. If a left paranthesis is encountered, push it onto Stack.
5. If an operator is encountered then
   (a) Add it to stack.
   (b) Repeatedly Pop from Stack & add P each operator which has the same precedence as or higher than operator.
6. If the right parenthesis is encountered :-
   (a) Repeatedly Pop from Stack & add it to P each operator until a left parenthesis is encountered.
   (b) Remove left parenthesis.
7. Exit.

Evaluating the Postfix Expression

Example :- P :- 5, 6, 2, +, *, 12, 4, /, -,)
↓ Sentenel

Symbol Scanned	Stack
5	5
6	5, 6
2	5, 6, 2
+	5, 8
*	40
12	40, 12
4	40, 12, 4
/	40, 3
-	37

Algorithm :-

→ This algo. finds the value of an arithmetic expression P written in postfix notation
→ This algo. uses Stack to hold operands to evaluate P.
→ (1) Add a right parenthesis ")" at the end of P.
(2) Scan P from L to R & repeat steps 3 & 4. ~~(2) If an operand is encountered until ")"~~ for each element of P until ")" is encountered.
(3) If an operand is encountered, put it on Stack.
(4) If an operator is encountered, then
(A) Remove the top two elements of Stack, where A is the top element & B is the next-to-top element.

(B) Evaluate B ⊗ A.

(C) Place the result of (B) on the Stack.

[End of step ② loop]

⑤ Set value equal to the top element on Stack.

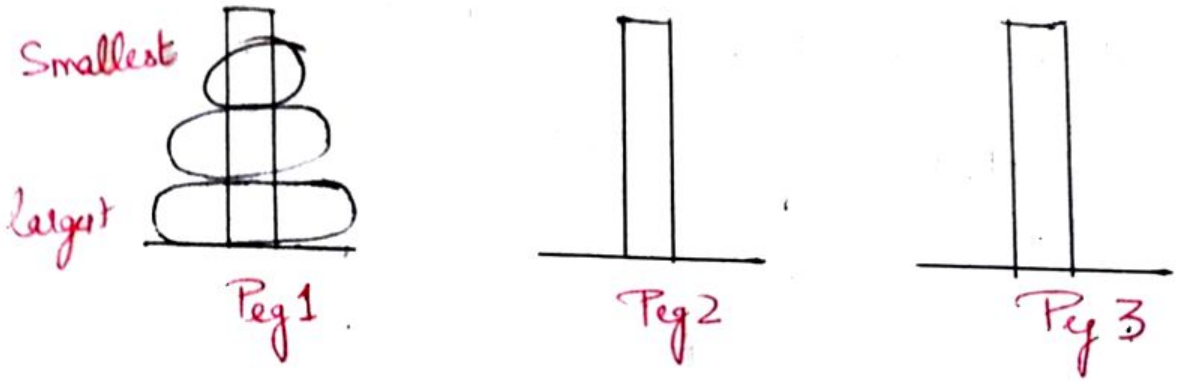
⑥ Exit.

Example ② :- 12, 7, 3, −, /, 2, 1, 5, +, *, +, } Sentinel

Symbol	Stack
12	12
7	12, 7
3	12, 7, 3
−	12, 4
/	3
2	3, 2
1	3, 2, 1
5	3, 2, 1, 5
+	3, 2, 6
*	3, 12
+	15

Tower Of Hanoi :-

* Tower of Hanoi is a mathematical puzzle which consists of three towers (pegs) and more than one rings is as depicted -



* These Rings are of different sizes & stacked upon in an ascending order i.e., smaller one is placed over the larger one.

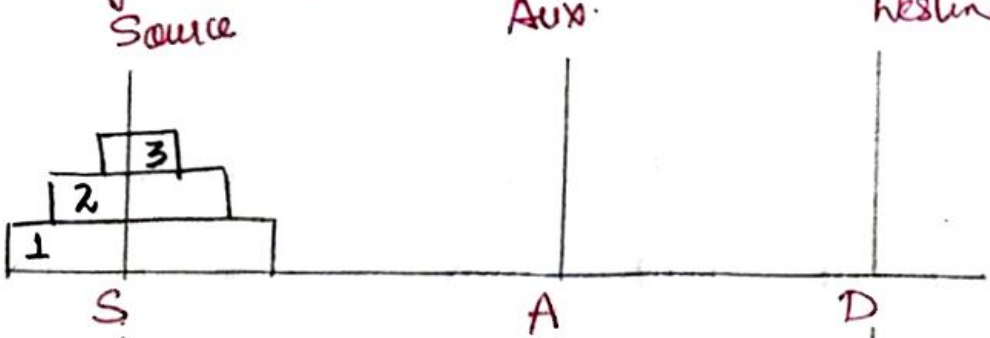
Note :- There are other variations of the puzzle where the no. of disks increase, but the tower count remain the same.

Rules :- The mission is to move all the disks to some tower without violating the sequence of arrangement. A few rules are:-

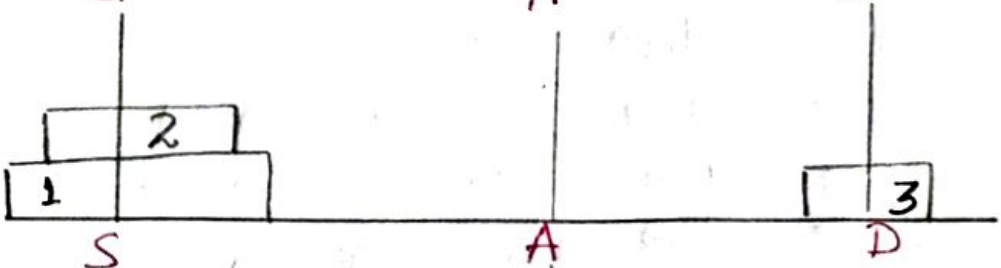
→ Only one disk can be moved among the towers at any given time.
→ Only the top disk can be removed.
→ No large disk can be placed over the small disk.

Tower of Hanoi puzzle with 3 disks

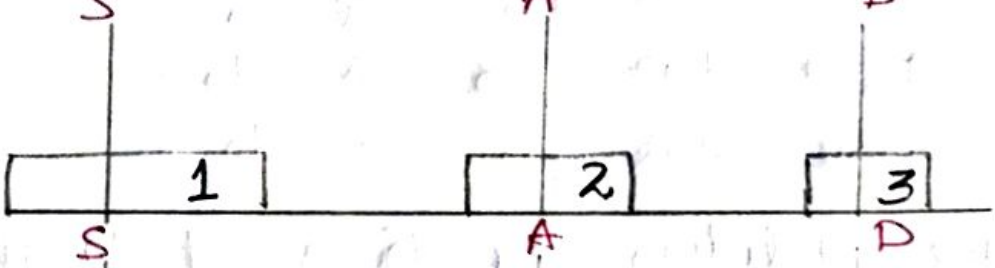
Problem



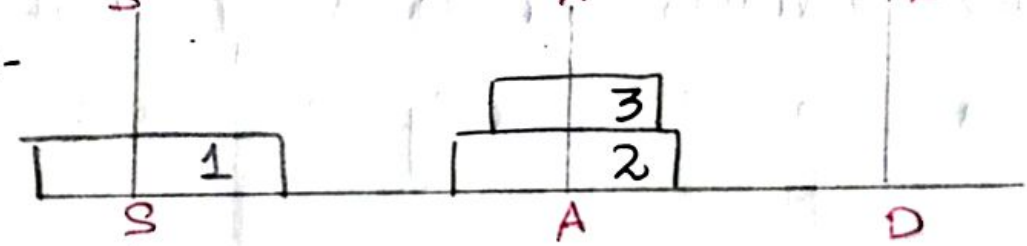
Step 1 :-



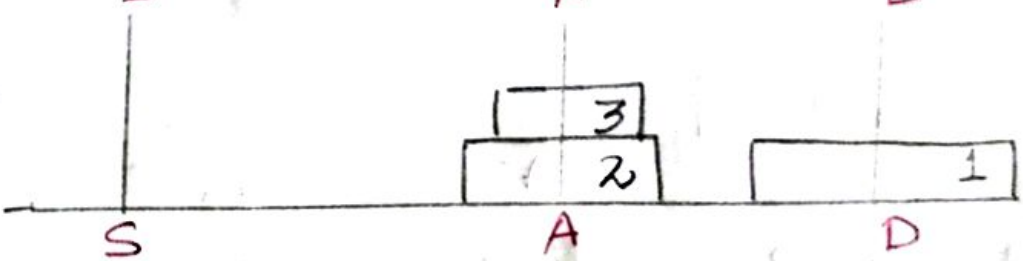
Step 2 :-



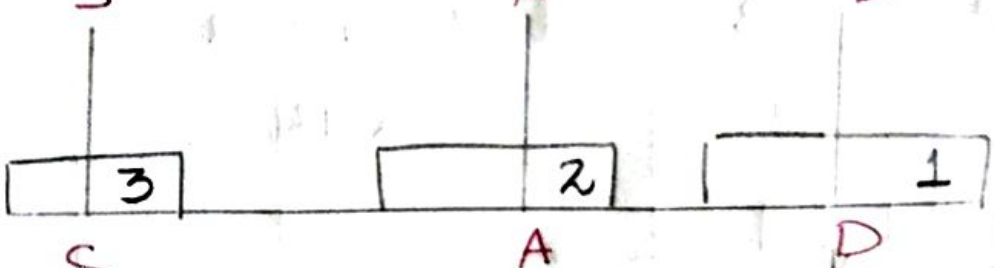
Step 3 :-



Step 4 :

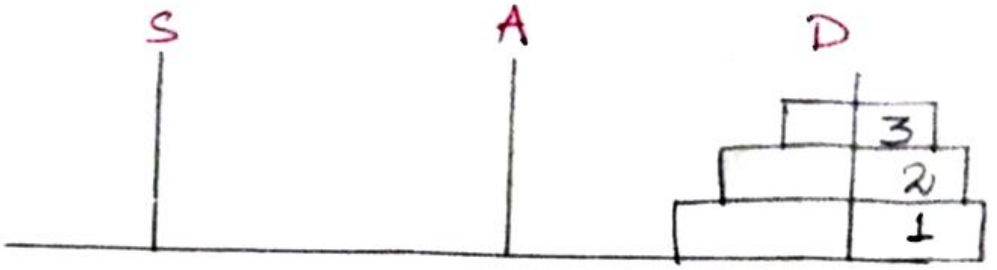


Step 5 :-



Step 6 :

Step 7 :-

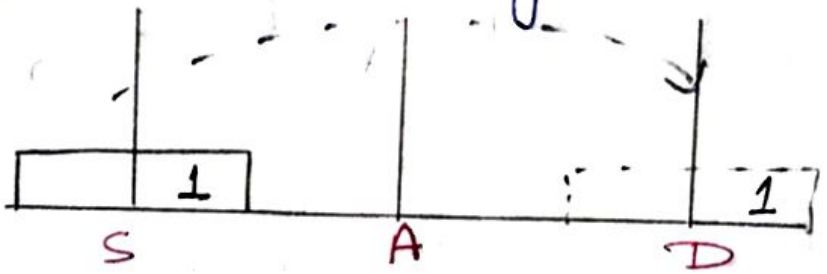


Thus for n = 3 :- the seven moves will be;

→ Move disk from S to D
Move disk from S to A
Move disk from D to A
Move disk from S to D
Move disk from A to S
Move disk from A to D
Move disk from S to D

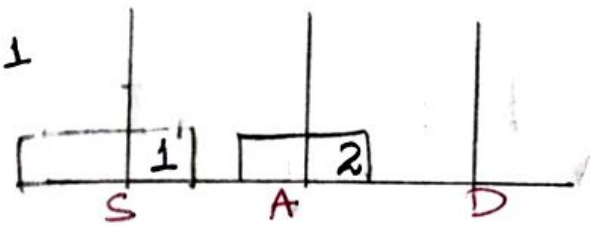
Recursive solution for Tower of Hanoi Problem :-

* For n = 1, only one move

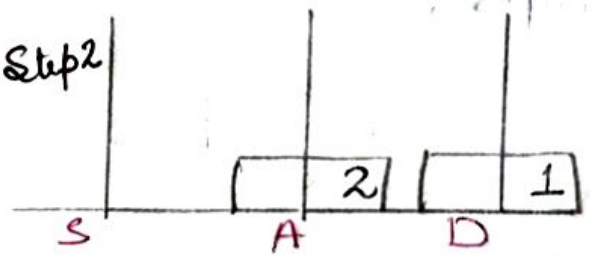


* For n = 2, three moves

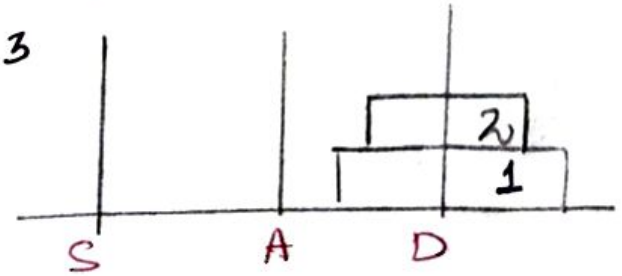
Step 1



Step 2



Step 3



* Rather than finding a separate solution for each value of n, we'll use the technique of recursion to develop a general solution.

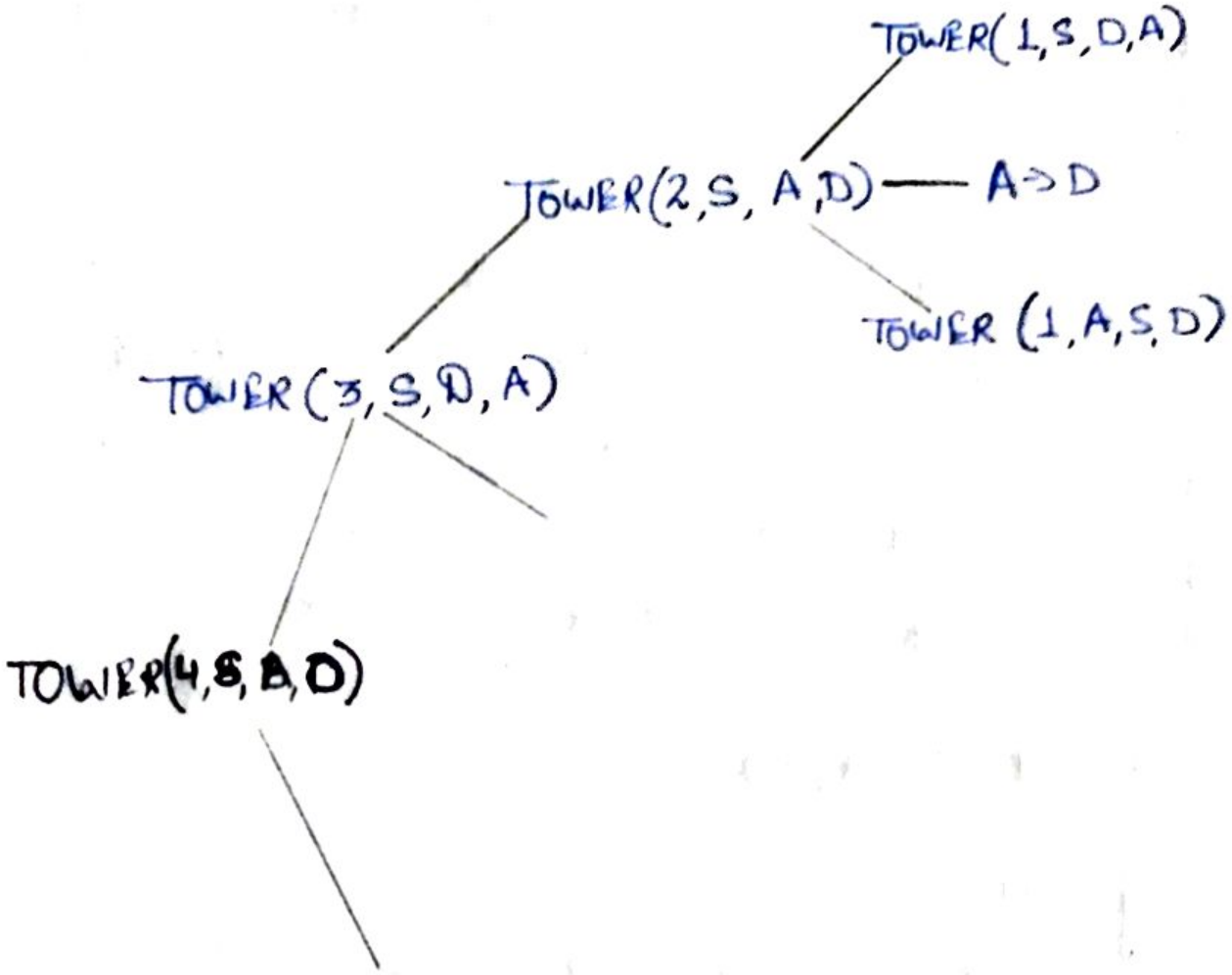
* First we observe that the solution to the Towers of Hanoi problem for n>1 disks may be reduced to the following subproblems:-

1) move top (n-1) disks from S to A.
2) move top disk from S to D.
3) move top (n-1) disks from A to D.

TOWER (N, S, A, D)

1) if N = 1 then
   print ("move disk from S to D").
2) Else TOWER (N-1, S, D, A)
3) print ("move disk from S to D")
4) TOWER (N-1, A, S, D)

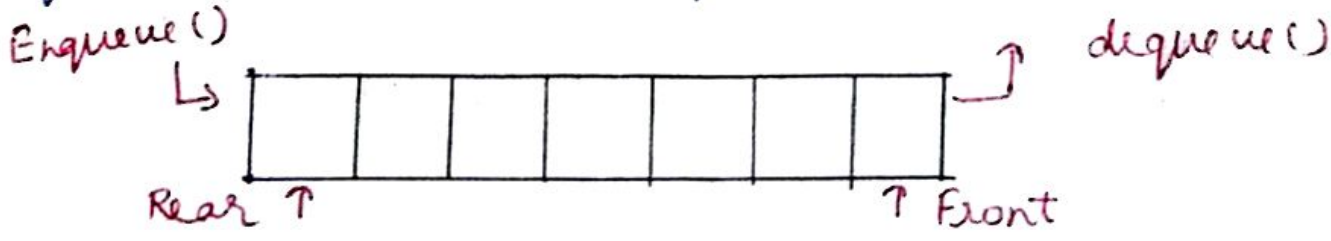
TOWER (4, S, A, D)



UNIT- II

Queue :-

* Queue is a linear data structure where the first element is inserted from one end called REAR and deleted from the other end called as FRONT.
* Front points to the beginning of the queue and Rear points to the end of the queue.
* Queue follows the FIFO (First-In-First Out) structure.
* According to its FIFO structure, element inserted first will also be removed first.
* This contrasts with stacks, which are last-in-first-out (LIFO).
* The process to add an element into queue is called Enqueue & the process of removal of an element from the queue is called Dequeue.



* A real-world example of queue can be a single-lane one-way road, where the vehicle enters first, exits first.

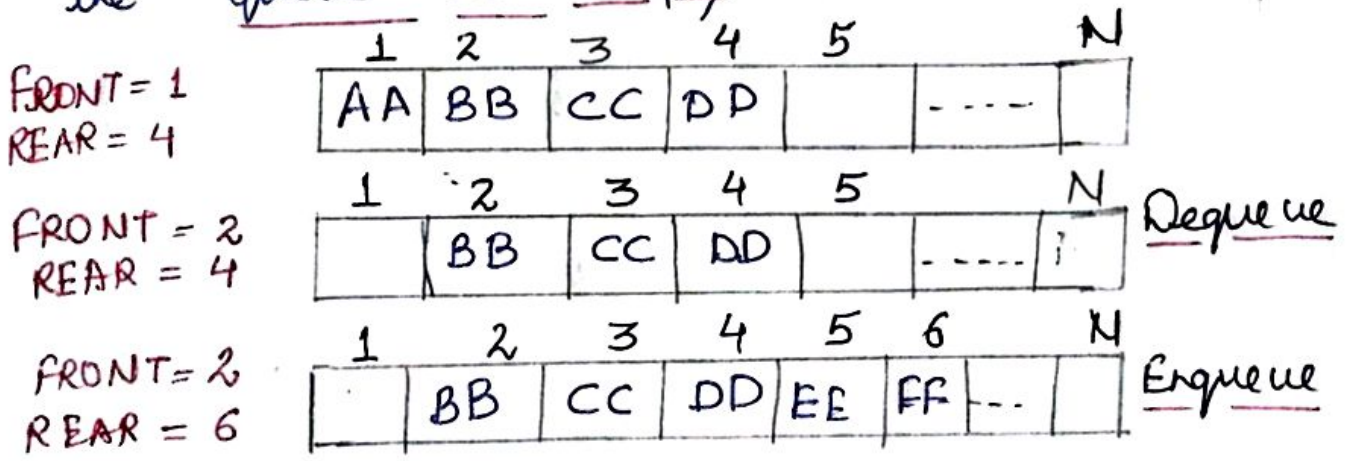
Representation of Queue :-

* Queues may be represented in the computer by means of one-way lists or linear arrays.
* Unless otherwise stated or implied, each of our queues will be maintained by a linear array QUEUE and two pointer variables :-

FRONT :- containing the location of the front element of the queue.

REAR :- containing the location of the rear element of the queue.

* The condition FRONT = NULL, indicates that the queue is empty.



* The above figure shows the way the array QUEUE will be stored in memory with N elements.
* Observe that whenever an element is deleted from the queue, the value of FRONT is increased by 1 i.e.,

FRONT = FRONT + 1

* Whenever an element is added to the queue, the value of REAR is increased by 1 i.e.,

REAR = REAR + 1

* this means that after N insertions, the rear element of the queue will occupy QUEUE[N]. This occurs even though the queue itself may not contain many elements.

Note :- Suppose we want to insert an element ITEM into a queue at the time queue does occupy last part of the array i.e., REAR = N.

* One way to do this is to simply move the entire queue to the beginning of the array, changing FRONT & REAR accordingly.
* The above procedure is very expensive. Thus we adopt the concept of circular queue.

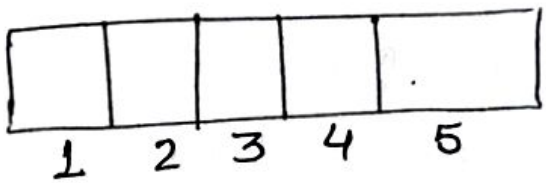
i.e., when REAR = N then reset REAR = 1
then assign QUEUE[REAR] = ITEM
{for inserting ITEM}

IIly if FRONT = N, then reset FRONT = 1

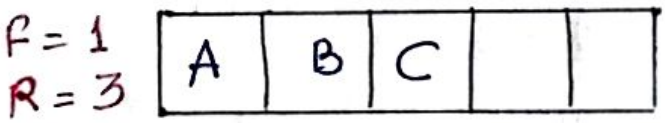
Suppose that our queue contains only one element i.e; FRONT = REAR ≠ NULL
& suppose that the element is deleted. Then we assign,
FRONT = NULL & REAR = NULL
to indicate the empty list.

Example :-

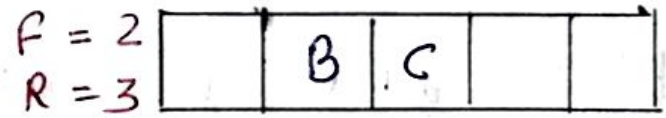
(a) Initially empty



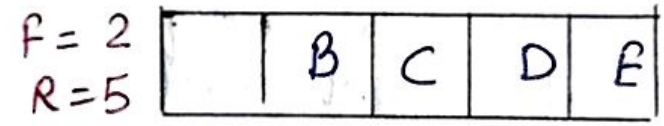
(b) A, B & C inserted



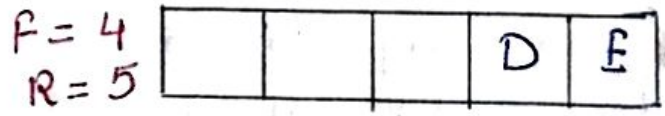
(c) A deleted



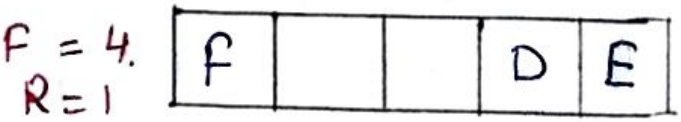
(d) D & E inserted



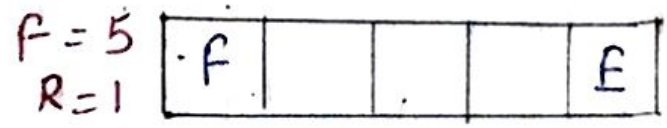
(e) B & C deleted



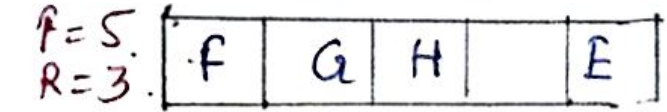
(f) F inserted



(g) D deleted



(h) G & H inserted



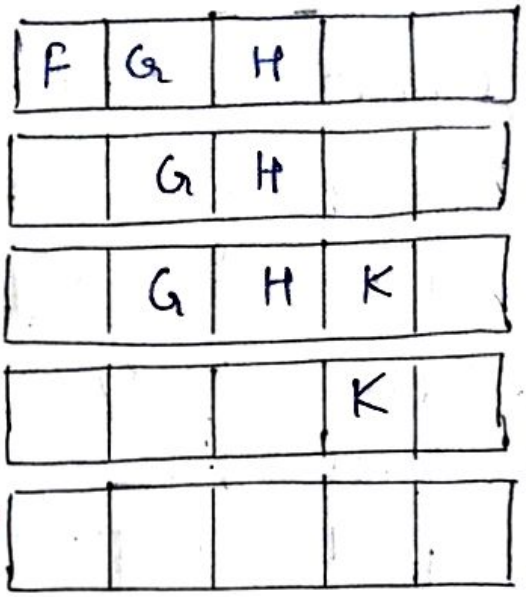
(i) E deleted — F = 1, R = 3

(j) f deleted — F = 2, R = 3

(k) K inserted — F = 2, R = 4

(l) G & H deleted — F = 4, R = 4

(m) K deleted, QUEUE empty — F = 0, R = 0



Algorithm QINSERT (QUEUE, N, FRONT, REAR, ITEM)

This procedure inserts an element ITEM into queue.

1. [Queue already filled]
   if FRONT = 1 and REAR = N or if FRONT = REAR + 1
   then print "Overflow" & Exit.

2. [Find new value of REAR]
   if FRONT = NULL then [Queue initially empty]
   Set FRONT = 1 and REAR = 1

   else if REAR = N then
   set REAR = 1

   else
   Set REAR = REAR + 1
   [End of if]

3. Set QUEUE [REAR] = ITEM [Inserts new element]

4. Return

Algorithm QDELETE(QUEUE, N, FRONT, REAR, ITEM)

This procedure deletes an element from a queue & assigns it to variable ITEM

1. [Queue already empty]
if FRONT = NULL. then
print "overflow" & Exit

2. Set ITEM = QUEUE[FRONT]

3. if FRONT = REAR then [Queue has only one element]
Set FRONT = NULL & REAR = NULL

else if FRONT = N then
set FRONT = 1

else
set FRONT = FRONT + 1

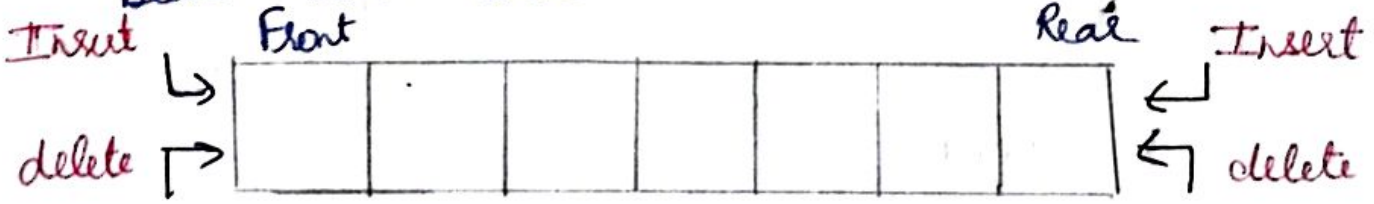
4. Return

Deque :-

* A deque, also known as double-ended queue is an ordered collection of items similar to the queue.

* We'll maintain deque by a circular array DEQUE with pointers ~~left~~ LEFT and RIGHT, which points to the two ends of the deque.

* Double ended queue is a more generalized form of queue data structure which allows insertion & removal of elements from both the ends.

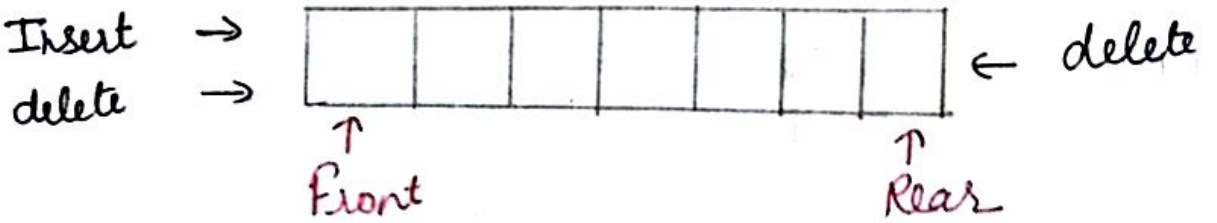


* Double Ended Queue can be represented in. Two ways :-

1. Input Restricted Double Ended Queue
2. Output " " " "

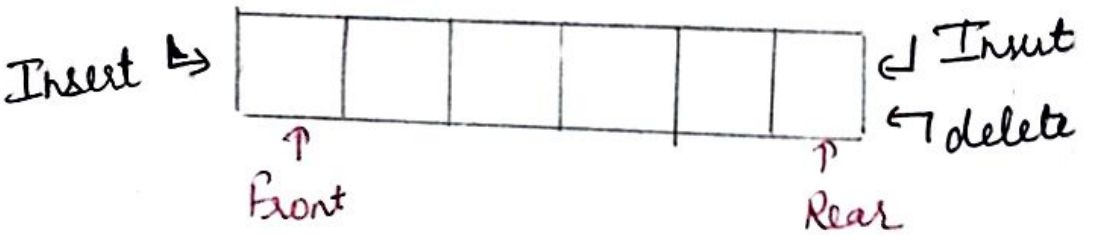
Input Restricted Double Ended Queue :-

→ The insertion operation is performed at one the ends and deletion operation is performed at both the ends.



Output Restricted Double Ended Queue :-

→ The deletion operation is performed at only one end and insertion operation is performed at both the ends.



Priority Queue :-

* A priority queue is a collection of elements such that each element has been assigned a priority.
* The order in which elements are deleted and processed comes from the following rule.

1) An element of higher priority is processed before any element of lower priority.

2) Two elements with the same priority are processed according to the order in which they are added to the queue.

* There are various ways of maintaining a priority queue in memory.

  ↳ One-way list

  ↳ Multiple Queues

One-way list representation of a Priority Queue

One way to maintain a priority queue in memory is by means of a one-way list, as follows:-

(a) Each node in the list will contain three items of information:-

* an information field INFO
* a priority no. PRN
* a link no LINK

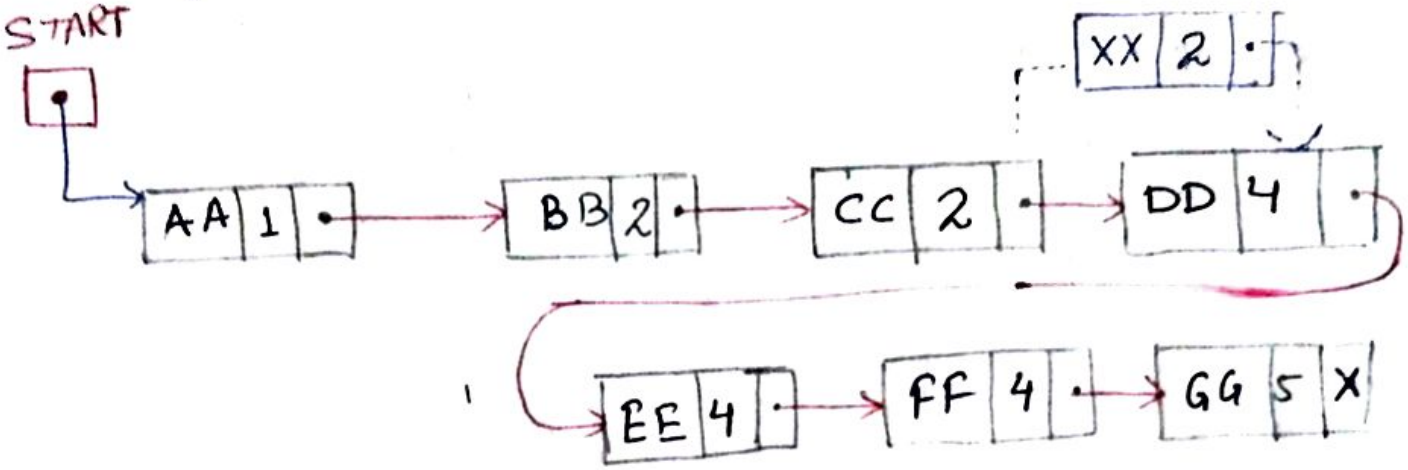
(b) A node X precedes a node Y in the list

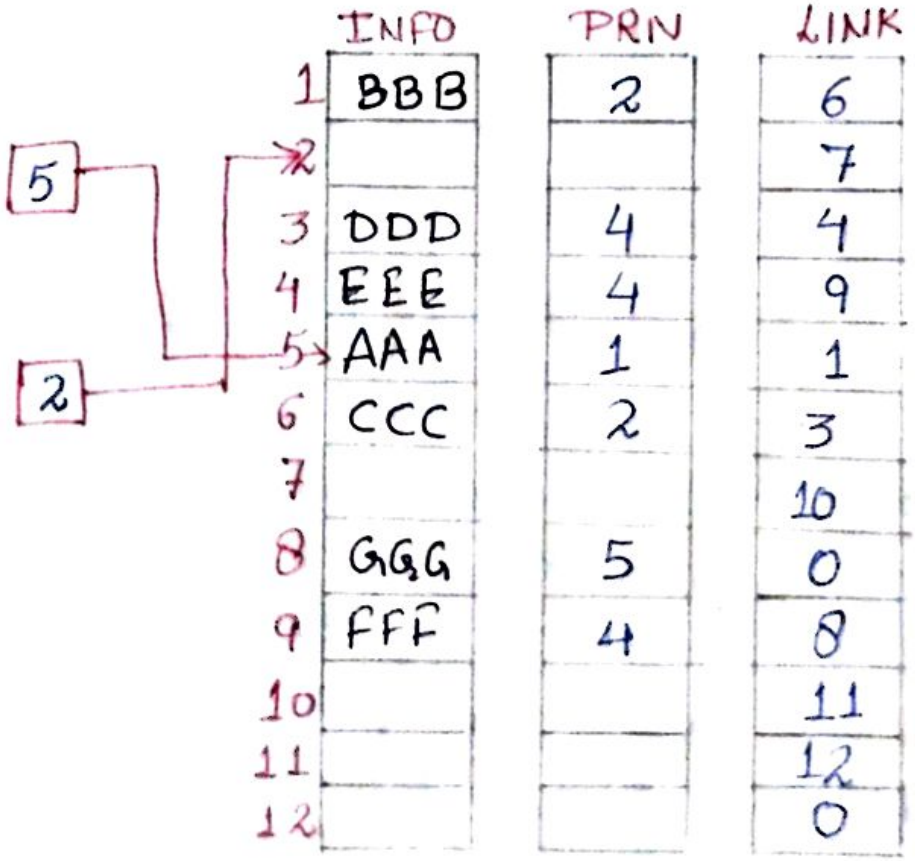
(i) When X has higher priority than Y

or (ii) When both have the same priority but X was added to the list before Y.

{Note:- Thus the order in the one-way list corresponds to the order of the priority queue.}

* Priority no. will operate the usual way, the lower the priority no., the higher the priority.





Q. Consider the above priority queue, suppose an item XXX with priority no. 2 is to be inserted into the queue:

* We traverse the list, comparing priority nos. Observe that DDD is the first element in the list whose priority exceeds that of XXX.
* Hence XXX is inserted in the list in front of DDD.

Note :- Observe that XXX comes after BBB & CCC, which have the same priority as XXX. Suppose now, an element is to be deleted from the queue, it will be AAA, then BBB → then CCC then XXX & soon.

Array Representation of a Priority Queue :-

* Another way to maintain a priority queue in memory is to use a separate queue for each level of priority (or for each priority no.)
* Each such queue will appear in its own circular array & must have its own pair of pointers, FRONT & REAR.
* In fact, each queue is allocated the same amount of space, a 2-D array QUEUE can be used instead of the linear arrays.

* The following figure indicates this representation for the priority queue. Observe that FRONT[K] and REAR[K] contain, respectively, the front & rear elements of row K of QUEUE, the row that maintains the queue of elements with priority no. K.

	FRONT	REAR
1	2	2
2	1	3
3	0	0
4	5	1
5	4	4

	1	2	3	4	5	6
1		AAA				
2	BBB	CCC	XXX			
3						
4	FFF				DDD	EEE
5				GGG		

Application of Queues : Round Robin Algo.

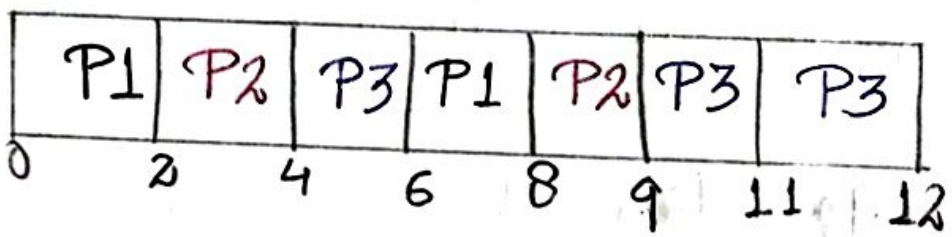
* A popular use of queue data structure is the scheduling problem in the operating system.

* Round-Robin is one of the simplest scheduling algorithm for processes in an operating system, which assigns time slices to each process in equal portions and in order, handling all processes without priority.

Example :- consider following three process

Process Queue	Burst time
P1	4
P2	3
P3	5

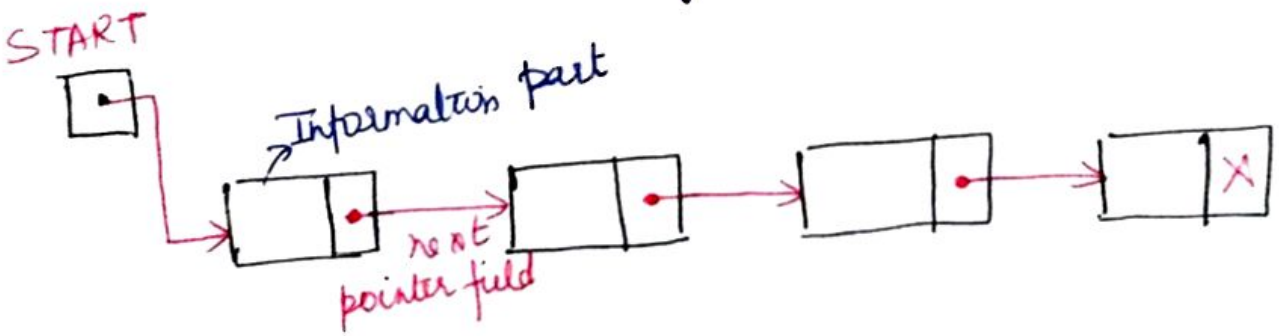
Time Slice = 2



Linked lists

* A linked list or one-way list is a linear collection of data elements called nodes, where the linear order is given by means of pointers.

* Each node is divided into two parts :-
  1) Information of the element
  2) link field or nextpointer, contains the address of the next node in the list



* the linked list contains a list pointer variable START- which contains the address of the first node in the list.

* The pointer to the last node contains a special value, called the null pointer (which is any invalid address)

Representation of linked list in Memory :-

* Let ~~List~~ LIST be a linked list. Then LIST will be maintained in memory as follows :-

  * First of all LIST requires two linear arrays - INFO and LINK, such that INFO[K] and LINK[K] contain respectively, the information part & next pointer field of a node LIST.

  * LIST also require a variable name such as START - which contains the location of the beginning of the list.

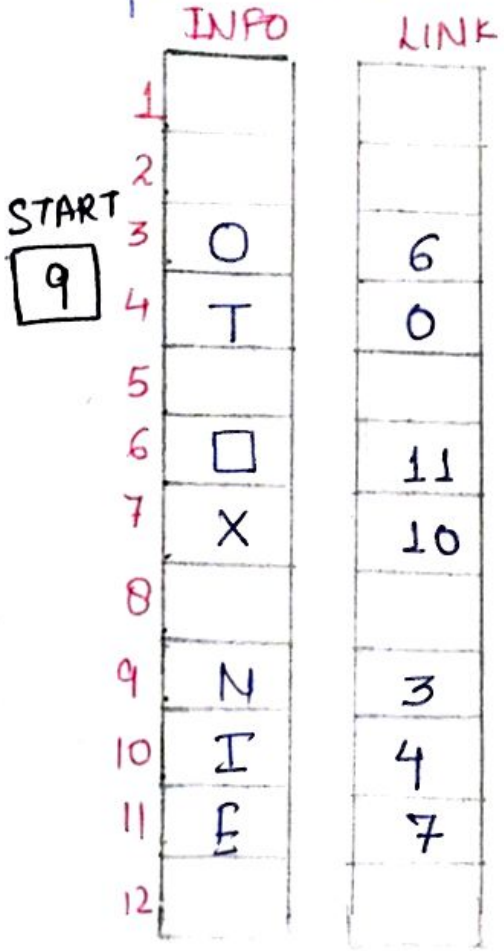
  * A next pointer sentinel - denoted by NULL, which indicates the end of the list.

Note :- We'll choose '0' to indicate NULL.

* The following examples of linked list indicate that the nodes of a list need not occupy adjacent elements in the arrays INFO and LINK.

* More than one list may be maintained in the same linear arrays INFO & LINK.

* However, each list must have its own pointer variable giving location of its first node.



START = 9, so INFO[9] = N is the first character.

LINK[9] = 3, INFO[3] = O

LINK[3] = 6, INFO[6] = □ (blank)

LINK[6] = 11, INFO[11] = E

LINK[11] = 7, INFO[7] = X

LINK[7] = 10, INFO[10] = I

LINK[10] = 4, INFO[4] = T

LINK[4] = 0, the list ended.

In other words, NO EXIT is the character string.

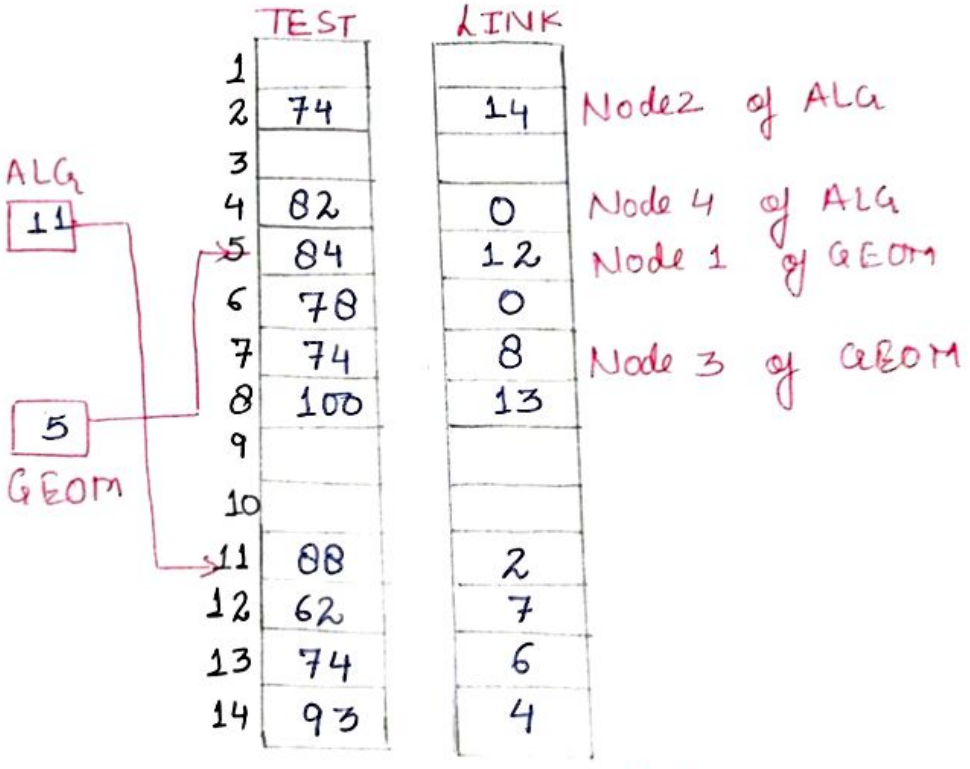
Example :-

Following figure shows how two lists of test score, ALG and GEOM, may be maintained in memory where the nodes of both lists are stored in the same linear array test & link.

* ALG contains 11, the location of its first node, and GEOM contains 5, the location of its first node.

ALG :- 88, 74, 93, 82

GEOM :- 84, 62, 74, 100, 74, 78



Creating a singly linked list

```
# include <stdio.h>
# include <conio.h>
# include <alloc.h>

void creat()
{
  char ch ;
      do
      {
      struct node * new_node , * current ;
new_node = (struct node *) malloc (sizeof (struct node));

 printf (" Enter the data:");
 scanf (" %d ", & new node -> data);
  new node -> next = NULL ;
   if (start == NULL) {
     start = new_node ;
     current = new_node ;
   }
  else
   {
    current -> next = new_node ;
    current = new_node ;
   }
 printf ("Do you want to create another");
    ch = getche (),
  }
  while (ch!='n');
}
```

Step 1 :- include alloc.h

* We'll be allocating memory using Dynamic Memory Allocation f^n.
* All those functions are included in alloc.h.

Step 2 :- define node structure

```
* struct node {
        int data ;
        Struct node *next ;
  } * start = NULL;
```

Step 3 :- Create node using Dynamic memory allocation:

* We don't have prior knowledge about no. of nodes. So we are calling malloc function to create node at run time.

```
new node = (struct node *) malloc (sizeof (struct node));
```

Step 4 :- Fill information in newly created node:

* Whenever we create a new node, make its next field NULL.

```
new_node → next = NULL
```

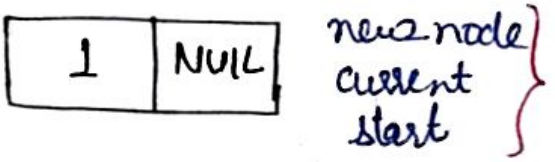
Step 5 :- Creating very first node:

* If node created in the above step is very first node then we need to assign it as starting node.

Step 6 :- Creating very First Node.

If the node created in the above step is very first node then we need to assign it as starting node.

Thus first three nodes have names :-



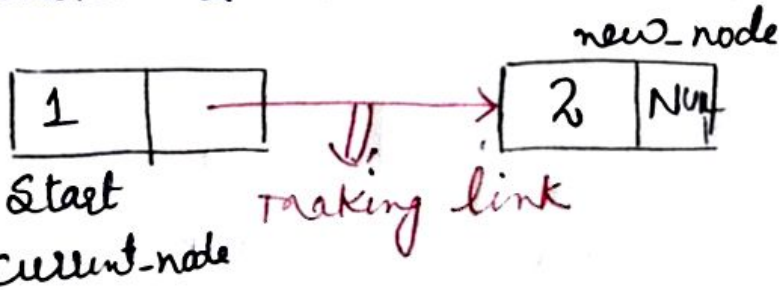
Step 7 :- Creating Second or n^{th} node :-

→ Lets assume, we have 1 node already created i.e., we have first node.

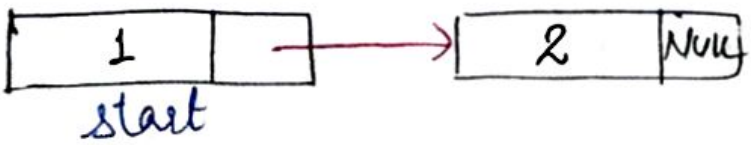
→ Now we have called creat() function again.

```
else
{
    current → next = new_node;
    current = new_node;
}
```

(i) first of all we'll create link bet^n current & new node



(ii) Now, move current pointer to next_node



Different operations on Single linked list

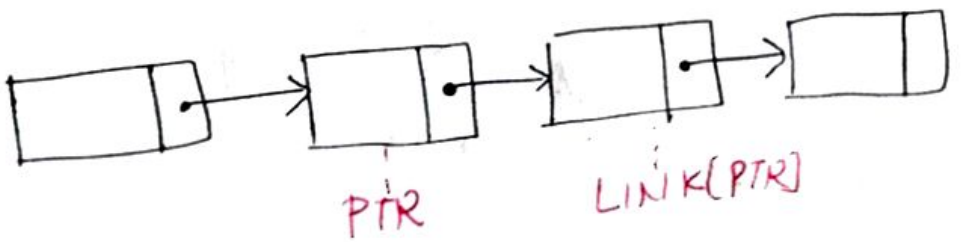
* Let LIST be a linked list in memory stored in linear arrays INFO and LINK.

Traversing a linked list :-

* Let START pointing to the first element and NULL indicating the end of list.
* Suppose we want to traverse LIST in order to process each node exactly once.
* This traversing algo. uses a pointer variable PTR which points to the node that is currently being processed.
* Accordingly LINK[PTR] points to the next node to be processed. Thus the assignment,

$$\boxed{PTR = LINK[PTR]}$$

moves the pointer to the next node in the list.



TRAVERSE_LIST( )

1. Set PTR = START [Initialize pointer PTR]
2. Repeat steps ③ and ④ while PTR ≠ NULL
3. Print INFO[PTR] [Process Node]
4. Set PTR = LINK[PTR] [PTR now points to next node.]
5. Exit

Searching A linked list :

* Let LIST be a linked list in memory.
* If ITEM is actually a key value & we are searching in LIST (assuming ITEM can appear only once in the LIST)

SEARCH (INFO, LINK, START, ITEM, LOC)

*LIST is a linked list in memory. This algo. finds the location LOC of the node where ITEM first appears in LIST or sets LOC=NULL

1) Set PTR = START
2) Repeat Step 3 while PTR ≠ NULL
3) If ITEM = INFO[PTR] then
   LOC = PTR & Exit

Else

Set PTR = LINK[PTR]

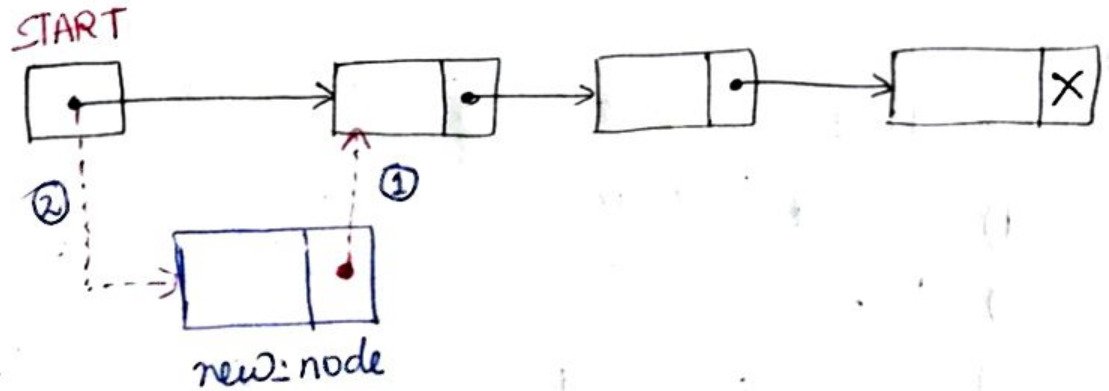
[End of Step 2 loop]

4. Set LOC = NULL [Search is Unsuccessful]
5. Exit.

Insertion into a Linked List :-

* Inserting nodes into linked list come up in various situations. We'll discuss three of them.
  (1) Inserting a node at the beginning
  (2) Inserting a node at the end
  (3) Inserting a node at specified location.

(1) Inserting at the Beginning of a list :-



Algo. INSFIRST

Step 1 : [Check for free space]

If new_node = NULL then Print "Overflow" and Exit.

Step 2 :- [allocate free space]

else new_node = create new node

Step 3 :- [Read info. part of new node]

data [new_node] = value

Step 4 : [link address of currently created node with the address of start is pointing]

next [new_node] = start

Step 5 :- [Now, assign address of newly created node to the start]

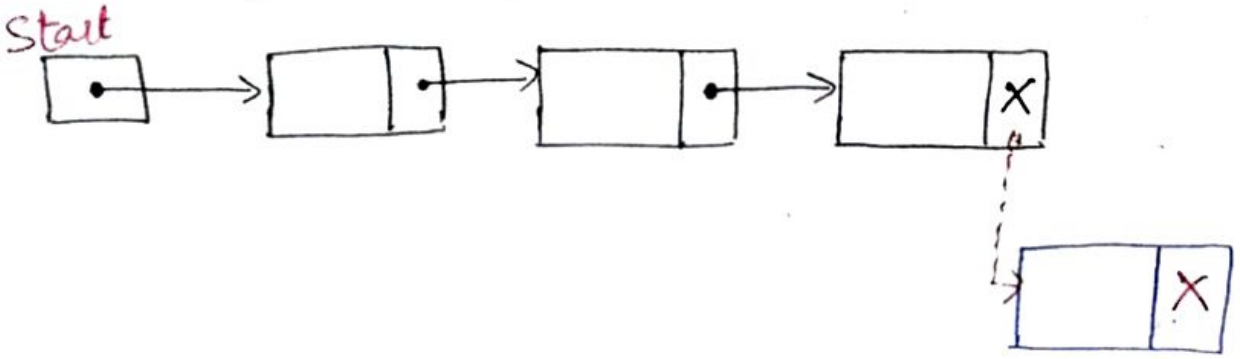
start = new_node

Step 6 :- Exit

Note :-

```
void insertion (struct link * node) {
        node = start.next ;
        previous = & start;
new_node = (struct link *) malloc (sizeof (struct link));
  new_node -> next = node ;
  previous -> next = new_node;
}
```

(2) Inserting at the end in the list :-



Step 1: [Check for free space]

if new_node = NULL then Print "Overflow"

& Exit

Step 2: [allocate free space]

Else

new_node = create new node.

Step 3: [Read value of info part of new node]

data [new_node] = value

Step 4: [Move the pointer to end of the list.]

Repeat while node ≠ NULL

node = next [node]

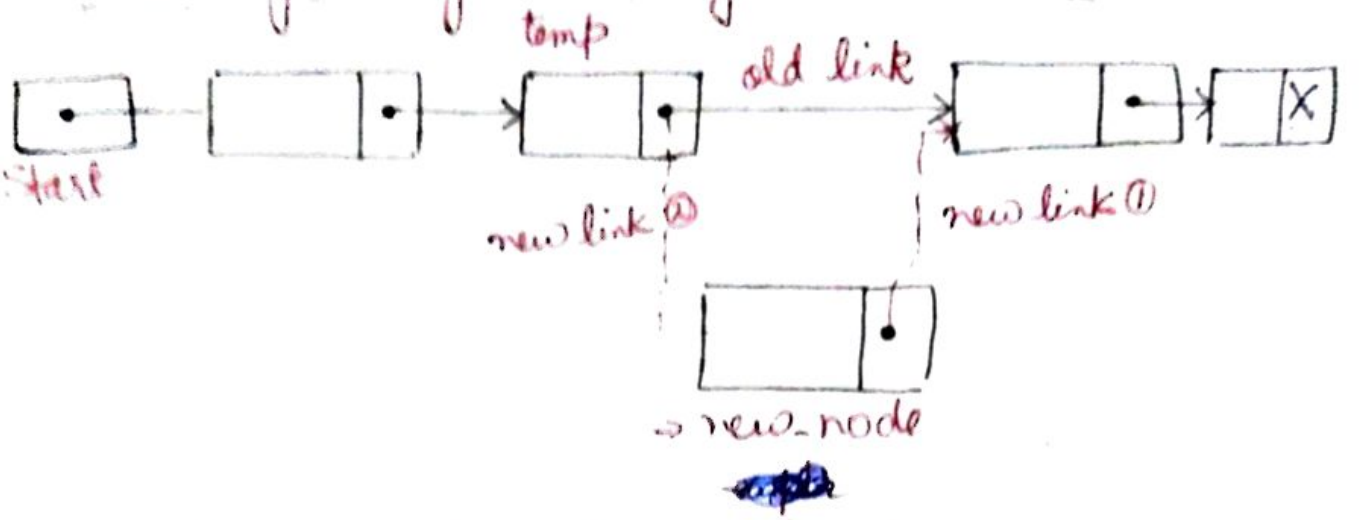
Step 5:-

next [node] = new node

next [new node] = NULL

Step 6: Exit

(2) Inserting after a given node in



Step 1: [check for free space]

if new_node = NULL then Print "Overflow" and Exit

Step 2: [allocate free space]

Else new_node = create a new node.

Step 3: [Read value of info part of new node]

data [new_node] = value

Step 4: [Move the pointer to desired location]

temp = start;

for (i=0; i < loc; i++) // loc is the desired location

temp = temp → next;

Step 5:

new_node → next = temp → next

temp → next = new_node;

Step 6: Exit

Deletion from the linked list

(1) Algo. for deleting the first node

Step 1 :- [Initialization]

node = start → next // points to first node in the list

prev = start // points at start

Step 2 :- [Perform deletion]

if node = NULL then Print "Underflow" & Exit.

else

prev → next = node → next;

Step 3 :- free the memory space associated with node.

Step 4 :- Exit

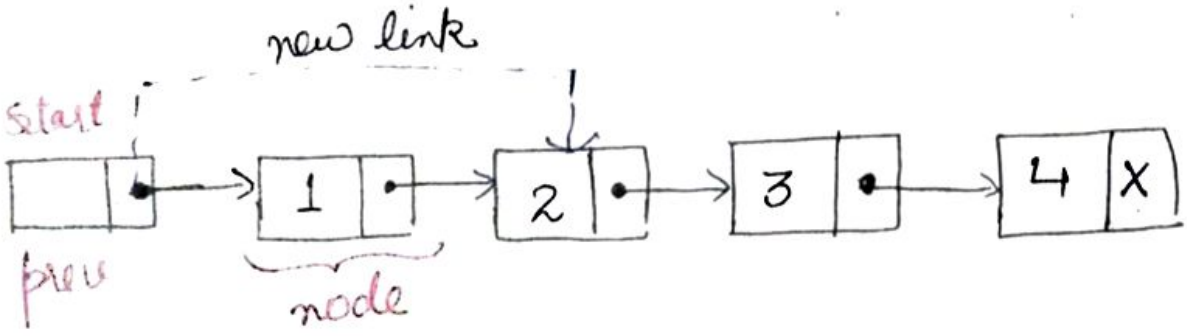


fig :- deleting first node

② Algo. for deleting last node :-

Step ① :- [Initialization]

node = start → next ;
prev = start ;

Step ② :- if node = NULL ; output "Underflow" & Exit.

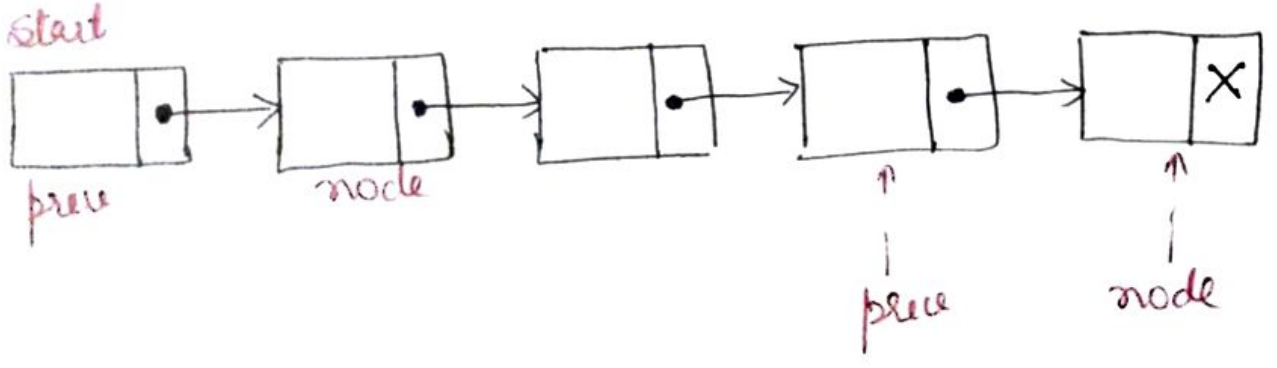
Step ③ :- [Reaching to the last node.]

Repeat while node ≠ NULL
node = node → next ;
prev = prev → next ;

Step ④ : prev → next = node → next // or NULL

Step ⑤ :- Free the memory space associated with node.

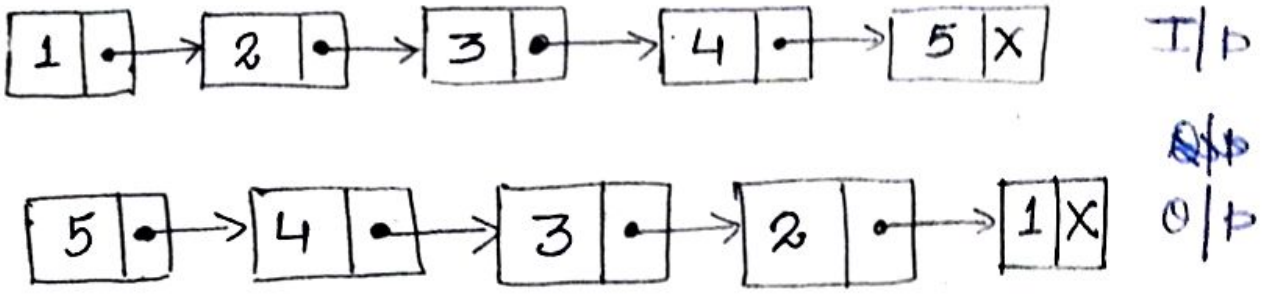
Step ⑥ :- Exit



⇒

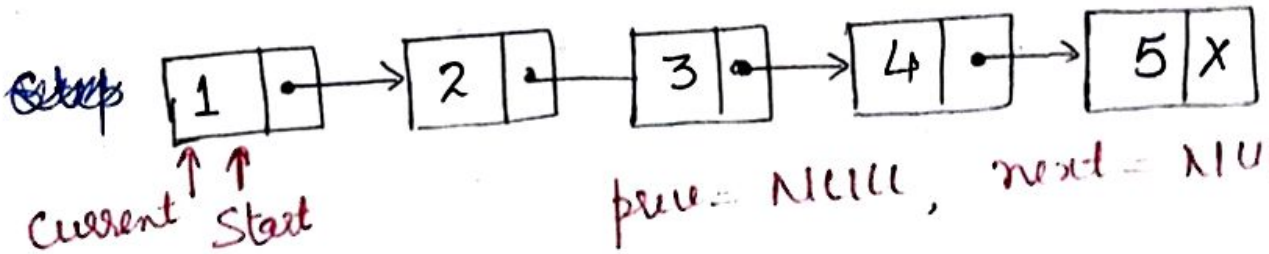
⇒

Reversing a Single Linked List :-



Step 1 :- Initialize three pointers;

prev = NULL
current = start
next = NULL



prev = NULL, next = NULL

Step 2 :- Traverse / iterate through the linked list;

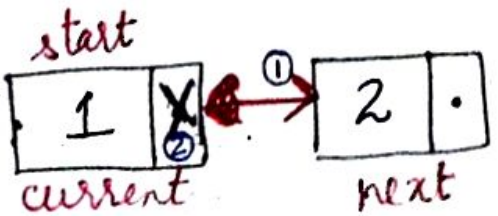
① --- next = current → next // store address of next node

② -- current → next = prev // Reversing the link

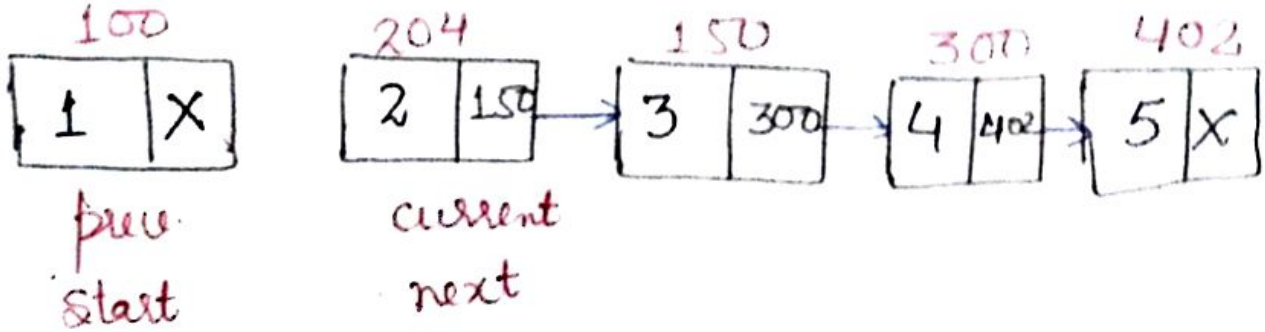
③ - prev = current
④ - current = next } Moving prev & current one step forward

← Pass 1

after ① & ②

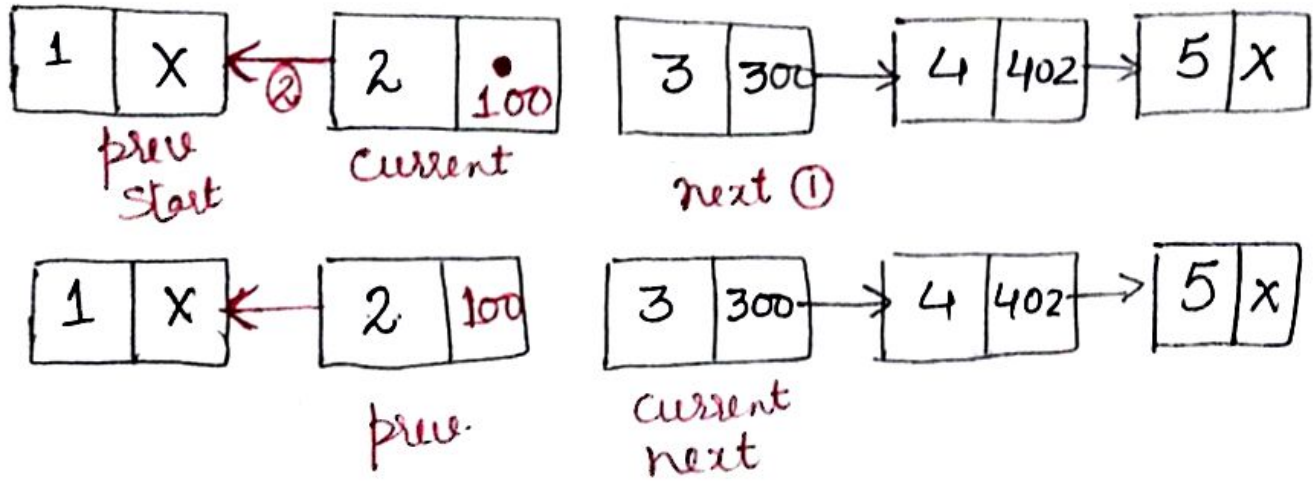


after ③ & ④



Pass ② :-

①&②




```
/* C function to reverse the linked list
Void reverse()
{
Node * current = start;          // Initialization
Node * prev = NULL, * next = NULL;  // ,,
while (current != NULL) {
        next = current -> next;     // Store next
        current -> next = prev;     // Reverse current node.
        prev = current;             } move pointers one
        current = next;             } position ahead.
}
```

```
    head = prev;
}
```

Advantages & disadvantages of single linked list :-

Advantages :-

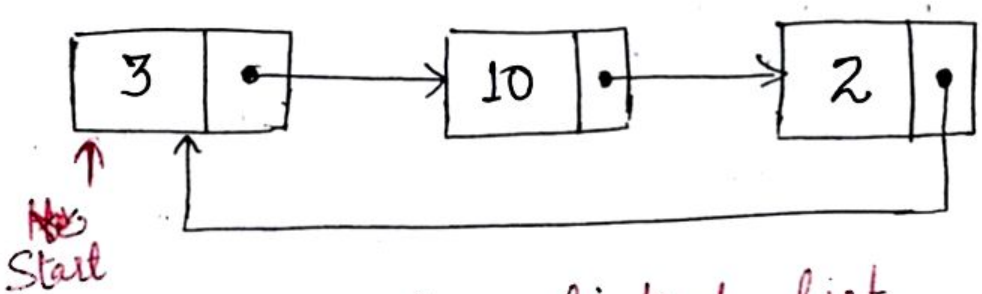
1. It is a dynamic data structure that is, it can grow & shrink during run-time.
2. Insertion & deletion operations are easier.
3. Efficient memory utilization (no need to pre allocate memory)
4. Linear data structures such as Stack, Queue can be easily implemented using linked list.

Disadvantages :-

1. Reverse traversing is very difficult in case of singly linked list.
2. No random access in linked list, we have to access each node sequentially.

Circular Linked List

* A circular linked list is a variation of a normal linked list.
* In circular singly linked list, the last node of the list contains a pointer to the first node of the list.



circular linked list

Implementing circular Linked List :

* This implementation is very similar to single linked list with the only difference that the last node will have it's next point to the Head / Start of the list.

Double linked list :-

* Doubly linked list / two-way list is a variation of linked list in which traversing can be done in both directions. i.e.
* in the usual forward direction from the beginning of the list to the end, or in the backward direction from the end of the list to the beginning.
* Thus if given the location of a node N in the list, one now has immediate access to both the next node & the preceding node in the list.

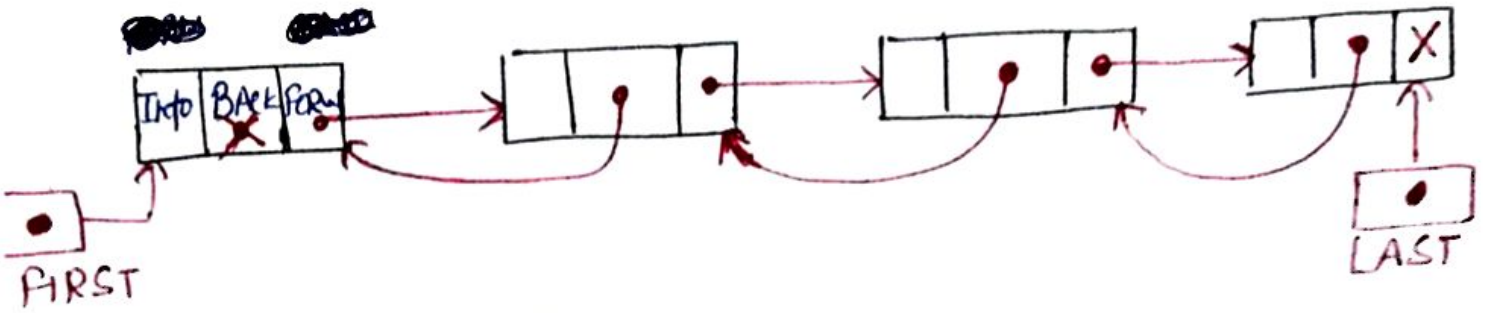
* A double linked list / two way list is a linear collection of data elements, called nodes, where each node is divided into three parts :

(1) An info. field INFO which contains the data of N.
(2) A pointer field FORW which contains the location of next node in the list.
(3) A pointer field BACK which contains the location of the preceding node in the list.

The list also require two list pointer variables :

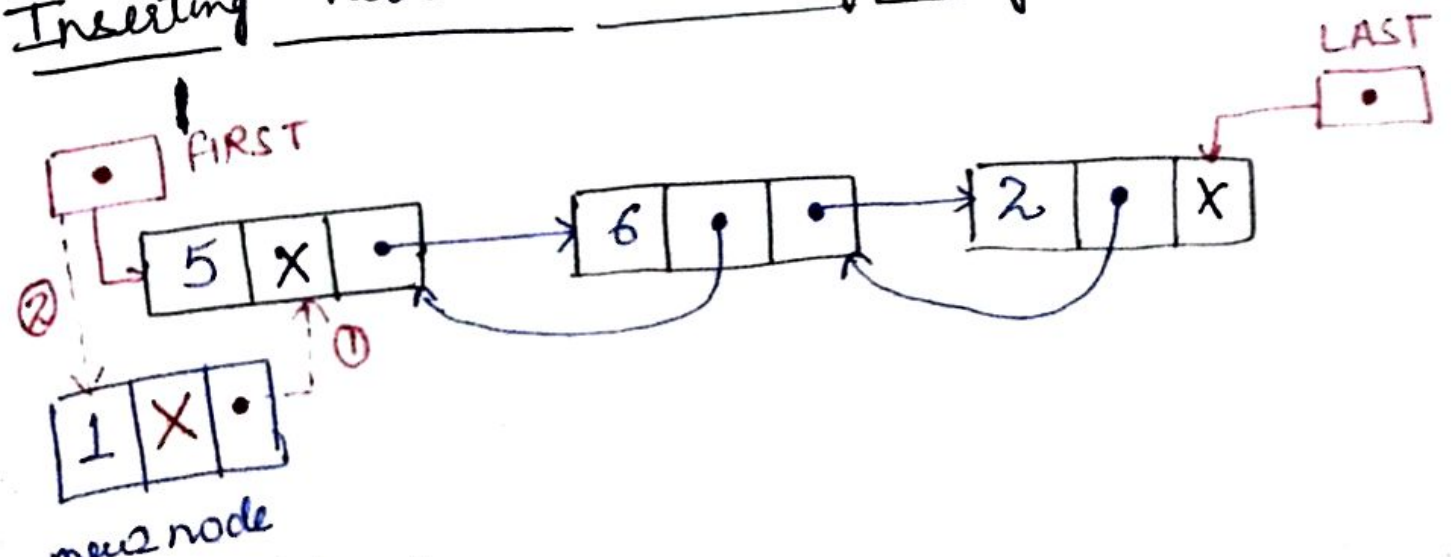
FIRST : which points to the first node in the list.

LAST :- which points to the last node in the list.




```
Struct node {
    int info;          // data
struct node *back;     // A reference to previous node.
struct node *forw;     // A reference to next node.
    }
```

Inserting node in the beginning :-



Algorithm :-

① Allocate the space for the new node in the memory.

new_node = (struct node *) malloc (sizeof (struct node));

UNIT-III

* Searching is the process of finding a given value position in a list of values.
* It decides whether a search key is present in the data or not.

Searching techniques :- There are two ways :-

1) Linear Search
2) Binary Search

Linear Search :-

⇒ linear search is a very simple search algorithm.

⇒ A sequential search is made over all items one by one.

⇒ Every item is checked & if a match is found then that particular location is returned otherwise search continues till the end of the data collection.

Algo. :-

→ Read the element to be searched.

→ compare the search element with the first element in the list.

→ if matched, display "Given element is found.".

→ if both are not matched, compare search element with the next element in the list

→ Repeat the above steps until search element is compared with last element in the list.

```
void main() {
    int A[20], n, i, ele;
pf("Enter the size of the list");
sf("%d", &n);

for(i=0; i<n; i++)
    sf("%d", &A[i]);
// linear search logic
for(i=0; i<n; i++)
  {
      if (ele == A[i]) {
    pf("Element is found at %d index", i);
        break;
    }
}
if (i==n)
  pf("Not found");
getch();
}
```

Binary Search :-

* Binary Search algo. can be used with only sorted list of elements.
* This search process starts comparing the search element with the middle element in the list.
* If both are matched, result is declared as found. otherwise we check whether the search element is smaller or larger than the middle element in the list.
* then the same process is repeated for upper or lower half.

```
// C Program for binary search.

Void main() {
int first, last, middle, n, i, ele, A[30];
pf("Enter the size of the list");
sf("%d", &n);
pf("Enter %d integer values in ascending
    order", n);
for (i=0; i<n; i++)
    sf("%d", &A[i]);
```

```
pf(" Enter the value to be searched ");
sf(" %d ", & ele);

first = 0;
last = n-1;
middle = (first + last) / 2;

while ( first <= last)
{
  if (A[middle] < ele)
      first = middle + 1;
  else if (A[middle] == ele)
   {
   pf("%d found at location %d", ele, middle+1);
   break;
  }
  else
   last = middle - 1;
  middle = (first + last) / 2;
}
 if ( first > last)
  pf(" Element % d not found ", ele);
return 0;
}
```

Sorting

* Sorting refers to the operation of arranging data in some given order, such as increasing or decreasing.

Bubble Sort :-

* Bubble sort is a simple algorithm which is used to sort a given set of n elements by comparing all the elements one by one & sort them based on their values.
* If the array has to be sorted in ascending order, then bubble sort will start by comparing the first element of the array with the second element, if first element is greater than the second element, it will swap both the elements. then move on to compare the second & third element & so on.
* It is known as bubble sort because with every complete iteration / pass the largest element in the given array, bubbles up towards the last place. (like a water bubble rises up to the water surface.)

Example :-

Pass / Iteration 1 : 14, 33, 27, 35, 10

14, 33 — No Swap

14, 33, 27, 35, 10

33, 27 — Swap

14, 27, 33, 35, 10

33, 35 — No Swap

14, 27, 33, 10, (35)

Pass 2 :- 14, 27, 33, 10, 35

14, 27 — No Swap

14, 27, 33, 10, 35

33, 10 — Swap

14, 27, 33, 10, 35 ⇒ 14, 27, 10, (33), (35)

Pass 3 :- 14, 27, 10, (33), (35)

14, 27 — No Swap

14, 27, 10, (33), (35)

27, 10 — Swap

14, 10, (27), (33), (35) ⇒ 14, 10, (27), (33), (35)

Pass 4 :- 14, 10, 27, 33, 35

14, 10 — Swap

14, 10, 27, 33, 35 ⇒ 10, 14, 27, 33, 35

Sorted List

BubbleSort (n, list)

Step 1 :- $i = 0$

Step 2 :- Repeat through step (5) while ($i < n-1$)

Step 3 :- $j = 0$

Step 4 :- Repeat through step (5) while ($j < (n-i)-1$)

Step 5 :- if list[j] > list[j+1]
temp = list[j]
list[j] = list[j+1]
list[j+1] = temp

Step 6 :- Exit

Insertion Sort

* Insertion sort works similarly as we sort cards in our hands in a card game.

Note :- we assume that first card is already sorted, we select an unsorted card. If unsorted card is greater than the card in hand, it is placed on the right otherwise to the left.

Pass 1 :- 9, 5, 4, 1, 3

Pass 2 :- 9, 5, 4, 1, 3 ⇒ 5, 9, 4, 1, 3

Pass 3 :- 5, 9, 4, 1, 3 ⇒ 4, 5, 9, 1, 3

Pass 4 :- 4, 5, 9, 1, 3 ⇒ 1, 4, 5, 9, 3

Pass 5 :- 1, 4, 5, 9, 3 ⇒ 1, 3, 4, 5, 9
sorted list

algorithm :-

1) for j = 2 to n
2) Key = A[j]
3) i = j - 1
4) while (i > 0 and A[i] > Key) {
5) A[i+1] = A[i]
6) i = i - 1 }
7) A[j+1] = Key

C Program for Insertion Sort

```
main() {
   int A[50], i, j, k, n, temp;
   pf("Enter no of elements");
   sf("%d", &n);
   pf("Enter the array elements");
     for (i=0; i<n; i++)
       sf("%d", &A[i]);
    for (k=1; k<n; k++)
     {
          temp = A[k];
          j = k-1
       while (temp < A[j] && j>=0) {
            A[j+1] = A[j];
```

```
        j = j - 1;
    }
    A[j+1] = temp;
}
```

Q :- 25, 15, 30, 9, 99, 20, 26

Selection Sort

* Selection Sort algorithm works by finding the smallest number from the array & then placing it to the first position
* The next array that is to be traversed will start from index next to the position where the smallest no. is placed.

15, 20, 10, 30, 50, 18, 5, 40

Iteration 1 :- 15, 20, 10, 30, 50, 18, 5, 40

No Swap

15, 20, 10, 30, 50, 18, 5, 40

Swap ⇓

10, 20, 15, 30, 50, 18, 5, 40

No Swap

No Swap

No Swap

Swap ⇓

5, 20, 15, 30, 50, 18, 10, 40

After Iteration 2 :- 5, 10, [illegible], 30, 50, 18, 15, 40

After Iteration 3 :- 5, 10, 15, 30, 50, 20, 18, 40

after Iteration 4 :- 5, 10, 15, 18, 50, 30, 20, 40

after Iteration 5 :- 5, 10, 15, 18, 20, 50, 30, 40

after Iteration 6 :- 5, 10, 15, 18, 20, 30, 50, 40

after Iteration 7 :- 5, 10, 15, 18, 20, 30, 40, 50

algorithm :-

Step 1 :- Select 1st element of the list.

Step 2 :- Compare the selected element with all the other elements in the list.

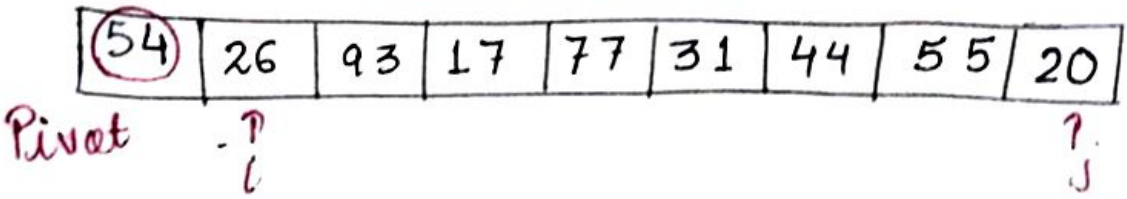
Step 3 :- In every comparison, if any element is found smaller than the selected element, both are swapped.

Step 4 :- Repeat the same procedure with element in the next position in the list till entire list is sorted.

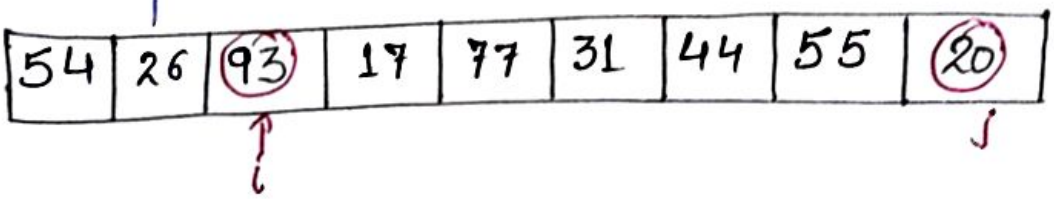
C Program for Selection Sort

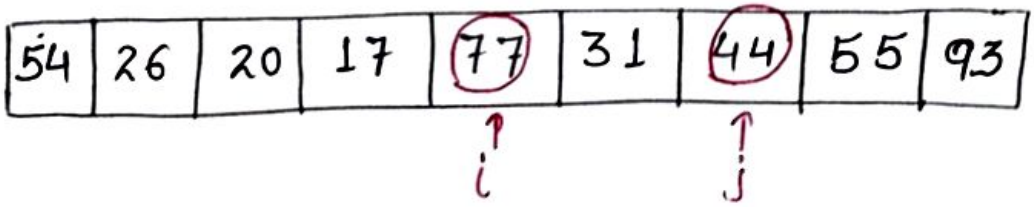
```
void main() {
  int n, i, j, temp, A[50];
  pf("Enter the size of the array");
  sf("%d", &n);
  for (i=0; i<n; i++)
    sf("%d", &A[i]);
  // Selection Sort logic
  for (i=0; i<n; i++)
  {
    for (j= i+1; j<n; j++) {
      if (A[i] > A[j])
      {
        temp = A[i];
        A[i] = A[j]=
        A[j] = temp;
      }
  } }
```

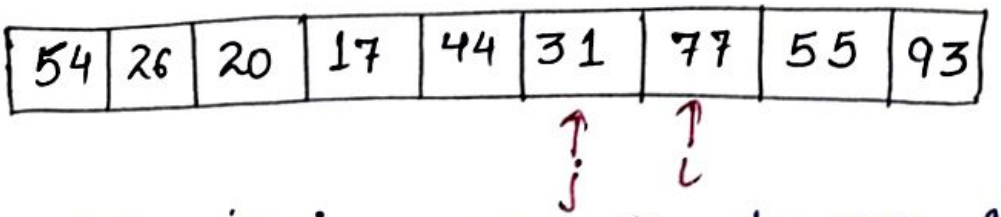
Quick Sort



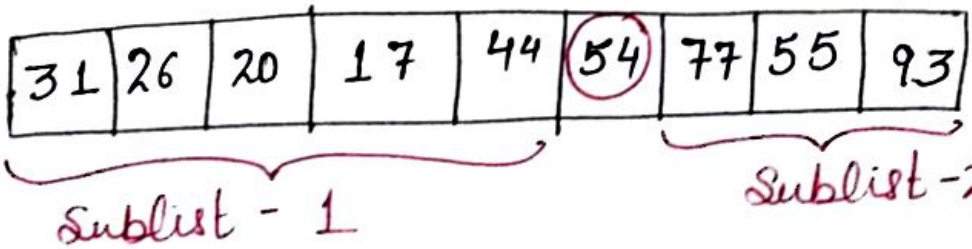
Increment i from L to R until we get an element greater than pivot (54).
Simultaneously decrement j until we get less than pivot!







as $i > j$, Stop the process & interchange A[j] with Pivot



* Thus, after applying the same method again & again for new sublists, we get the sorted list.

```
Procedure QuickSort (A, p, r)
1.    if p < r then
2.      q = call to Partition (A, p, r)
3.      call QuickSort (A, p, q)
4.      Call QuickSort (A, q+1, r)
```

```
Partition (A, p, r)
This algo arranges the subarray A[p..r]
1.  Set x = x[p]
2.     i = p
3.     j = r+1
4.  while (i < j) {
5.        while (A[i] <= x)
6.           Set  i = i+1
7.        while (A[j] > x)
8.           Set  j = j-1
9.     if (i < j) then
10.      Set A[i] ↔ A[j]}   Swap
11.      Swap A[p] & A[j]
12.    return j.
```

(5) Heap Sort

* A heap is defined as an almost complete binary tree of n nodes such that the value of each node is less than or equal to the value of its father.

* This implies that root of binary tree has the largest key in Key. This type of heap is usually called descending heap or max heap.

Sorting of array using Heapsort procedure consists of 3 procedures :-

(1) Build-Max-Heap procedure produces a max heap from an unordered i/p array.

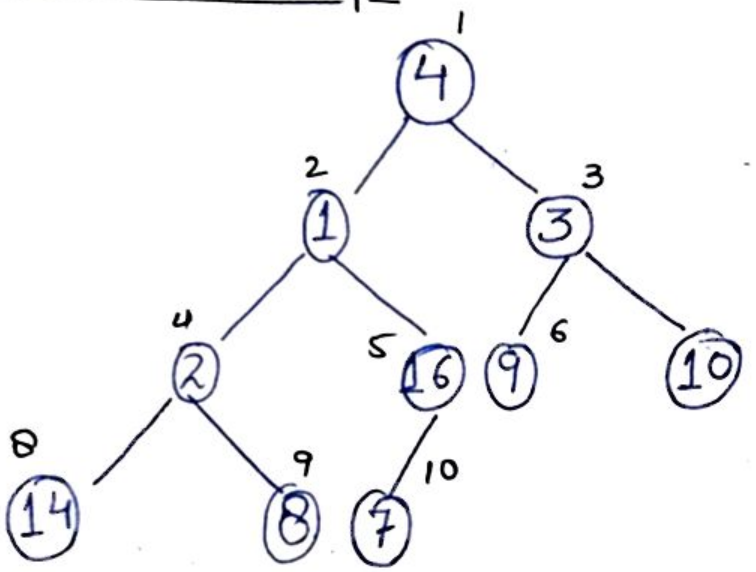
(2) Max-Heapify procedure is the key to maintain the max-heap property.

(3) Heap Sort, sorts an array.

Eg.,

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

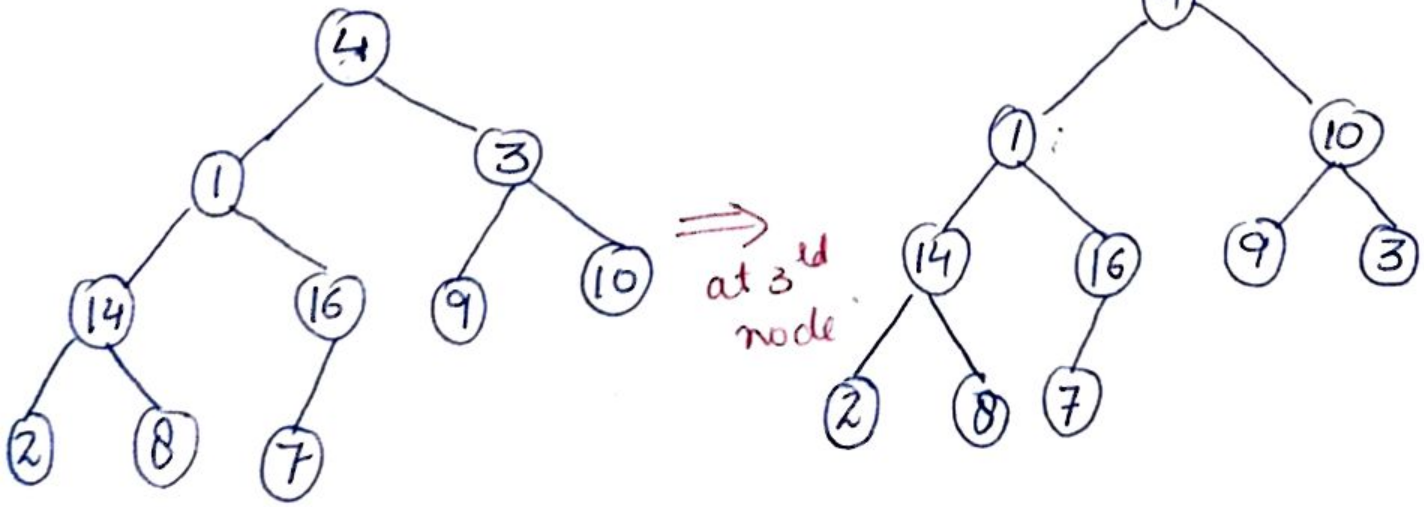
Build-Max-Heap :-



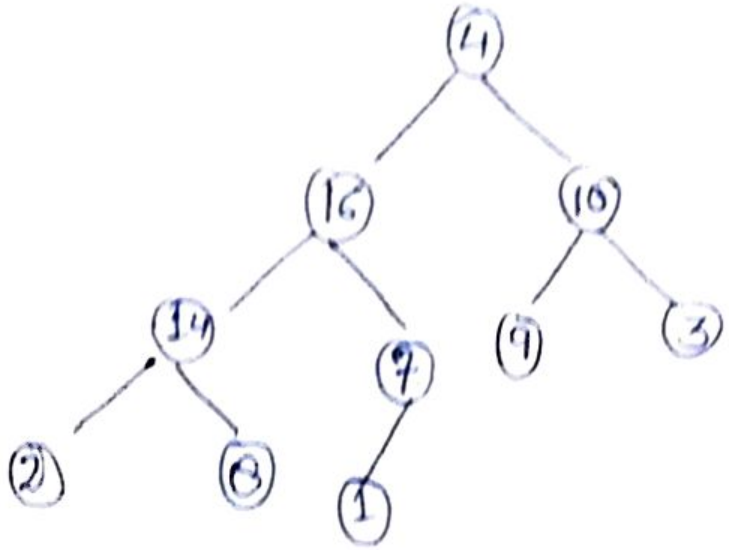
we'll start heapifying the above tree from length[A]/2 to 1 i.e., 5 to 1

→ at 5th node Parent - child already creates a heap.

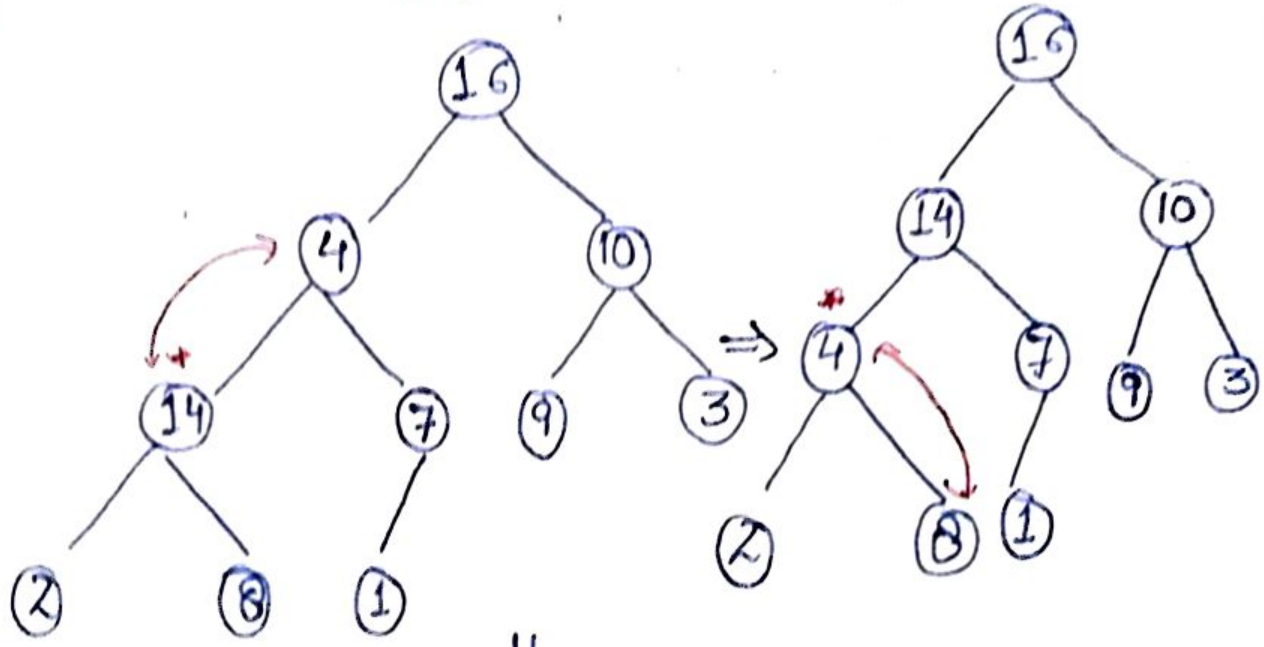
→ at 4th node we'll do following :-



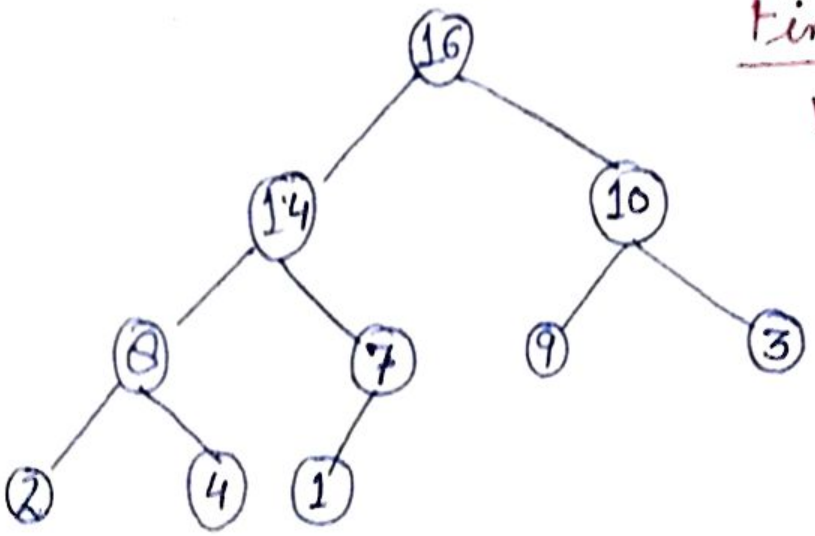
after max-Heapify at 2nd node



after max-Heapify at 1st node :-



Final Heap



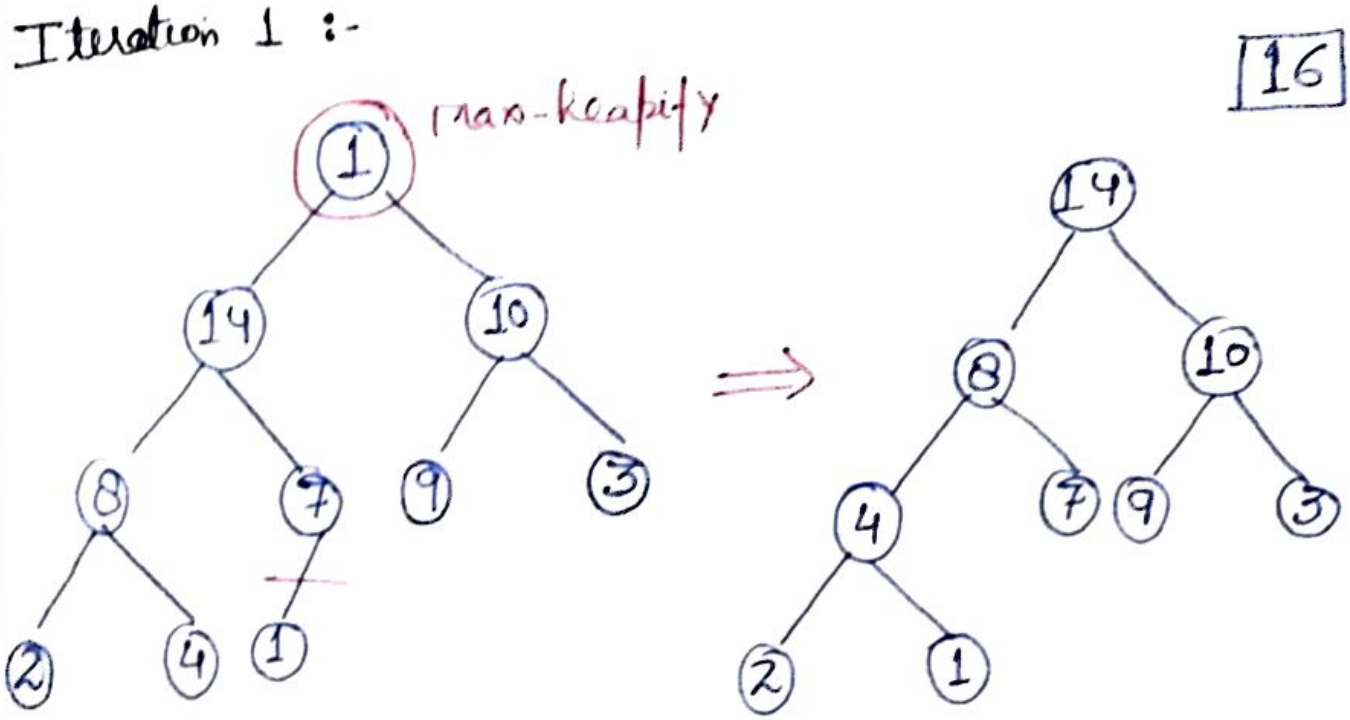
Heap - Sort

Swap A[1] with A[n]

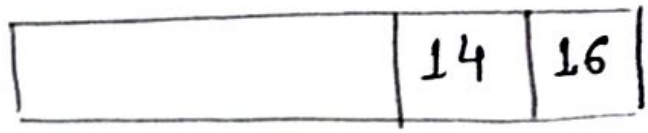
Reduce n by n-1 then again

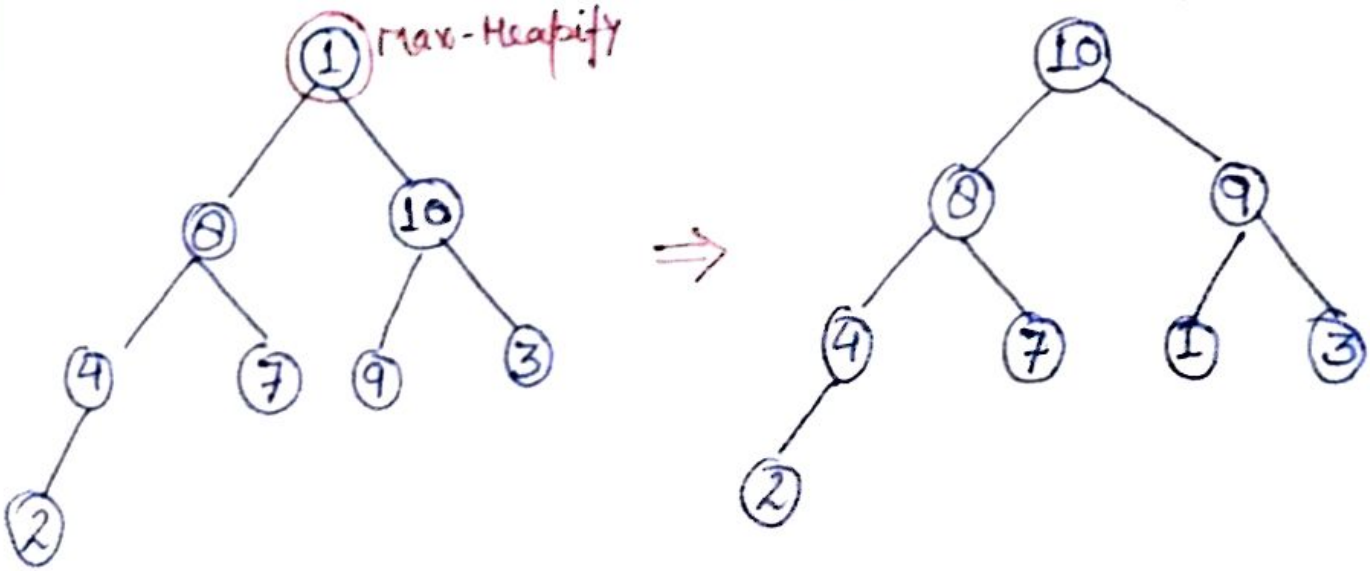
Max-Heapify at A[1]

Iteration 1 :-

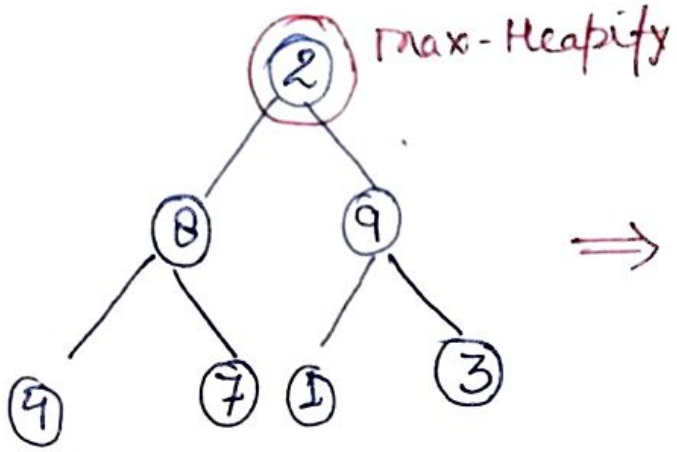


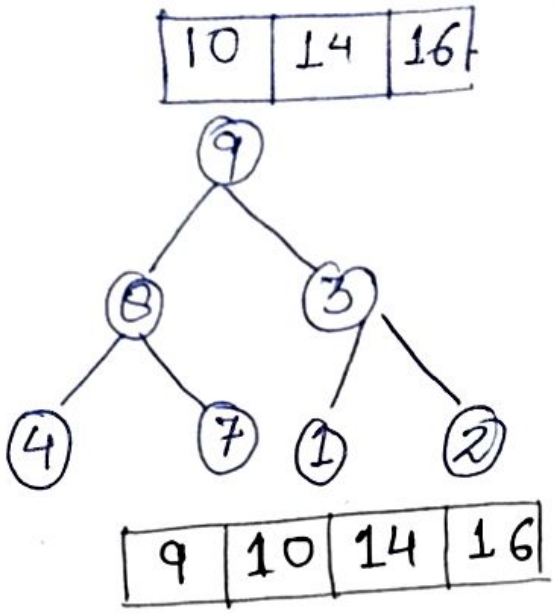
Iteration 2 :-





Iteration 3 :-

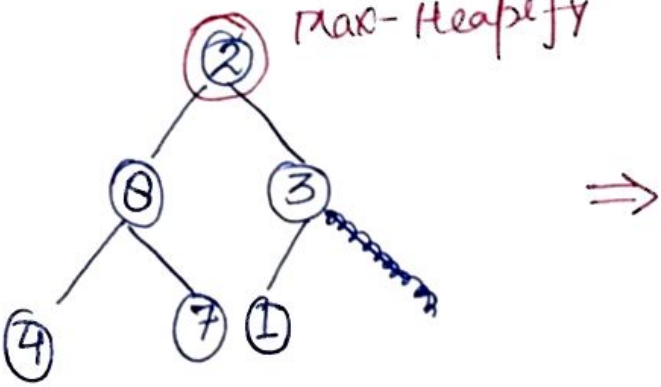


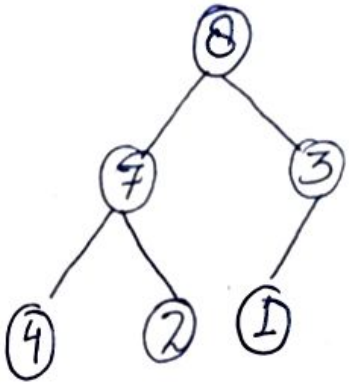


10	14	16

9	10	14	16

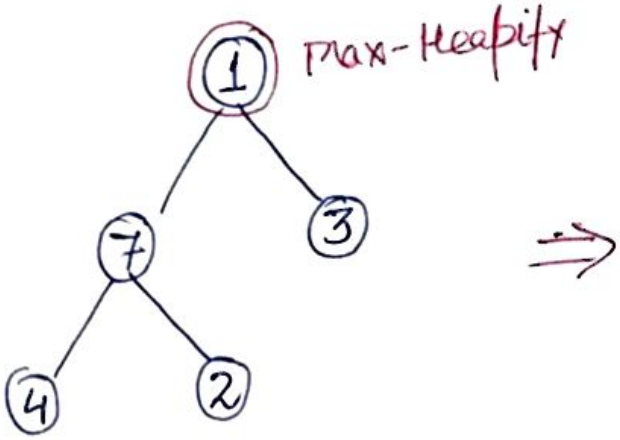
Iteration 4 :-

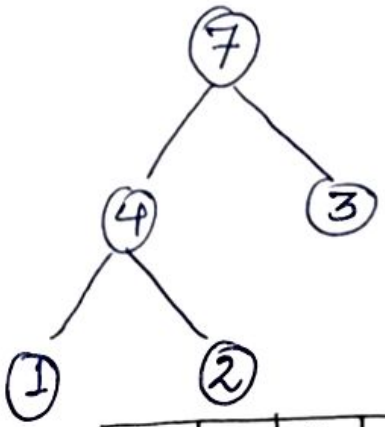




8	9	10	14	16

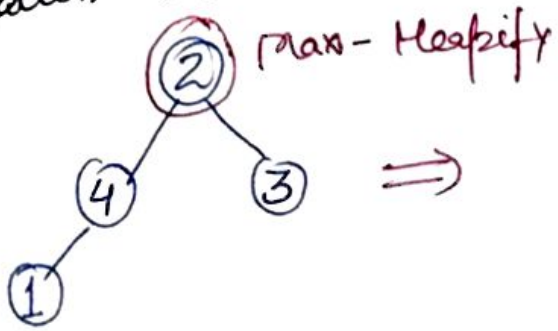
Iteration 5 :-

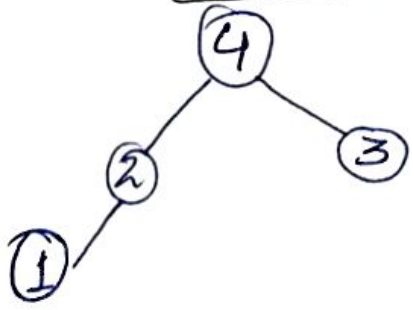




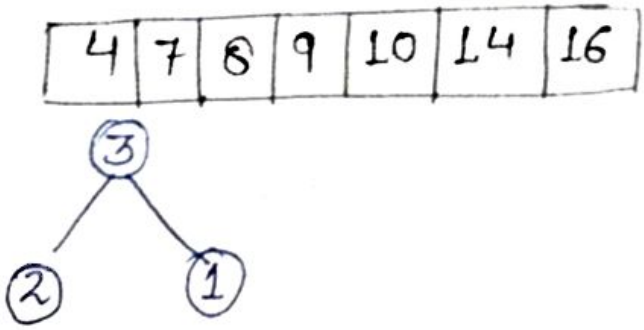
7	8	9	10	14	16

Iteration 6 :-

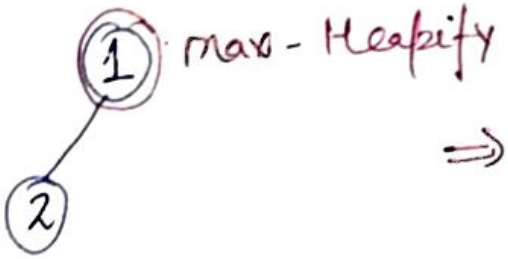


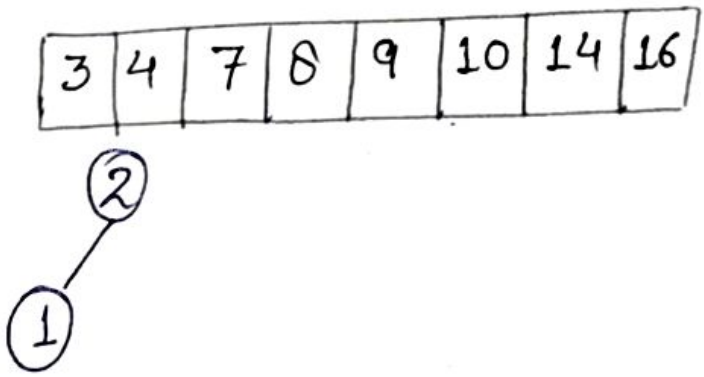


Iteration 7 :-



Iteration 8 :-





Question :- Sort the following list using Heap Sort

25, 55, 46, 35, 10, 90, 84, 31

Max - Heapify :

* When Max-Heapify is called, it is assumed that the binary trees rooted at Left(i) & right(i) are max Heaps., but A[i] may be smaller than its children, thus violating the max-Heap property.

* Thus its function is to let the value at A(i) "float down" in the heap so that subtree rooted at i becomes a max-Heap.

```
Max-Heapify(A,i)
1.  l = left(i)
2.  r = Right(i)
3.  if l ≤ Heap-size[A] and A[l] > A[i]
4.      then largest = l
5.          else largest = i
6.  if r ≤ Heap-Size[A] and A[r] > A[largest]
7.      then largest = r
8.      if largest ≠ i
9.          then exchange A[i] & A[largest]
10.             Max-Heapify(A, largest).
```

Build - Max-Heap (A)

1. Heap-size [A] = length[A]
2. for i = length[A]/2 downto 1
3. do Max-Heapify (A, i)

Running time :- $O(n \lg n)$, as we'll call Max-Heapify ~~(lgn) for n times~~.

Heap-Sort (A)

1. Build-Max-Heap (A) — n
2. for i = length[A] downto 2
3. do exchange A[1] with A[i]
4. Heap-size [A] = Heap-size [A] - 1
5. Max-Heapify (A, 1)

Merge Sort (6)

5) Divide & Conquer Approach :-

* Most of the algo. are recursive in nature (i.e., for solving a particular problem, they call themselves repeatedly one or more times). All these algo. follow the Divide & Conquer approach to accomplish a given task.

* In this approach, whole problem is divided into several subproblems. These subproblems are similar to the original problem but smaller in size. All these subproblems are then solved recursively. & then these solutions are combined.

* At each level of recursion the divide & conquer approach follows three steps :-

Step 1 :- Divide

The whole problem is divided into several subprob.

Step 2 :- Conquer

The subprob. are conquered by solving them recursively. Only if they are small enough to be solved; otherwise step 1 is executed.

Step 3 :- Combine

Finally, the solutions obtained by the subproblems are combined to create solution to the original problem.

The Merge Sort algo. closely follow the divide & conquer Paradigm. It operates as follows:-

Divide :- Divide the n-element seq. to be sorted into two subsequences of $n/2$ elements each.

Conquer :- Sort the two subsequences recursively using Merge Sort.

Combine :- Merge the two sorted subsequences to produce the sorted answer.

* The key operation of the mergesort algo. is the merging of two sorted subsequences in combine step.

* To perform merging, we use an auxiliary procedure MERGE(A, p, q, r), where A is an array and p, q & r are indices numbering elements of the array s.t. $p \le q \le r$.

* The procedure assumes that the subarrays $A[p \ldots q]$ & $A[q+1 \ldots r]$ are in sorted order. It merges them to form a single sorted subarrays that replaces the current subarray $A[p \ldots r]$.

```
Merge (A, p, q, r)
1  n1 ← q - p + 1          { length of subarrays A[p .. q]
2  n2 ← r - q              { length of subarray A[q+1 .. r]
3  create arrays L[1 ... n1 + 1] & R[1 .. n2 + 1]
4  for i ← 1 to n1
5       do L[i] ← A[p + i - 1]      } copies the subarrays A[p .. q]
                                      into L[1 .. n1]
6  for j ← 1 to n2
7       do R[j] ← A[q + j]          } copies the subarrays A[q+1 .. r]
                                      into R[1 .. n2]
```

```
8.  L[n1 + 1] ← ∞      } Put Sentinel at the end of L.
9.  R[n2 + 1] ← ∞      } Put Sentinel at the end of R.
10.  i ← 1             } Initialization
11.  j ← 1
12.  for K ← p to r
13.      do if L[i] ≤ R[j]
14.          then A[K] ← L[i]
15.               i ← i + 1
16.          else A[K] ← R[j]
17.               j ← j + 1
```

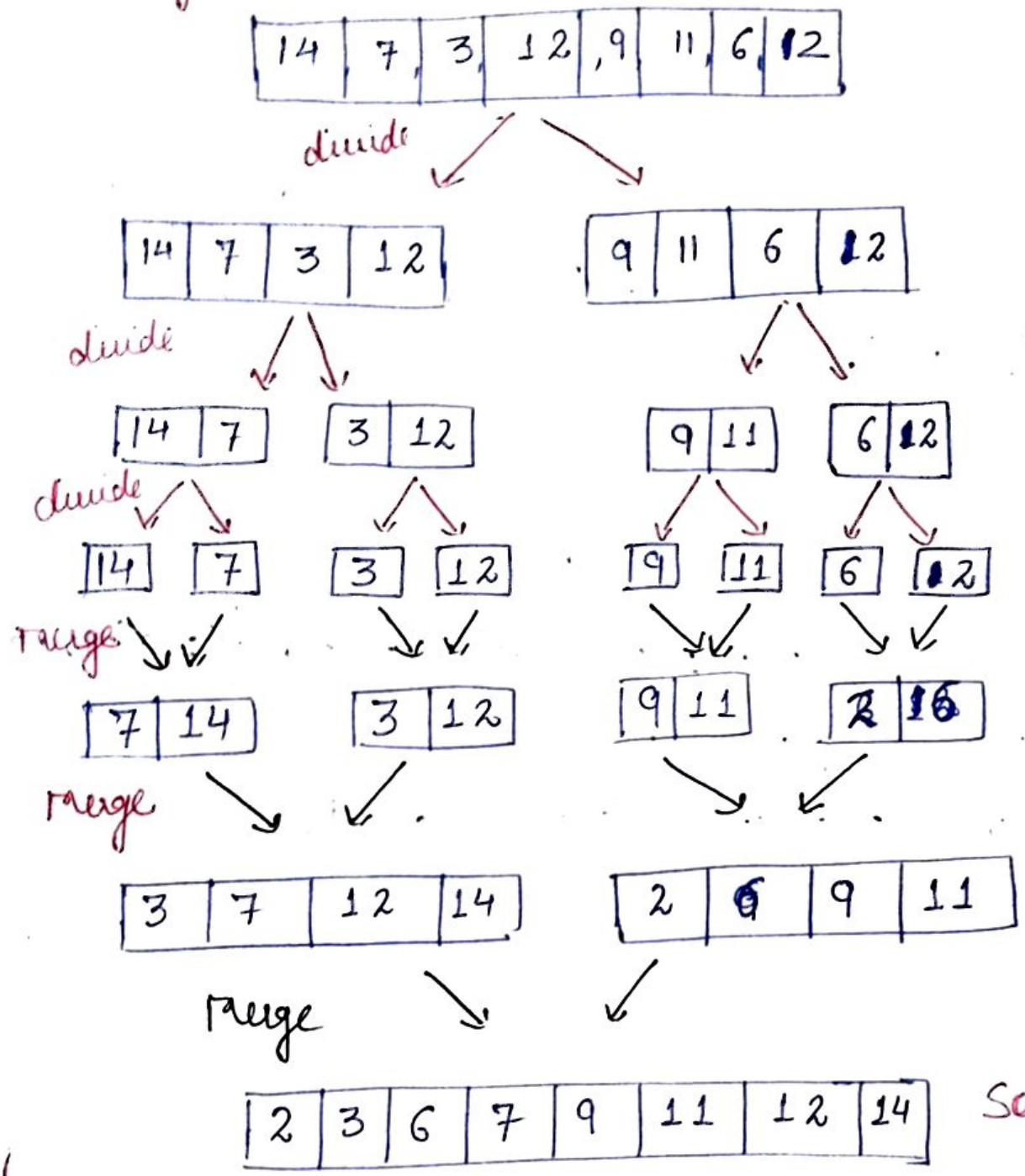
Running time of Merge Procedure is $\theta(n)$. As

* lines 1-3 & 8-11 takes constant time,
* lines 4-7 take $\theta(n_1 + n_2)$ i.e., $O(n)$ time

* We can now use the Merge Procedure as a subroutine in the Mergesort algo. The procedure MergeSort (A, p, r) sorts the elements in the subarray $A[p..r]$.
* If $p \geq r$, the subarray has at most one element & is therefore already sorted. Otherwise, the divide step computes an index q that partition $A[p..r]$ into two subarrays: $A[p..q]$ containing $\lceil n/2 \rceil$ elements & $A[q+1..r]$ containing $\lfloor n/2 \rfloor$ elements.

```
MergeSort (A, p, r)
1   if p < r
2       then q ← ⌊(p + r)/2⌋
3            MergeSort (A, p, q)
4            MergeSort (A, q+1, r)
```

Merge sort Example :-



(8) Counting Sort

* Counting Sort algo. is an efficient sorting algo. that can be used for sorting elements within a specific range.
* This sorting technique is based on the frequency / count of each element to be sorted.
* It assumes that each of the 'n' i/p element is an integer in the range 1 to K.

Example :- A

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
2	1	1	0	2	5	4	0	2	8	7	7	9	2	0	1	9

$n = 17$
$K = 9$

C

3	3	4	0	1	1	0	2	1	2
0	1	2	3	4	5	6	7	8	9

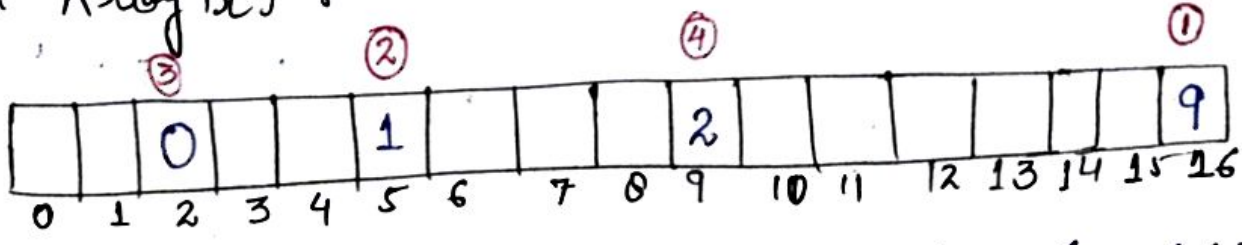
```
for( i = 0 ; i < n ; i++)
  {
    ++ C[A[i]] ;
  }
```

Update C :-

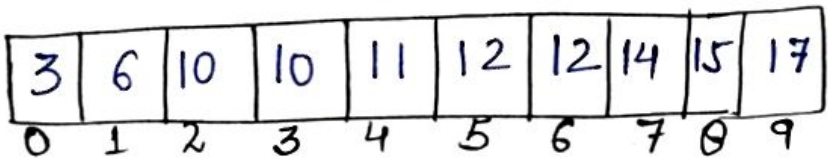
0	1	2	3	4	5	6	7	8	9
3	6	10	10	11	12	12	14	15	17

```
for (i=1; i<=K ; i++)
{
  C [i] = C [i] + C [i-1];
}
```

Output Array B[] :-



① Start with i = 16 the element is 9 (In A[])
Now check with updated C, value is 17.



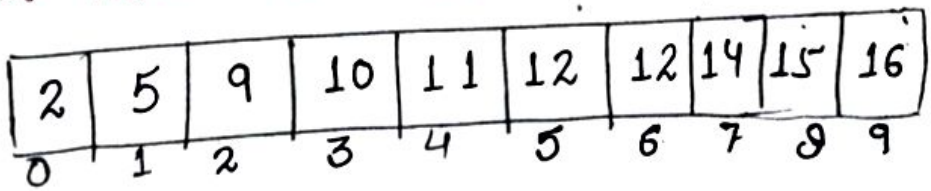
decrement by 1 i.e; 17-1 = 16,
store 9 at index 16 in B

② decrement i by 1 i.e; i = 15, A[15] = 1
C[1] = 6, decrement by 1 i.e; 6-1 = 5
store 1 at B[5]

③ Now i = 14, A[14] = 0, C[0] = 3,
decrement by 1, i.e; 3-1 = 2
store 0 at B[2]

④ Now i = 13, A[13] = 2, C[2] = 10
decrement by 1 i.e; 10-1 = 9
store 2 at B[9]

Till Now C is



we'll repeat the process till we reach $i = 0$ & we'll get the updated sorted array.

```
for (i = n-1; i >= 0; i--)
{
  B[--C[A[i]]] = A[i];
}
```

```
Counting-Sort (A, B, k)
1. for i = 0 to k
2.     do C[i] = 0
3. for j = 0 to n-1          /* length[A] = n */
4.     do ++C[A[j]];
5. for i = 2 to k
6.     do C[i] = C[i] + C[i-1]
7. for i = n-1 down to 0
8.     B[--C[A[i]]] = A[i];
```

Q.

0	1	2	3	4	5	6	7	
3,	5,	4,	1,	3,	4,	1,	4	← A

C

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

C

0	2	2	4	7	8
0	1	2	3	4	5

← Updated

Q :- A = 2, 1, 3, 5, 4, 2, 1, 4, 4, 3, 2, 1, 1, 3, 5

Radix - Sort

* It is the method that many people use when alphabetizing a large list of names. Specifically, the list of names is first sorted according to the first letter of each name.
* For the case of numbers, Least Significant digit is sorted first.

Example :-

* Radix Sort algo. requires the no. of passes which are equal to the no. of digits present in the largest no. among the list of nos.
* if largest no. is a 3-digit no., the list is sorted with 3 passes.

I/P
326
453
608
835
751
435
704
690

Pass 1
690
751
453
704
835
435
326
608

Pass 2
704
608
326
835
435
751
453
690

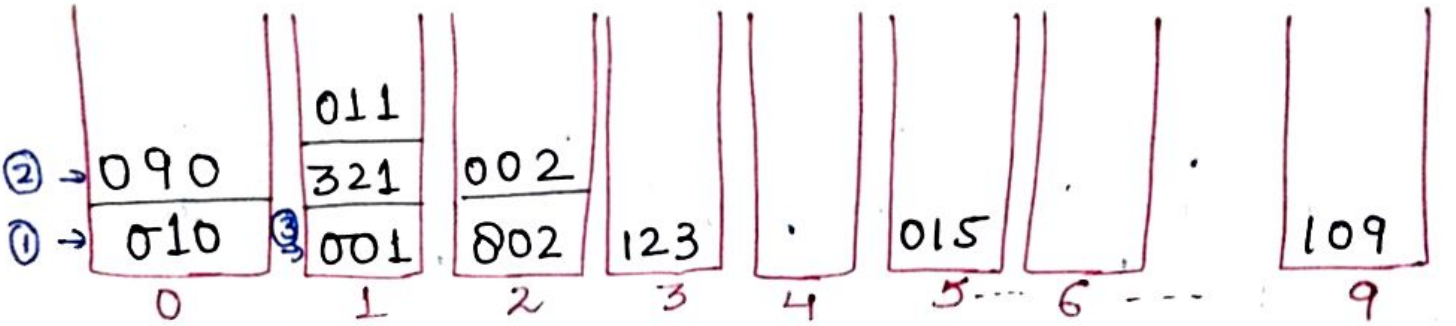
Pass 3
326
435
453
608
690
704
751
835

Example ② :-

015, 001, 321, 010, 802, 002, 123,

090, 109, 011

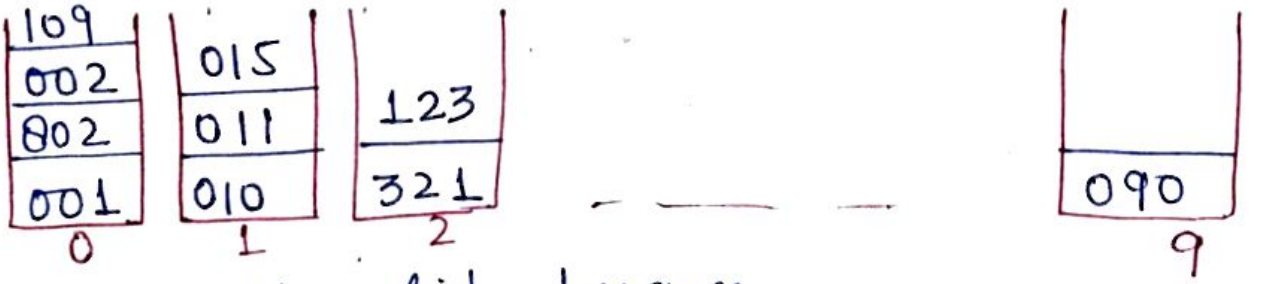
Pass 1 :- Consider LSB of all nos.



Now the list becomes,

010, 090, 001, 321, 011, 802, 002, 123, 015, 109

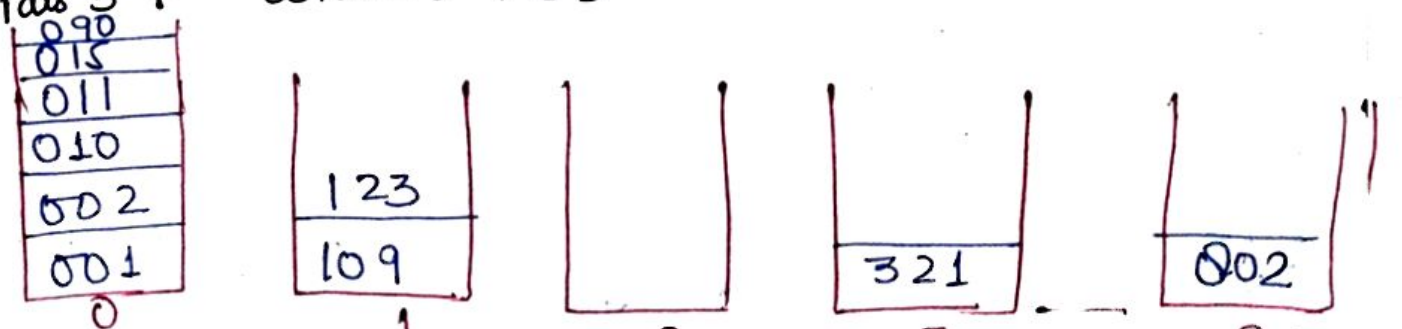
Pass 2 :- Consider II digit



Now, the list becomes,

001, 802, 002, 109, 010; 011, 015, 321, 123, 090

Pass 3 :- Consider MSB



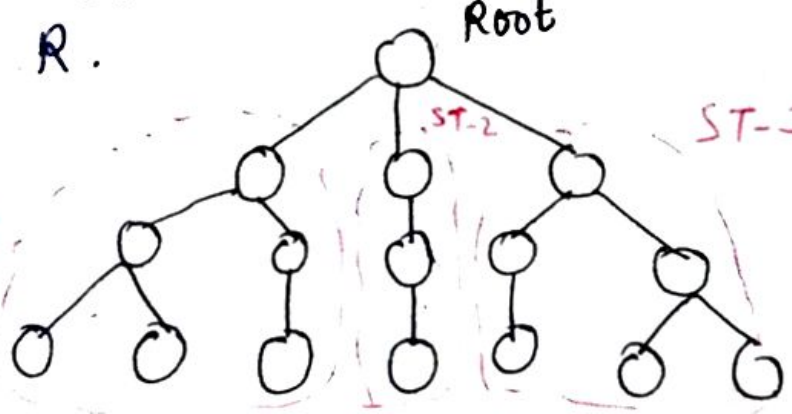
O/P: 1, 2, 10, 11, 15, 90, 109, 123, 321, 802 Sorted

Trees

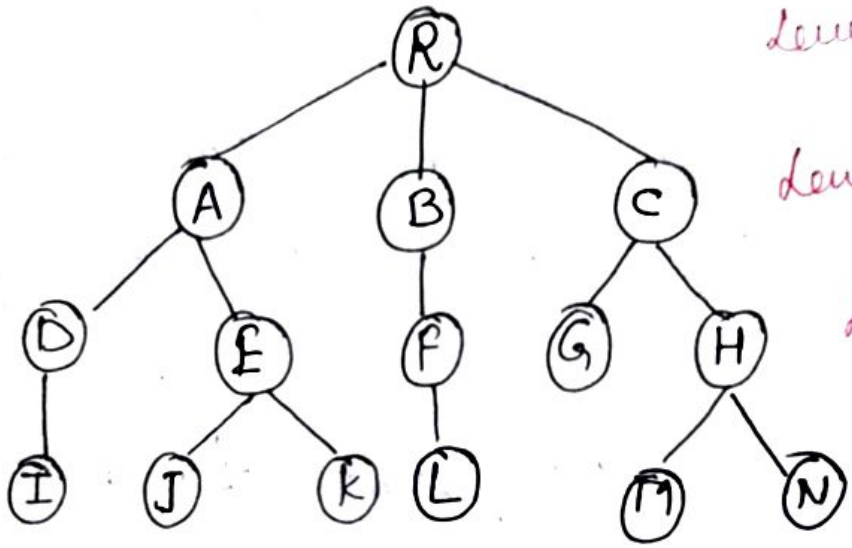
* A data structure is said to be, linear if its elements form a sequence or a linear list. Previous linear data structures that we have studied like arrays, stacks, queues & linked list organize data in linear order.
* Some data organizations require categorizing data into groups / subgroups. A data structure is said to be non-linear if its elements form a hierarchical classification where, data items appear at various levels.

Recursive def^{n} of trees :-

* A tree is a collection of nodes. The collection may be empty. Otherwise tree consists of distinguished node R called as the root node & zero or more non-empty subtrees $T_1, T_2 \ldots T_n$ each of whose roots are connected by an edge from R.



* If a tree contains n no. of nodes, out of which one is the root node, then there will be (n-1) no. of edges.



Tree Terminologies :-

* ① Siblings :- nodes with same parent are called siblings. e.g., A, B, C - Parent R, D, E - Parent A etc.

* ② Degree of node :- The no. of subtrees of a node in a given tree is called as the degree of that node. e.g., deg(R) = 3, deg(A) = 2

* ③ Degree of Tree :- The maximum degree of nodes in a given tree is called as the degree of the tree.

* ④ Leaf node / terminal node / External node :-

* A node whose degree is zero. i.e., node without a child. e.g; I, J, K, L etc.

* ⑤ Non-leaf / Non-terminal / Internal node :-

* A node whose degree is not zero.

* ⑥ Path :-

A path is a sequence of consecutive edges from the source node to the destination node.

Path from R to M

R → C → C → H , H → M (Path length = 3)

* There exists only one path from one source to a destination. otherwise it is not a tree.

* ⑦ Path Length :- Path length is determined by the no. of edges betn two nodes.

* ⑧ Depth of a Node :-

* depth of a node n is the length of the unique path from the root node to the node.

e.g., depth of (I) = 3
depth of E = 2

* In a tree, the depth of the root is always zero (0).

⑨ Height of a Node :-

Height of any node n is the length of the longest part from n to a leaf node.

Height of node (C) = 2
" " " (A) = 2

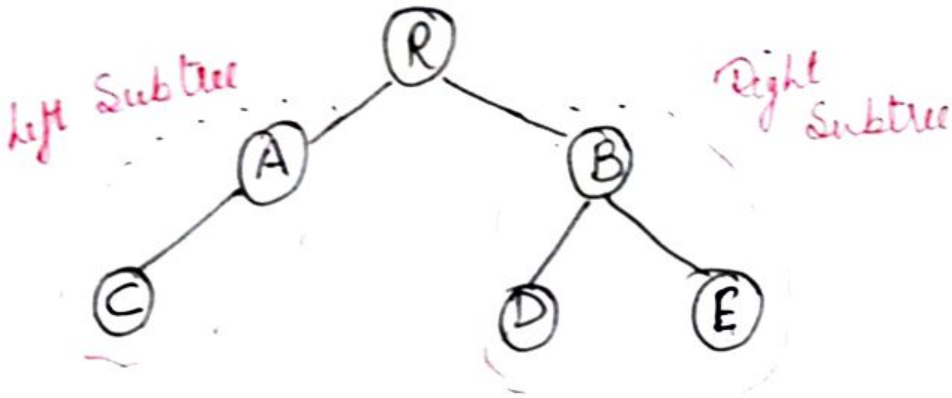
* Height of any leaf node = 0
* Height of a tree is same as Height of the root node.
* Height of a tree = Depth of a tree
* Depth of a node = level of the node.

⑩ Ancestor & Descendents :-

* if there is a path from node n_1 to node n_2, then n_1 is called as ancestor of n_2 & n_2 is called as the descendent of n_1.

(11) Binary Tree :-

* A Binary tree is a tree in which no node can have more than two children.
* Each node in binary tree contains either zero or 1 or max. two children.



Recursive Def^n for Binary Tree :

* A binary tree is a tree which is either empty otherwise tree contains a distinguished node called as Root & two non-empty binary Trees called as Left Subtree & Right Subtree.

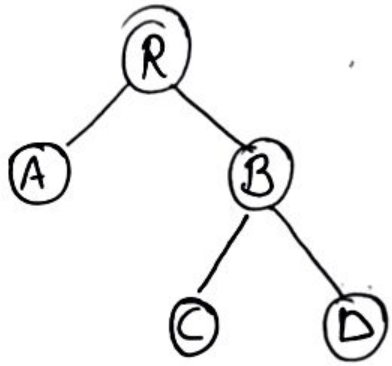
Strictly Binary Tree / Extended Binary tree / 2-tree :-

* If every non-leaf node in a Binary tree has non-empty left subtree & right sub tree, then the tree is called as strictly binary tree.

OR

* A Binary tree is called as strictly Binary tree if each node of the tree contains either zero or exactly 2 children.

Eg.)



Leaf nodes = 3

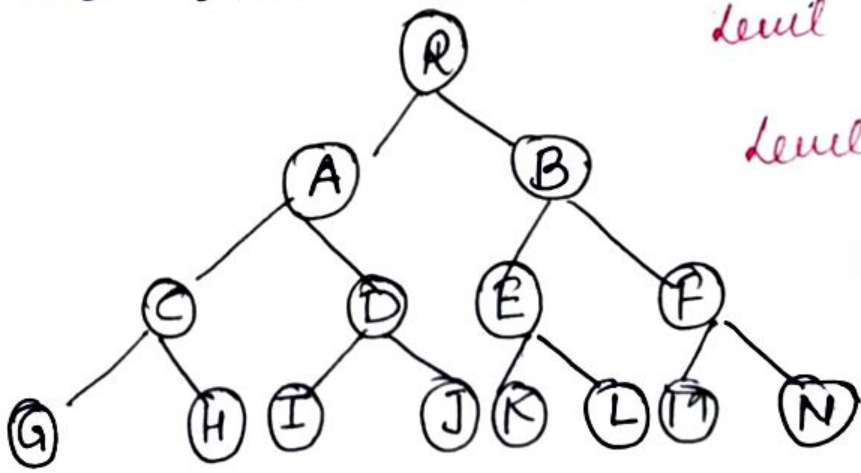
Total nodes = $2 \times 3 - 1$

$= 5$

* If a strictly Binary tree contains n leaf nodes, then the tree will have $(2n-1)$ no. of nodes

Complete Binary tree / Full tree :-

* A complete binary tree is a binary tree in which all internal nodes have degree 2 and all the leaf nodes are present at the same level!



* In a complete binary tree at any level i, there will be 2^i no. of nodes.
* In a complete binary tree if there are n no. of nodes at level l, then there will be $2n$ no. of nodes at level $l+1$.

Q. Find out the total no of nodes $t(n)$ of a complete binary tree of depth d.

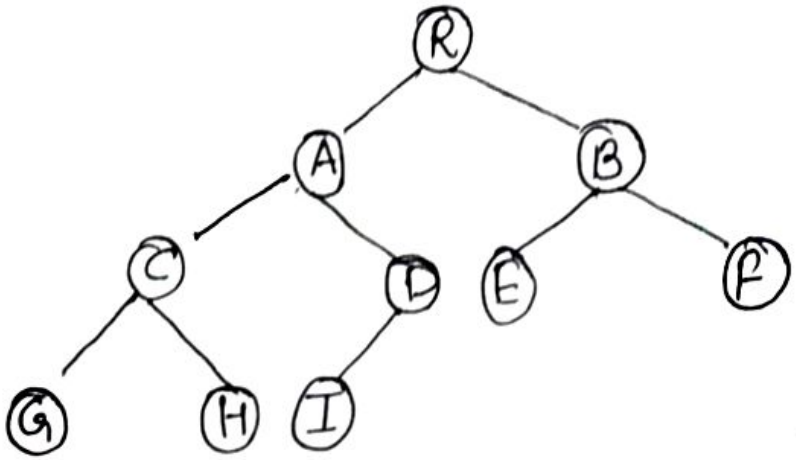
$$t(n) = 2^0 + 2^1 + 2^2 + \dots + 2^d$$

$$\boxed{t(n) = 2^{d+1} - 1}$$

Q If the complete binary tree contains t_n no. of nodes then its depth will be

$$\boxed{d = \log_2 (t_n + 1) - 1}$$

* **Almost complete Binary Tree :-**
* A binary tree is said to an almost complete B.T. if all its levels except the last level have maximum no. of possible nodes and all the nodes in the last level appear as far left as possible.



Level 0

Level 1

Level 2

Level 3

Q. Find out the minimum & the maximum no. of nodes of an almost complete binary tree the depth d of

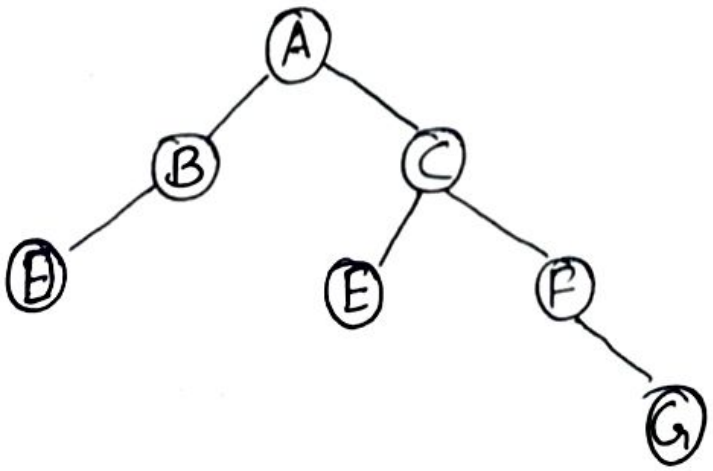
$$\text{Min } t_n = 2^d$$

$$\text{max } t_n = 2^{d+1} - 2$$

Representation of Binary tree in memory:-

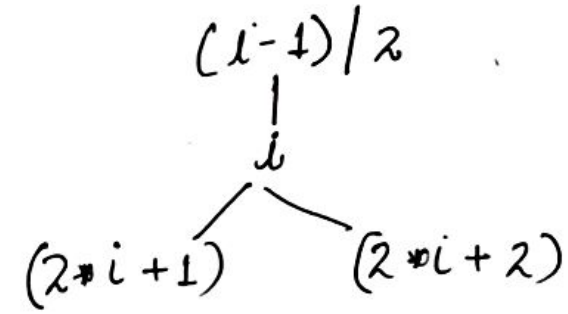
Two types of representation

1) Linear / Sequential representation using array
2) Linked representation using double linked list



(1) Linear / Sequential representation using array

* In sequential representation, if depth of the tree is d, then we need an array of size approximately 2^{d+1}.
* Always the root node of the tree will be stored at location 0.
* If any node is stored at location i, then its left child will be stored at $(2*i+1)$ location and its right child will be stored at $(2*i+2)$ and its parent will be stored at $(i-1)/2$ position.



Here, depth = Height = d = 3

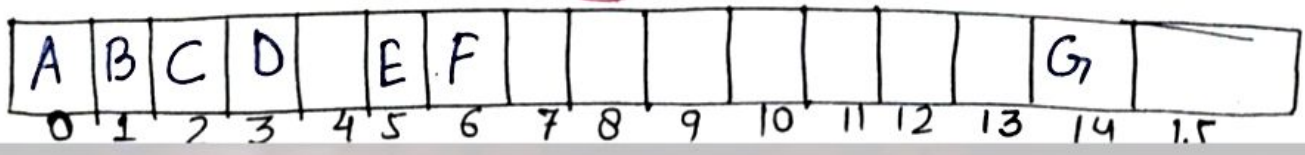
so, array size = $2^{3+1} = 16$

for the above example;

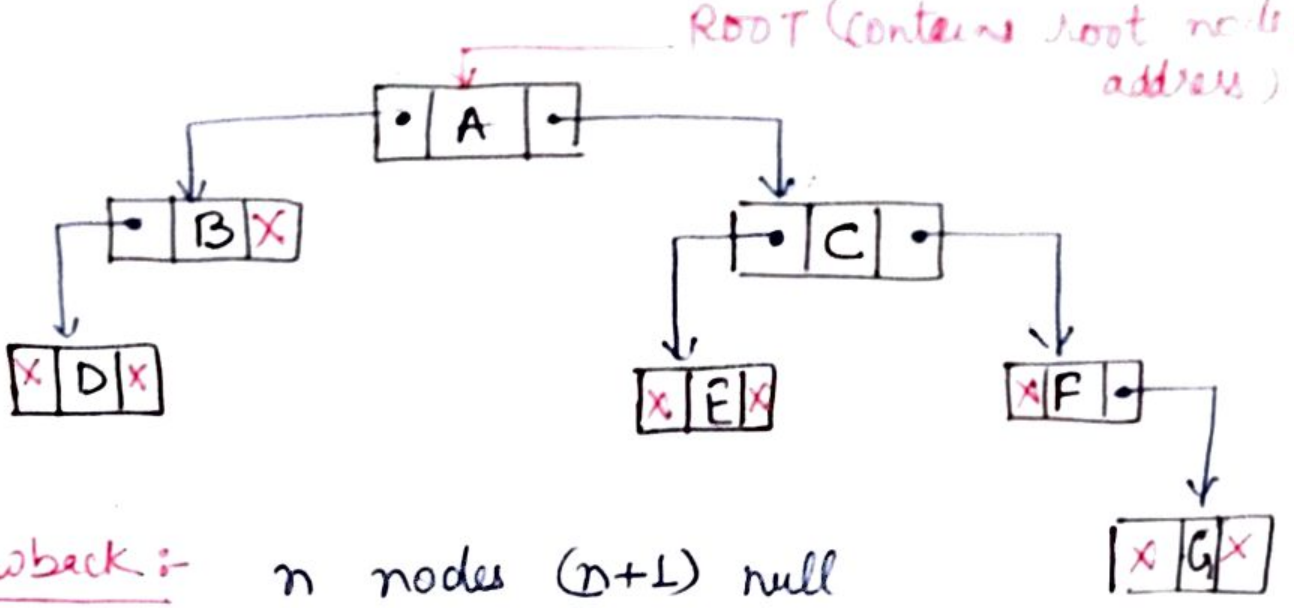
A = 0, B = 2*0 + 1 = 1, C = 2*0 + 2 = 2

D = 2*1 + 1 = 3, E = 2*2 + 1 = 5, F = 2*2 + 2 = 6

G = 2*6 + 2 = 14



② Linked Representation using double linked list :



Drawback :- n nodes (n+1) null pointers.

Traversing a Binary Tree

Three standard ways of traversal :

1) Pre-order traversal
2) In-order ,,
3) Post-order ,,

* Recursive algo for Pre-order traversal :

Step 1 : Process the root node
Step 2 : Traverse the left subtree in pre-order.
Step 3 : Traverse the right subtree in preorder.

* Recursive algo. for In-order Traversal :-

Step 1 :- Traverse the left subtree in In-order.

Step 2 :- Process the root node.

Step 3 :- Traverse the right subtree in In-order.

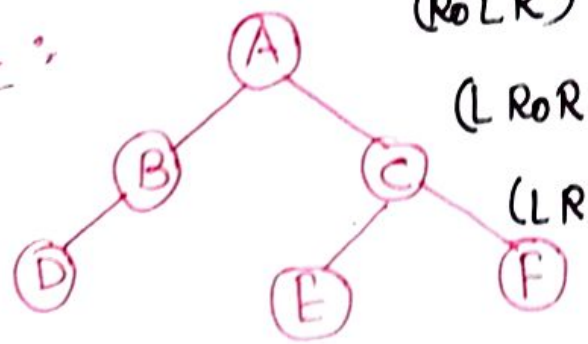
* Recursive algo. for Post-order Traversal :-

Step 1 : Traverse the left subtree in Post-order.

Step 2 : Traverse the right subtree in Post-order.

Step 3 : Process the root node.

Ex 1 :

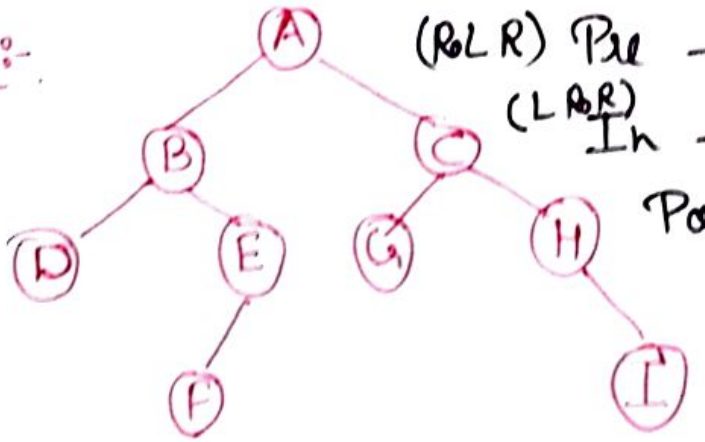


(RoLR) Preorder - A B D C E F

(LRoR) In-order - D B A E C F

(LRR) Post-order - D B E F C A

Ex 2 :-

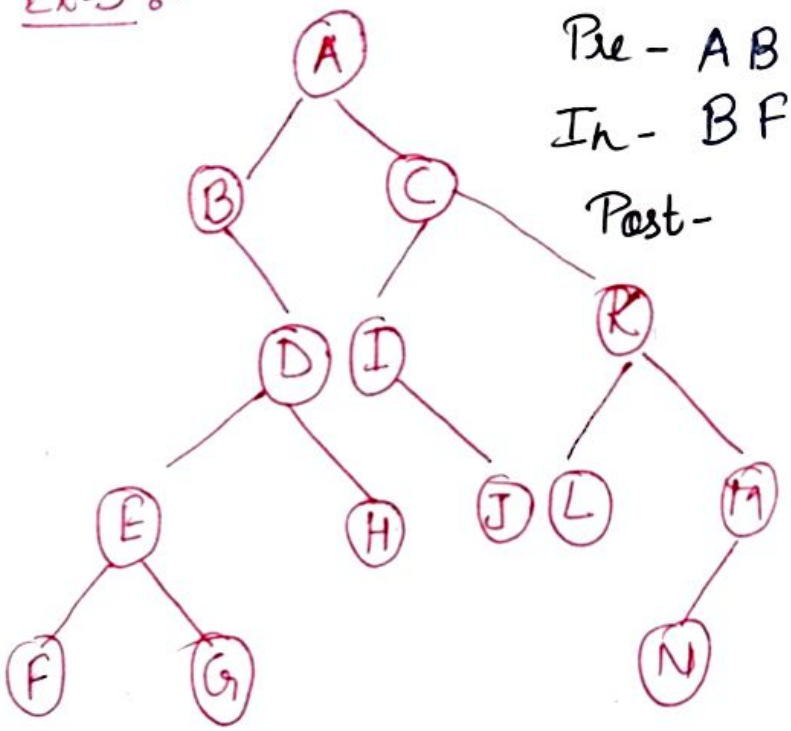


(RoLR) Pre - A B D E F C G H I

(LRoR) In - D B F E A G C H I

Post - D F E B G I H C A

Ex-3 :-

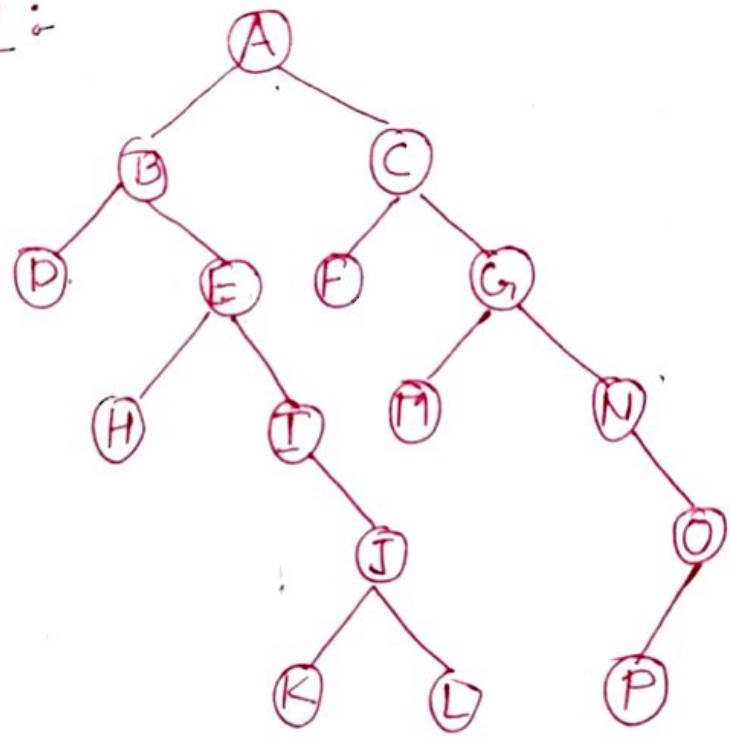


Pre - A B D E F G H C I J K L M N

In - B F E G D H A I J C L K N M

Post -

Ex 4 :-



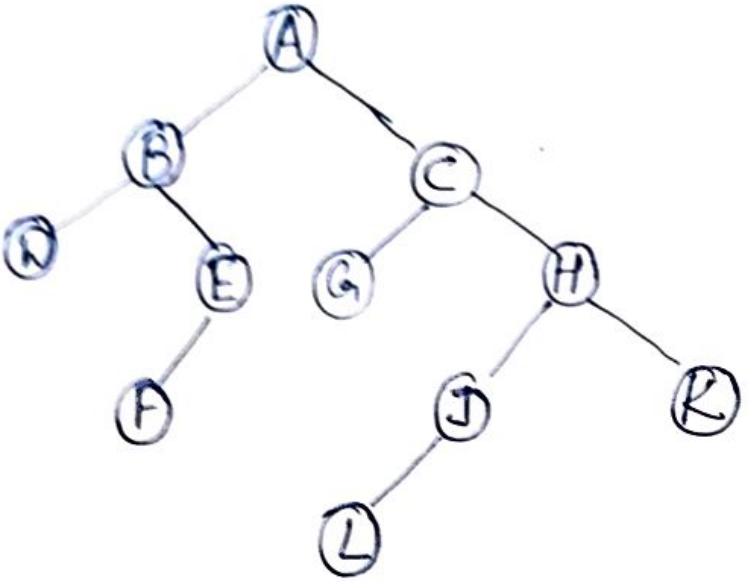
In-order : B H E I K J L A F C M G N P O

Post-order : H K L J I E B F M P O N G C A

Q: Pre-order : A B D E F C G H J L K (Ro L R)

In-order : D B F E A G C L J H K

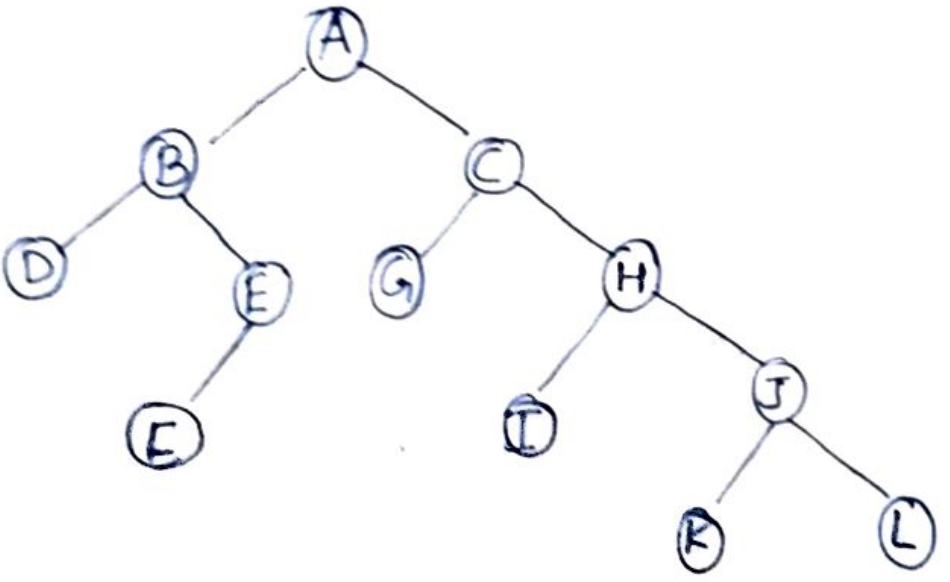
Left — Ro — Right



Q: Pre-order : A B D E F C G H I J K L (Ro L R)

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

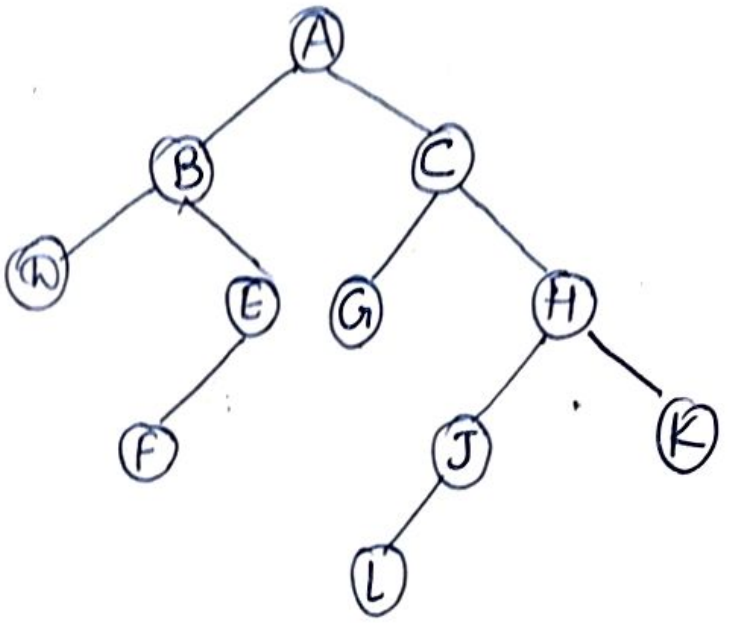
L — Ro — R



Q: Post : D F E B G L J K H C (A) (L R Ro)

IN : D B F E (A) G C L J H K (L Ro R)

L R



Note :- When post-order & pre-order traversal of a binary tree is given, then the constructed binary tree is not a unique tree.

Binary Search Tree (BST)

* A binary tree T is said to be BST or Binary Sorted Tree or if each node n of 'T' satisfies the following property :-

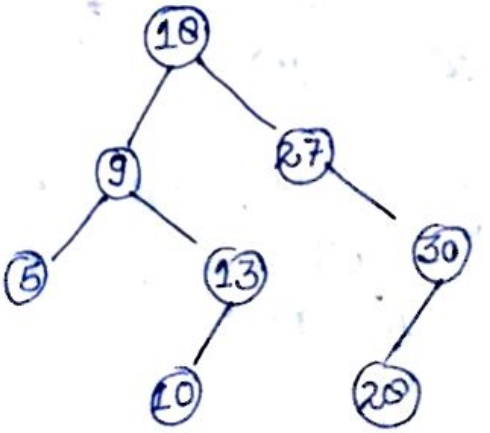
* The value at n is greater than every value of its left sub-tree & is less than or equal to every value of its right subtree.

Binary Search Trees

* A Binary Tree is a B.S.T. if it (every node x,) fulfills the following :-

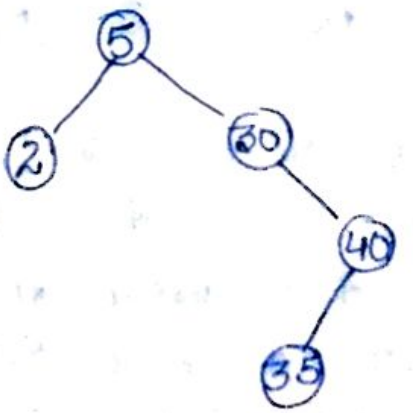
(i) the value of left-subtree of 'x' is less then the value at 'x'.

(ii) the value of right-subtree of 'x' is greater than the value at 'x'.



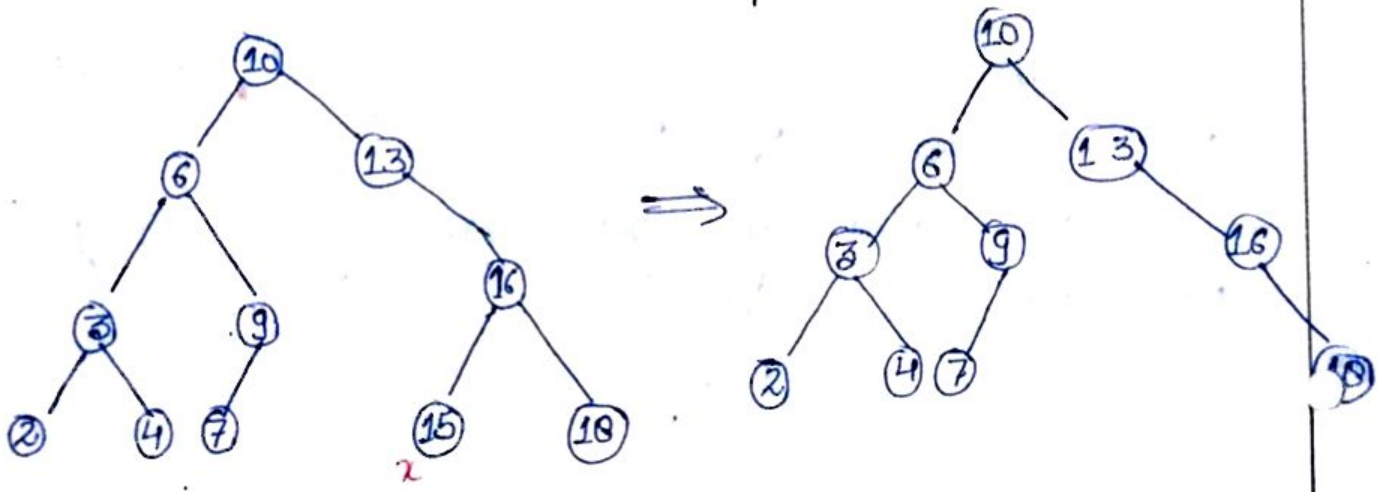
Insertion in B.S.T. :- Suppose T is an empty B.S.T. & now we have to insert the following data items.

5, 30, 2, 40, 35



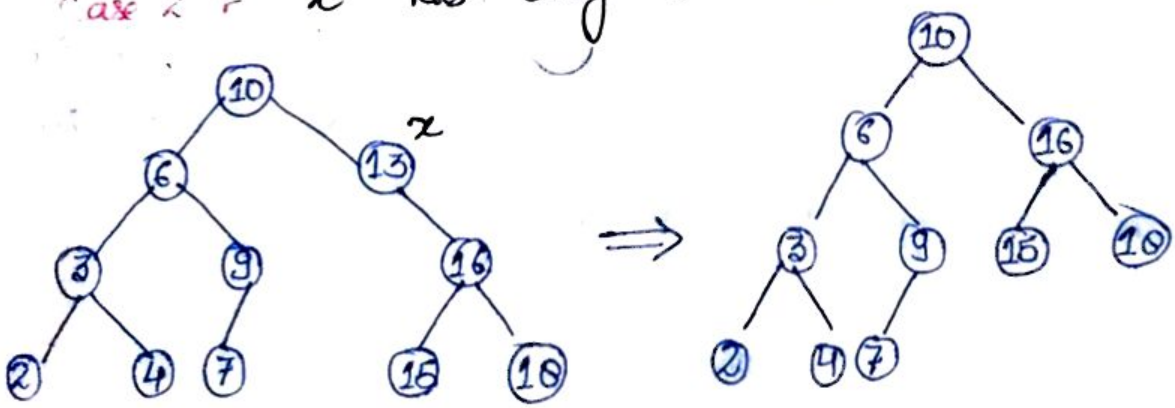
Deletion in B.S.T. :- There are 3 cases that arises, when we delete 'x' from B.S.T.

Case 1 :- 'x' is a leaf node.



* If 'x' is left child, set its parent's left pointer to NULL. otherwise, set its parent's right pointer to NULL.

Case 2 :- 'x' has only one child

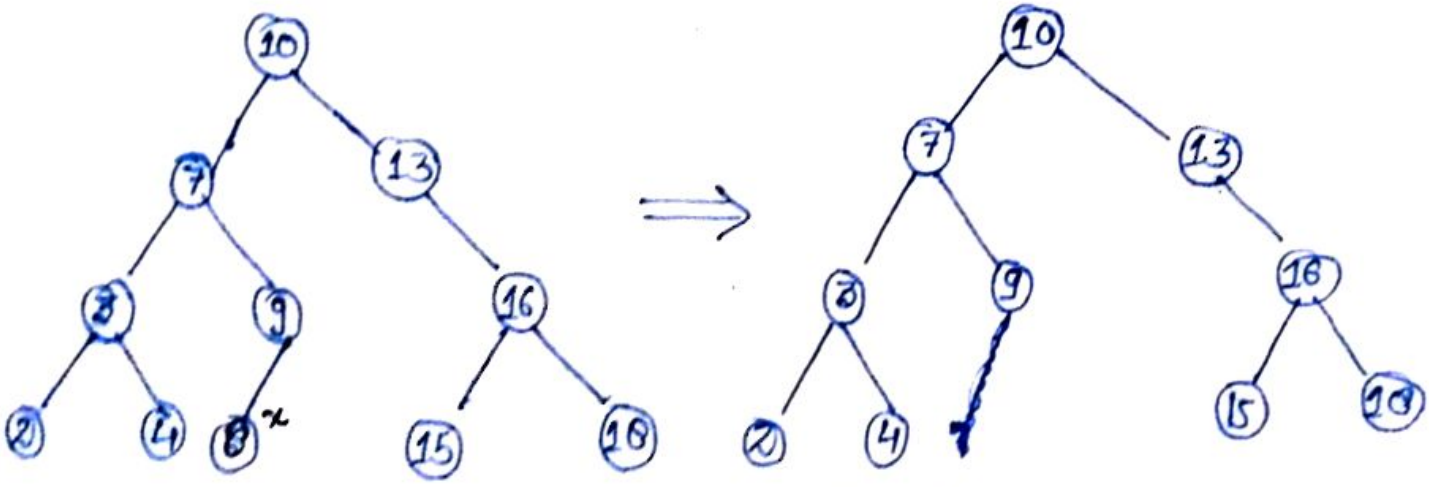


* If 'x' points to a node that has a right child then 'x' right child pointer is assigned to the right child value of its parent but if 'x' points to a node that has a left child then 'x' left child pointer value is assigned to the left child value of its parent.

Case 1 : 'x' has both the children.

* (i) Find Successor of 'x'.
  (ii) Interchange the values of 'x' with its successor.
  (iii) Delete the Successor. (Case 2:- As Successor of a node can have at most a single child.

For the given tree if we delete '6', its successor is '7'.



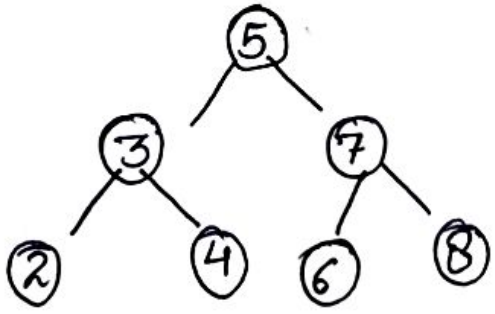
Resulting Tree

* Search trees are data structure that support many dynamic-set operations, including Search, Minimum, Maximum, Predecessor, Successor, Insert & Delete.

* A B.S.T. is a binary tree which is represent by a linked data structure. Each node contains Info, pointers to left, right & parent (p).

* The Info in BST are always stored in such a way as to satisfy the BST property :

  * the value at every node N is greater than every value in the left subtree of N & is less than every value in the right subtree of N.



Operations on B.S.T. :

1) Searching :

* In the following procedure given is a pointer to the root of the tree & a key k.

```
Tree-Search (x, k)
1. if x = NIL or k = Info[x]
2.     then return x
3. if k < Key[x]
4.     then return (Tree-Search (left[x], k))
5.     else return (Tree-Search (right[x], k))
```

2) Minimum & Maximum :

```
Tree-Minimum(x)
1. while (left[x] ≠ NULL)
2.     do x = left[x]
3. return x
```

```
Tree-Maximum (x)
1. while right[x] ≠ NULL
2.     do x = right[x]
3. return x
```

Insertion & Deletion :

③ Insertion

To insert a new value v into a BST T, we use the procedure Tree-Insert. The procedure is passed a node z for which key[z] = v, left[z] = right[z] = NULL. It modifies T in such a way that z is inserted into an appropriate position in the tree.

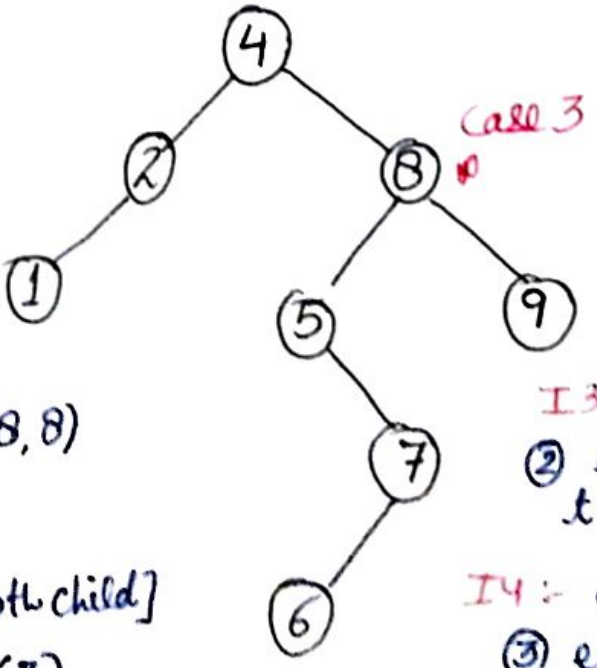
C- implementation for BST :- (using linked list)

```
struct node {
    struct node * left;
    int info;
    struct node * right;
}
typedef struct node * bstnode;

bstnode insert ( bstnode t, int x)
{
    if ( t == NULL)
    {
    t = (bstnode) malloc (sizeof(struct node));
      if (t == NULL)
      { printf ("Out of memory");
        return t;
      }
      else
      {
        t -> info = x
        t -> left = t -> right = NULL;
      }
    }
    else
    {
        if (x < t -> info)
            t -> left = insert(t -> left, x);
```

```
    else
    {
        if (x > t -> info)
            t -> right = insert (t -> right, x);
        }
    }
     return t;
}

bstnode delete ( bstnode t, int x)
{
    bstnode temp;
    if (t == NULL)
      printf(" BST is empty");
    else
    {
    ① if (x < t -> info)
            t -> left = delete (t -> left, x);
      ① else
        {
        ② if (x > t -> info)
            t -> right = delete ( t -> right, x);
        ② else // element found
          {
          ③ if (t -> left && t -> right) // both children
            {
              temp = findmax (t -> left);
              t -> info = temp -> info;
              t -> left = delete (t -> left, t -> info);
            }
```

```
③ else    // one or no child
  {
     temp = t ;
     ④ if (t -> left == NULL)
            t = t -> right ;
     ④ else
            if ( t -> right == NULL)
                  t = t -> left ;
            free (temp;
        }
  } }
  return t ;
} // end of delete ()
```

Example :



Case 3

I1:- delete (4, 8)
4<8 ; t -> right = delete(8, 8)

I2 : delete (8,8)
② else ③ if [as both child]
temp = add (7)
t -> data = 7
t -> left = delete (6, 7)

I3 :- delete (5, 7)
② else
t -> right = delete(7, 7)

I4 :- delete (7, 7)
③ else
temp = 7
④ else t = t -> left (6)

⑤ Successor & Predecessor :-

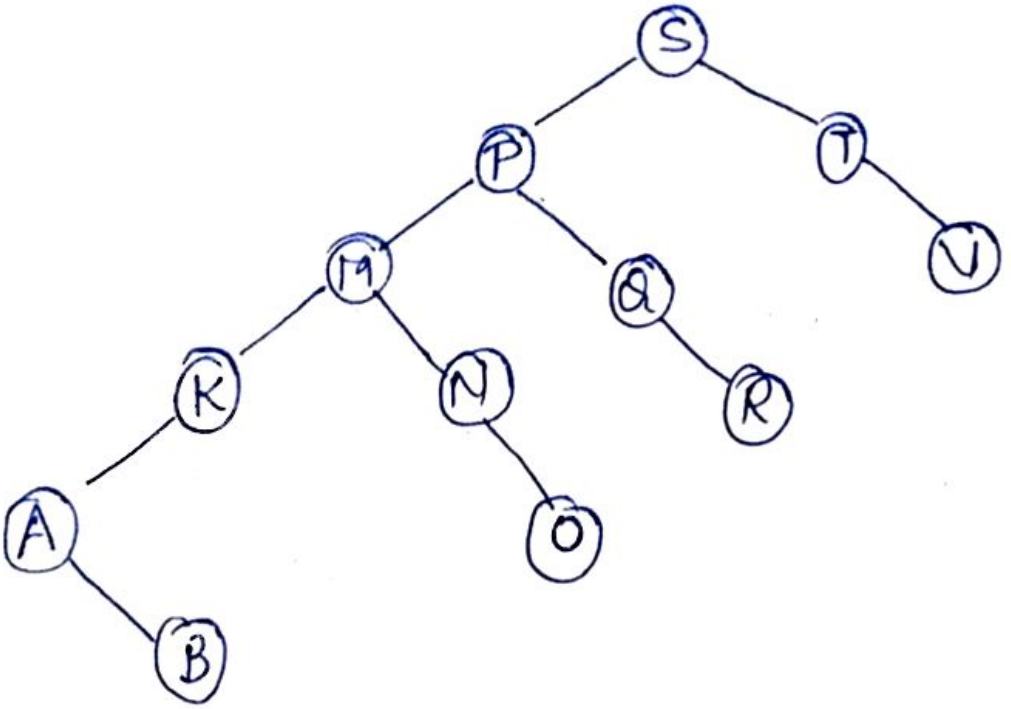
* Given a node in BST, it is sometimes important to be able to find its successor in the sorted order determined by an inorder walk.
* If all the keys are distinct, the successor of a node x is the node with smallest key greater than key [x].

```
Tree - Successor (x)
1.  if right [x] ≠ NULL
    then return (Tree - Minimum (right [x]))
2.
3.      y = p [x]
4.      while y ≠ NULL & x = right [y]
5.          do x = y
6.             y = p [y]
7.      return y
```

(lines 4–6: while-loop)

Q. Suppose the following list of letters is inserted in an empty BST.

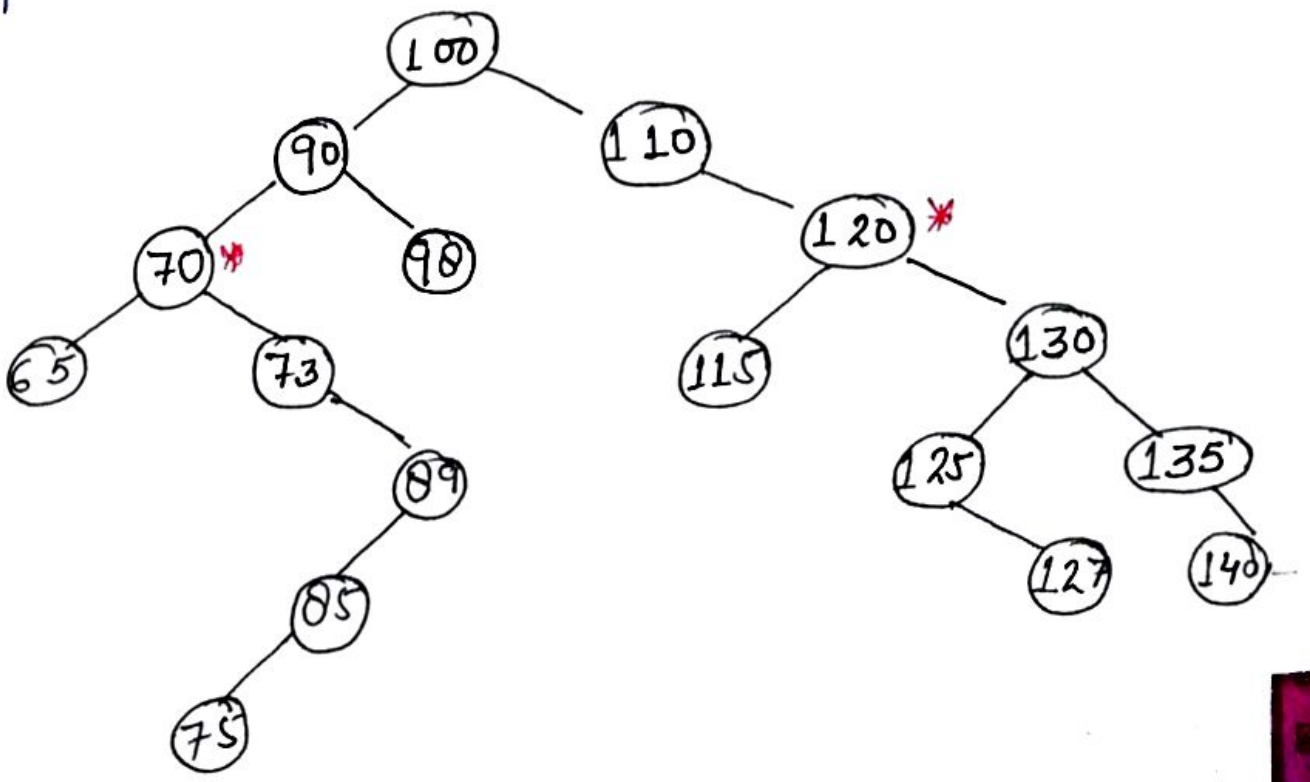
S, T, P, Q, M, N, O, R, K, V, A, B



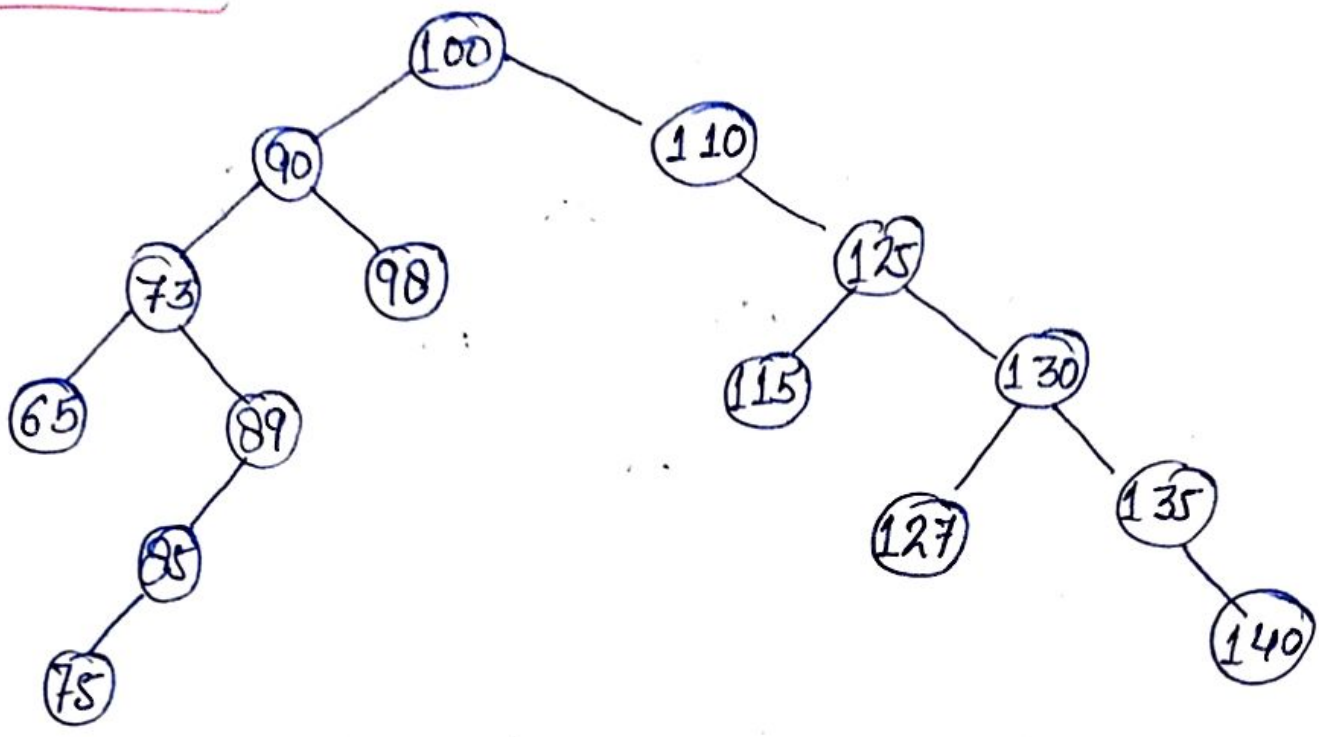
Q construct a BST with the following elements :-

100, 90, 110, 120, 70, 65, 73, 89, 130, 125, 135, 127, 140, 98, 85, 75, 115

then delete :- 120 & 70 from constructed BST



Resultant tree is :-



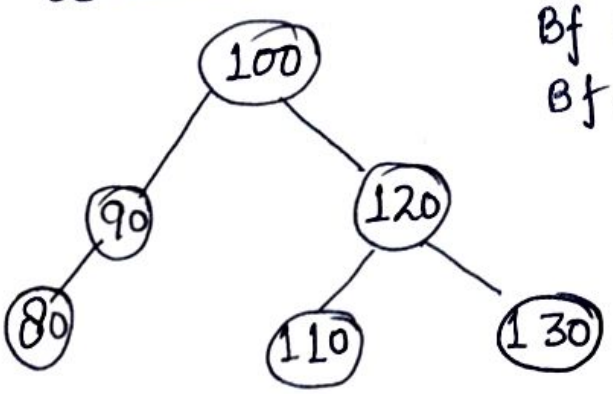
Height - Balanced Tree / AVL Trees :-

* One of the most popular balanced trees was introduced in 1962 by Adelson-Velskii and Landis and was known as AVL trees

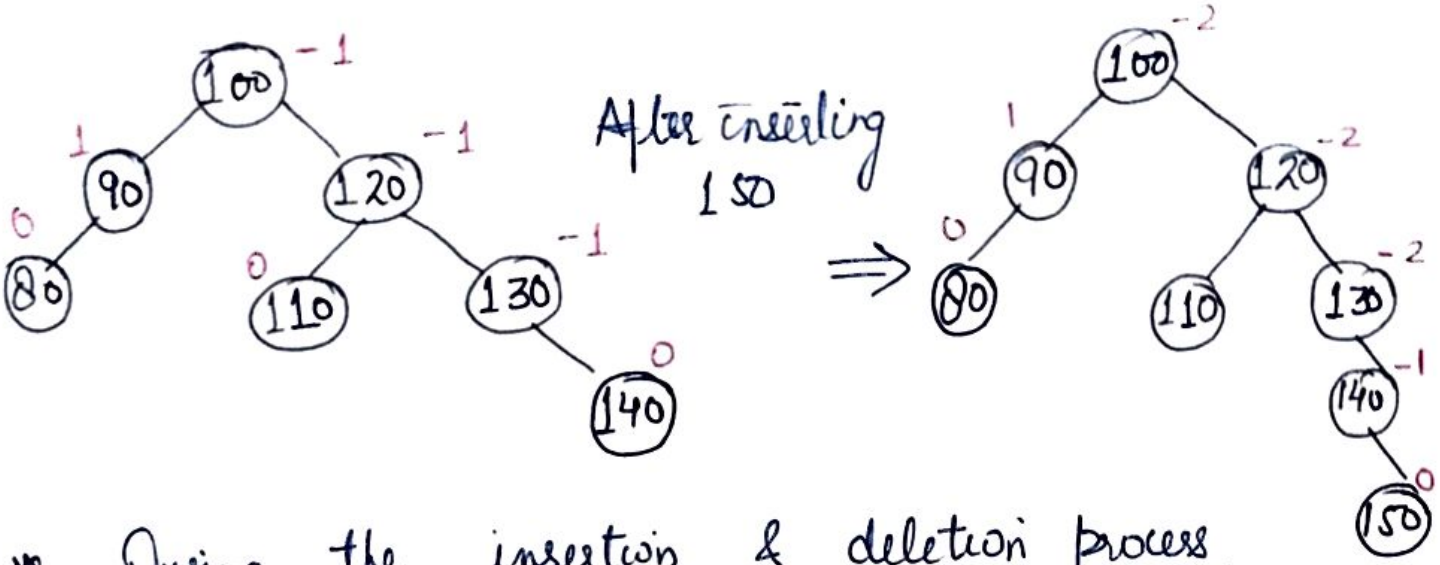
Definition :- Let T be a non-empty binary tree. T_L and T_R be the left subtree and right subtree of T. $H(T_L)$ and $H(T_R)$ be the height of left & right subtree respectively, then T will be called as an AVL Tree if,

$$|H(T_L) - H(T_R)| \leq 1$$

* $H(T_L) - H(T_R)$ is called as the Balance factor.
* If an AVL tree, every node balance factor is either 0, 1 or -1.
* An AVL tree is also a BST but with a balanced condition.



Bf (100) = 2-2 = 0
Bf (90) = 1-0 = 1



* During the insertion & deletion process, if some node's balance factor will be changed to 2 or -2 from 0, 1 or -1 then the AVL tree becomes unbalanced.

Rotation :- It is a technique to convert any unbalanced AVL tree to balanced AVL tree.

There are 4 types of rotation :-

(1) LL :- Inserted node is in the left subtree of left subtree of the reference node.

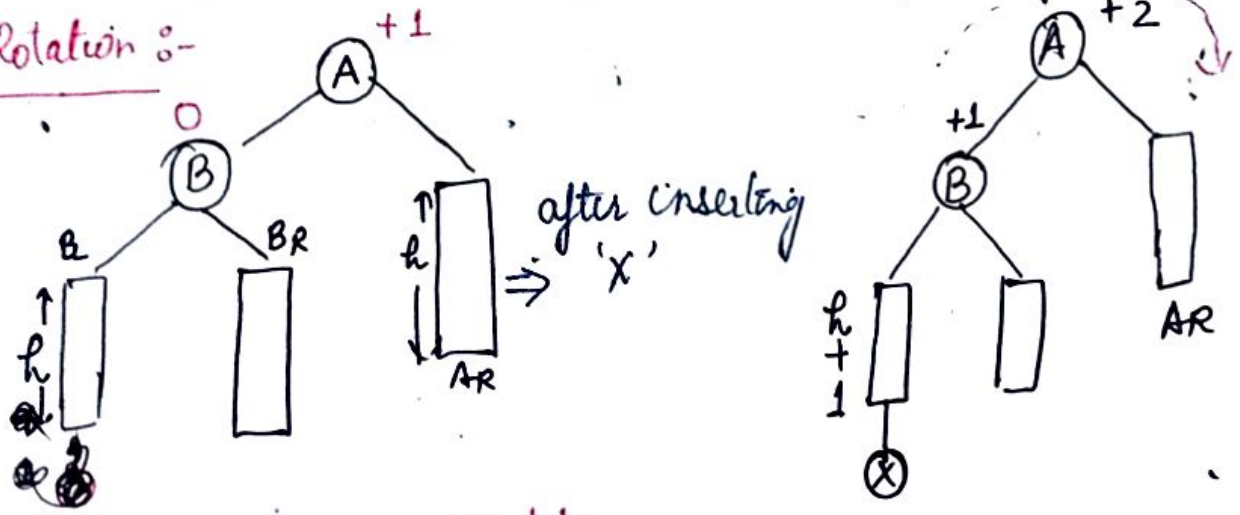
(2) RR :- Inserted node is in the right subtree of right subtree of the reference node

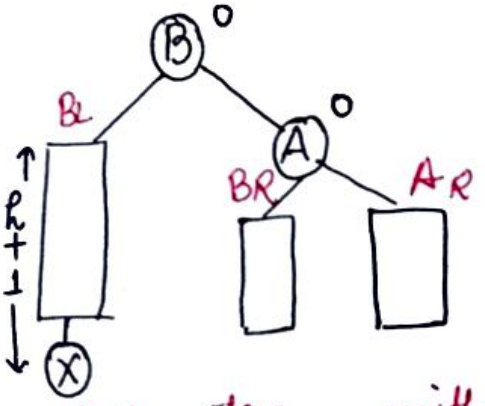
(3) LR :- Inserted node is in the right subtree of the left subtree of reference node.

(4) RL :- Inserted node is in the left subtree of the right subtree of reference node.

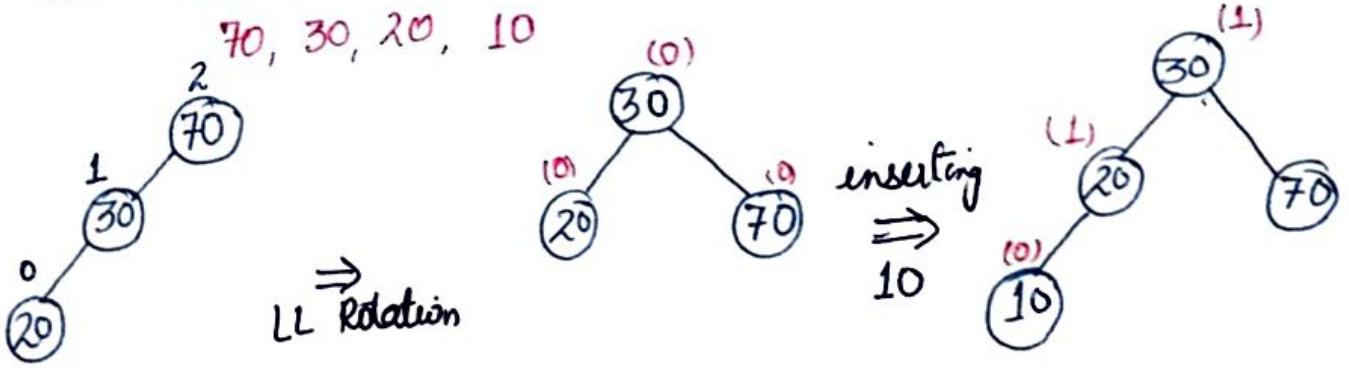
Reference Node :- is a node whose balance factor has been changed from 0, 1 or -1 to 2 or -2 and which is nearest to the insertion & deletion point.

LL Rotation :-

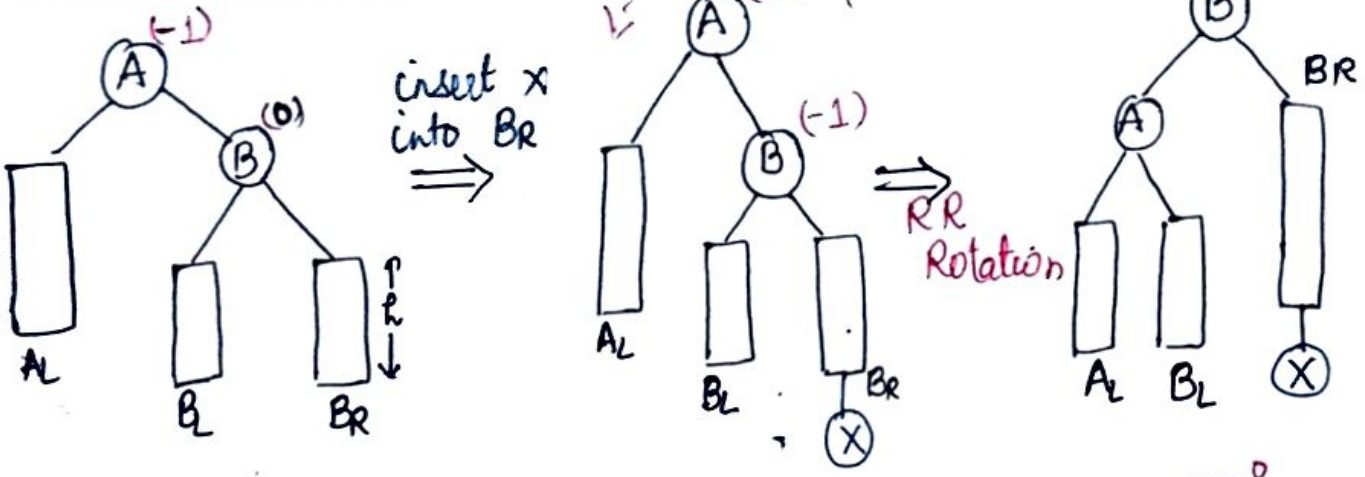




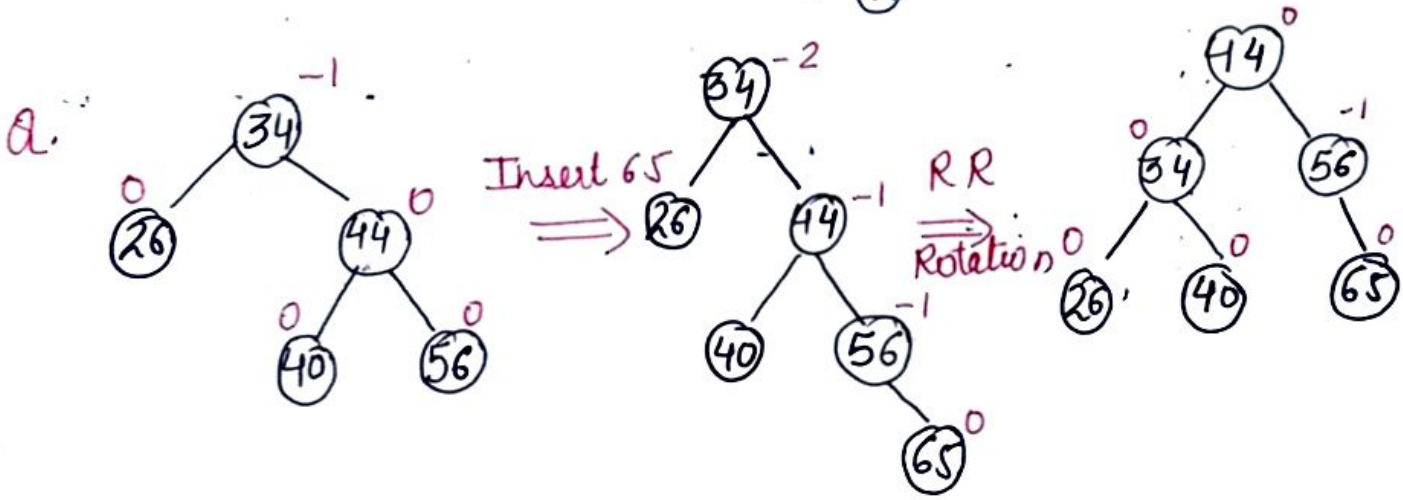
Q1. Create an AVL tree with the following keys-
70, 30, 20, 10



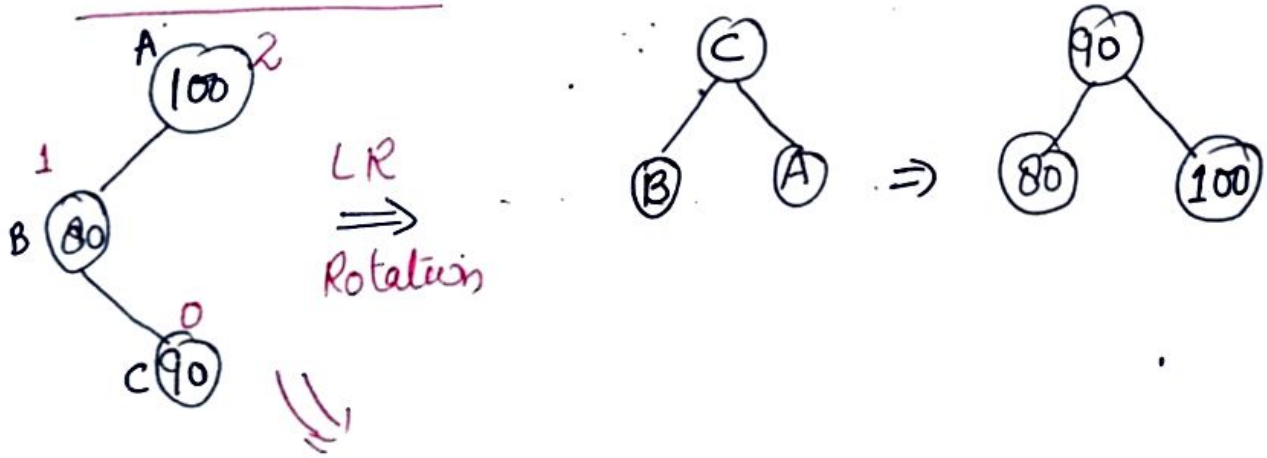
② RR Rotation :-



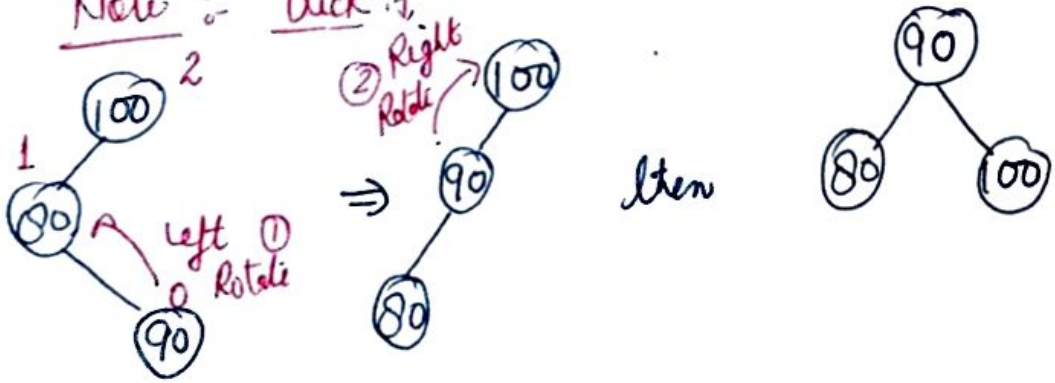
Q.



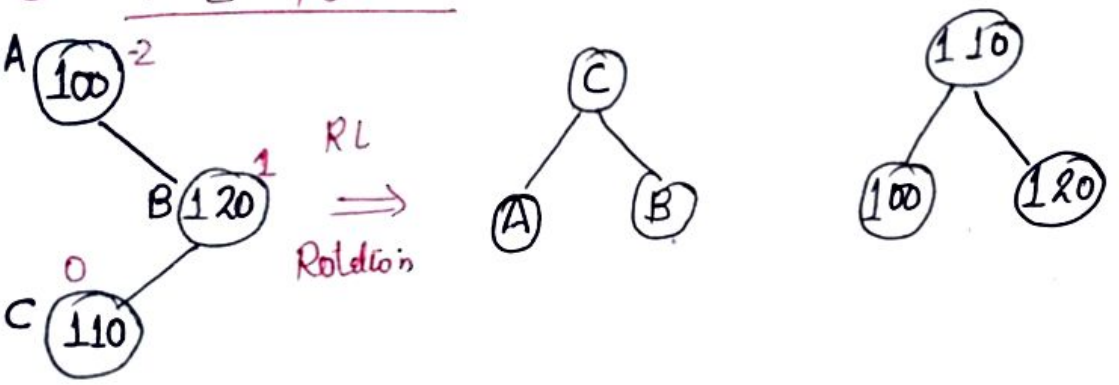
③ LR Rotation :-



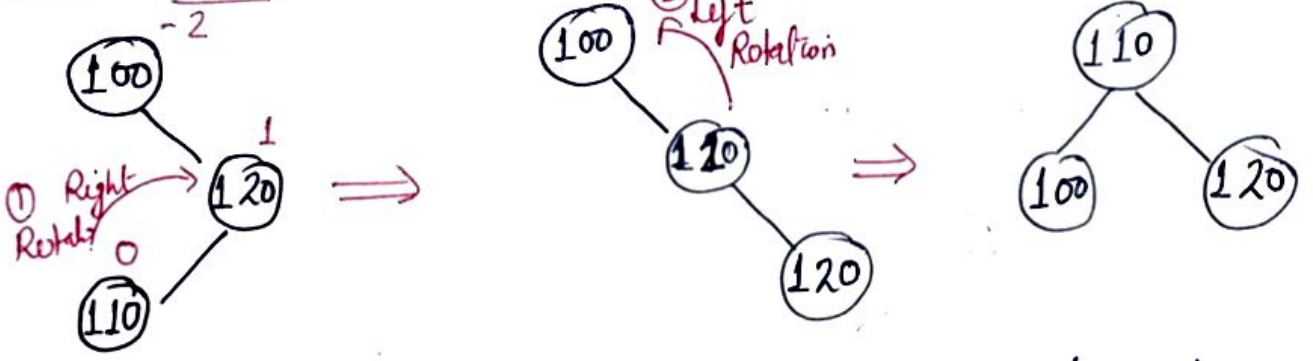
Note :- Trick :-



(4) RL Rotation :-



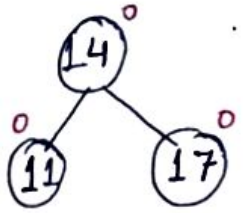
Note Trick :-



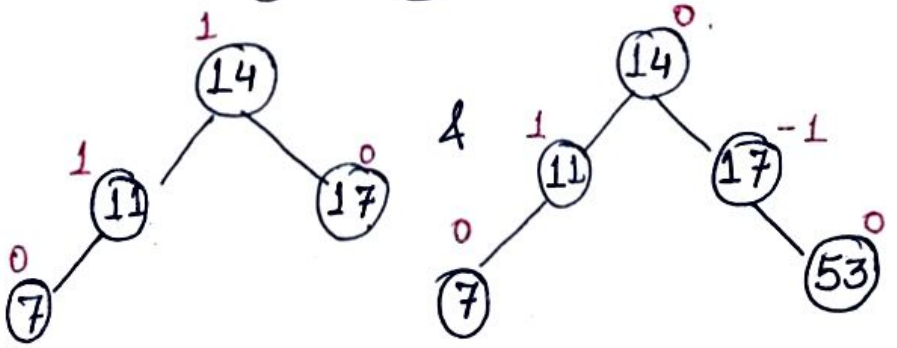
Question : construct an AVL search tree by inserting the following elements in the order of their occurrence

14, 17, 11, 7, 53, 4, 13, 12, 8, 60, 19, 16, 20

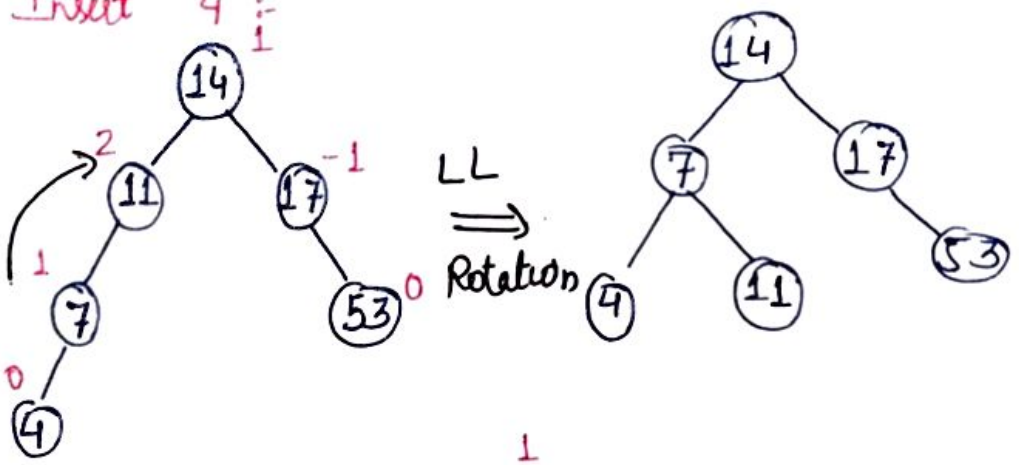
Insert 14, 17, 11



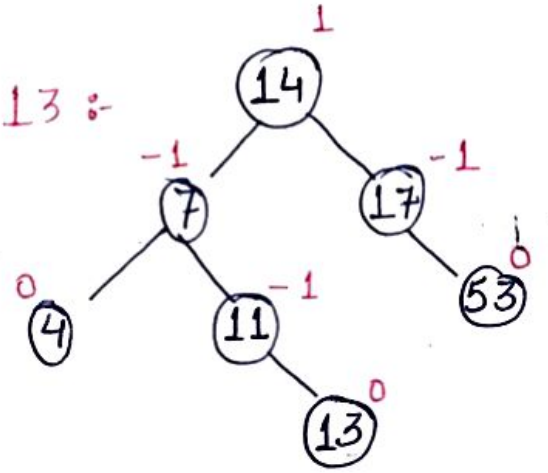
Insert 7, 53



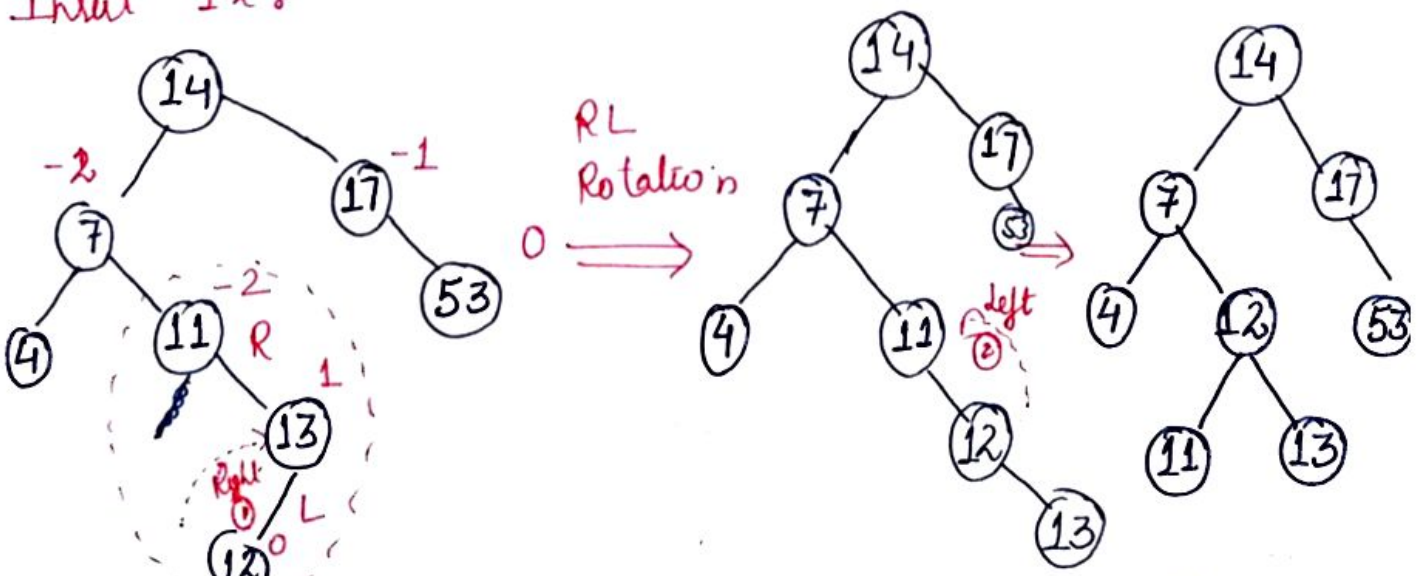
Insert 4 :-



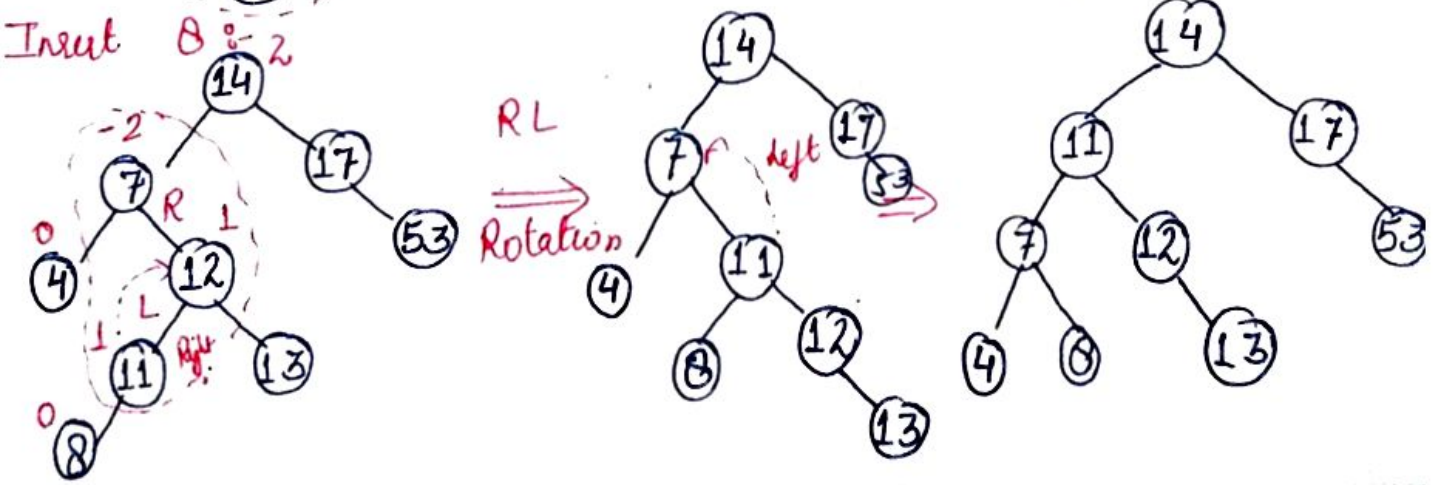
Insert 13 :-



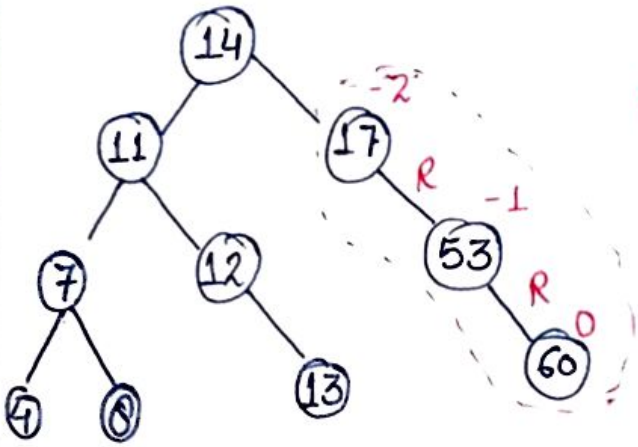
Insert 12 :-



Insert 8 :-

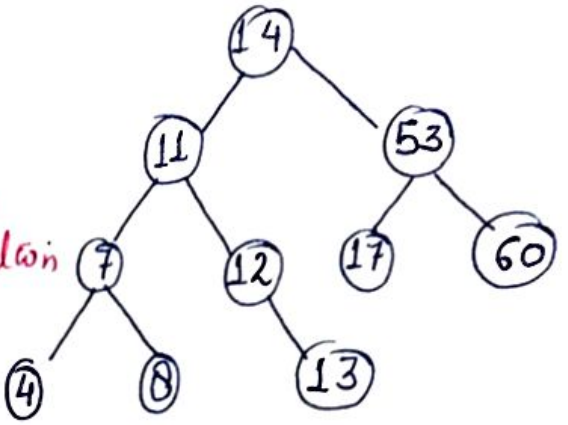


Insert 60 :-

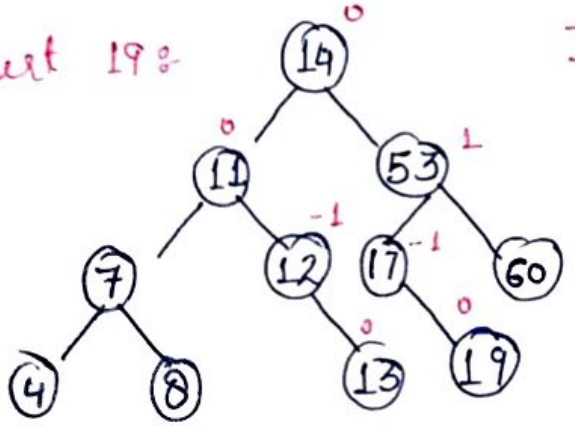


RR

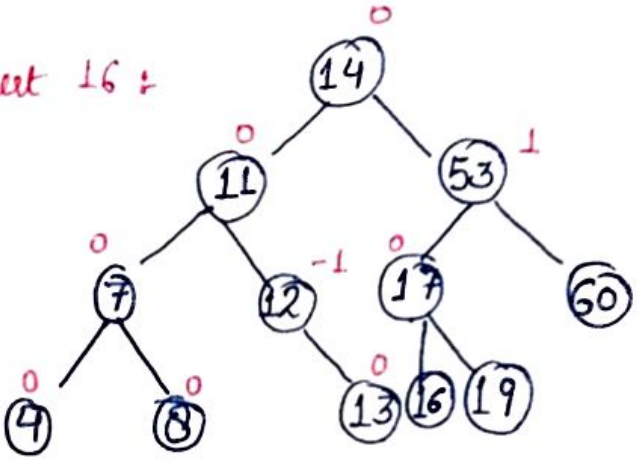
⇒ Rotation



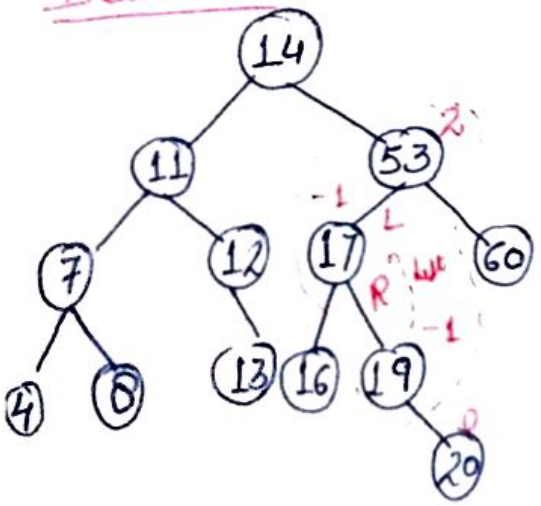
Insert 19 :-



Insert 16 :-

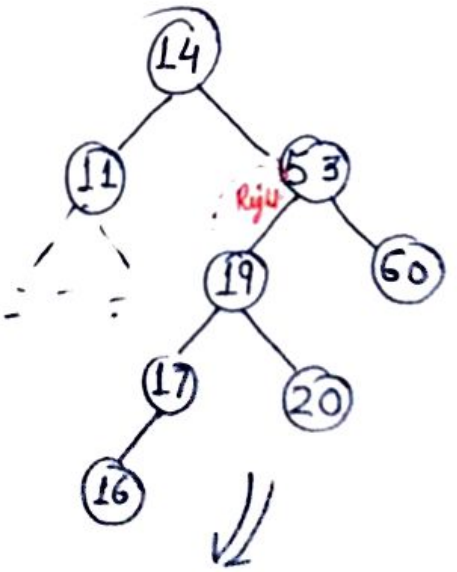


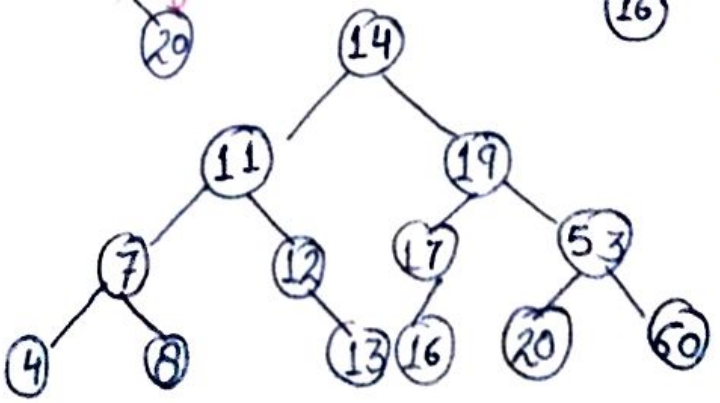
Insert 20 :-



LR

⇒ Rotation





Final AVL tree

Q : 64, 1, 44, 26, 13, 110, 98, 85

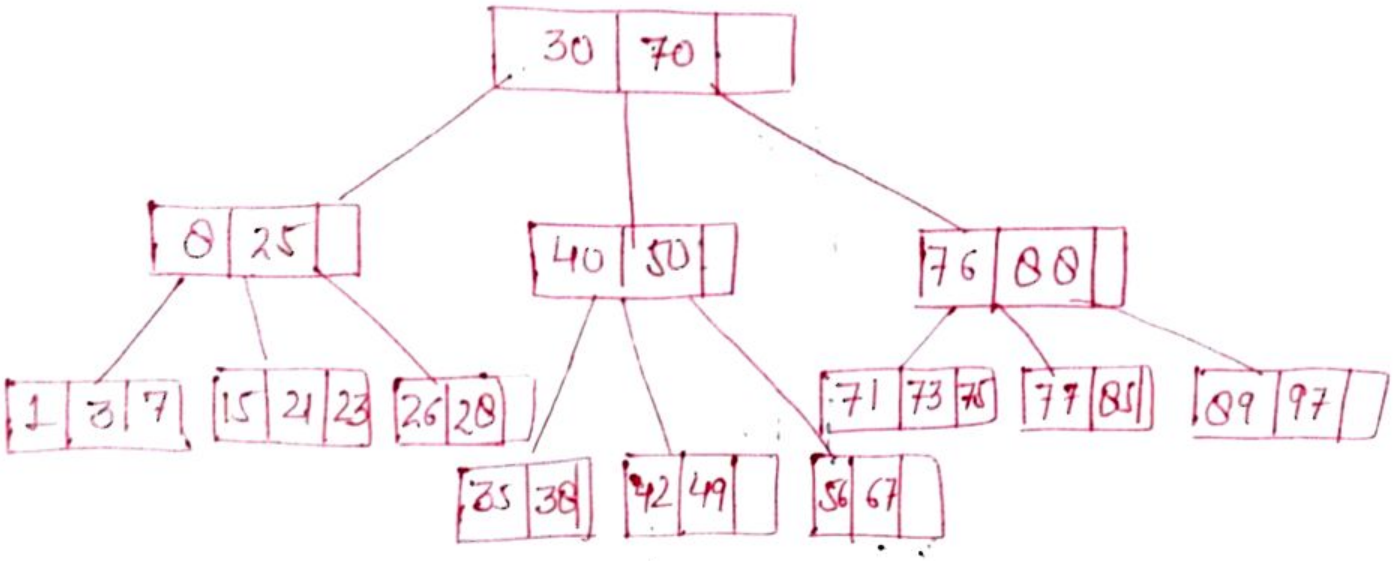
B-Tree & B+-Trees

* B-Trees are specialized m-way search tree. This can be widely used for disc access.
* A B-tree of order m, can have max (m-1) keys and m children.
* B-Tree is a special type of search trees in which a node contains more than one value (key) & more than two children.

B-tree of order 'm' has following properties:-

1) All leaf nodes must be at same level.
2) All nodes except root must have at least $(\lceil \frac{m}{2} \rceil - 1)$ keys and max of (m-1) keys.
3) All non-leaf nodes except root must have at least $m/2$ children.
4) Root node must have at least two children
5) A non-leaf node with (n-1) keys must have n no of children.
6) All the key values in a node must be in ascending order

for example :- B- tree of order 4 contains a maximum of 3 key values in a node and max. of 4 children for a node.



Note :

		max	min
Root	child	m	0
	data / key	m-1	1.
Internal node	child	m	$\lceil m/2 \rceil$
	key	m-1	$\lceil m/2 \rceil - 1$
Leaf node	child	0	0
	key		

Q Consider the following elements

5, 10, 12, 13, 14, 1, 2, 8, 4

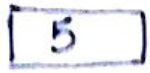
insert them into empty b-tree of order 3.

$m = 3$

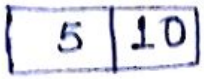
data → 3-1 = 2

child → 3

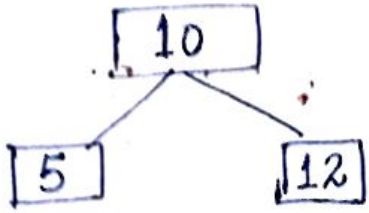
Insert 5 :

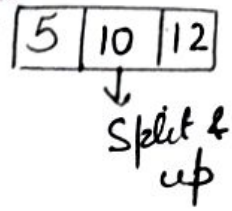


Insert 10 :

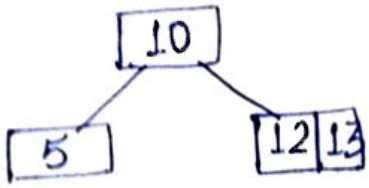


Insert 12 :

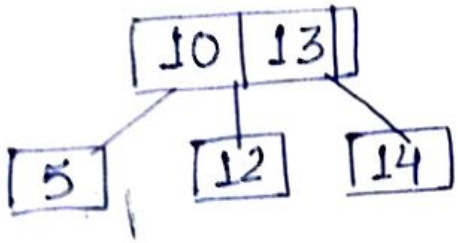


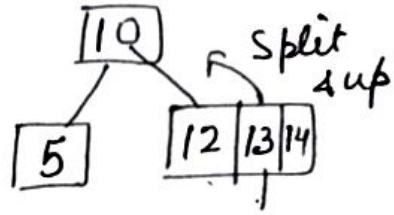


Insert 13 :

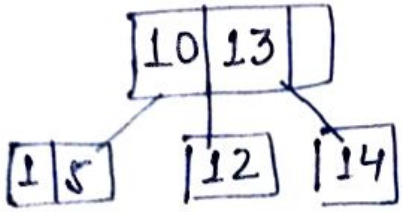


Insert 14 :

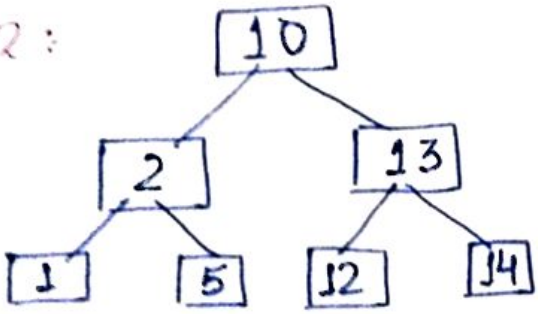


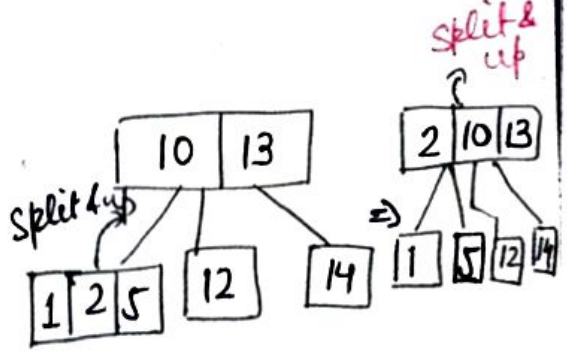


Insert 1 :

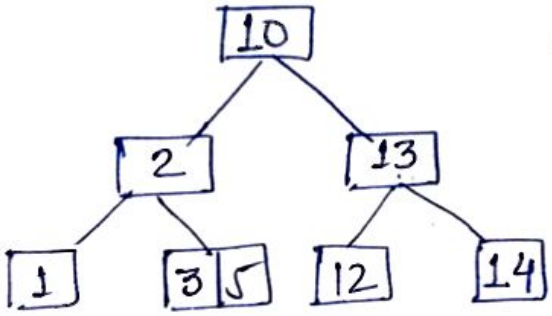


Insert 2 :

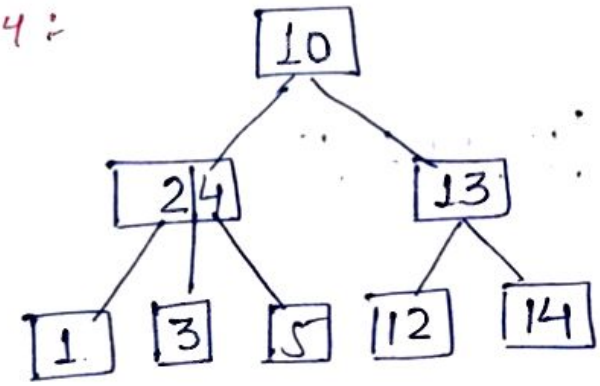


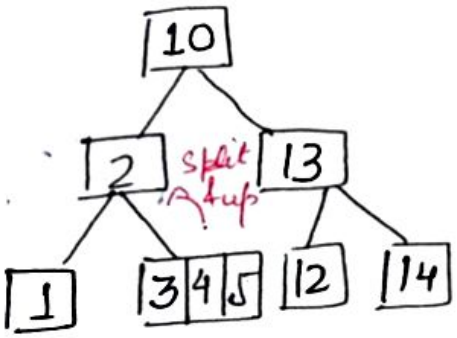


Insert 3 :



Insert 4 :





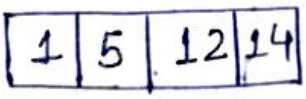
Q consider the following elements

5, 12, 14, 1, 2, 4, 18, 19, 17, 15, 25, 24, 22, 11, 30, 31, 28, 29, 13

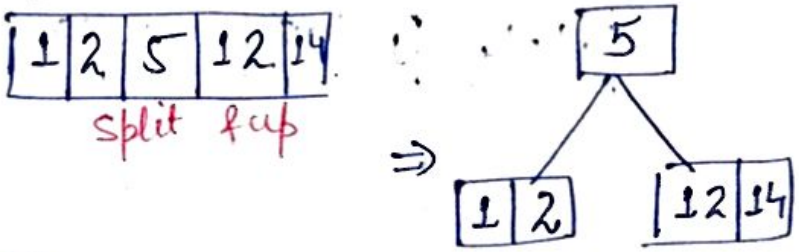
insert them into empty b-tree of order 5.

$m = 5, \quad D = 4$.

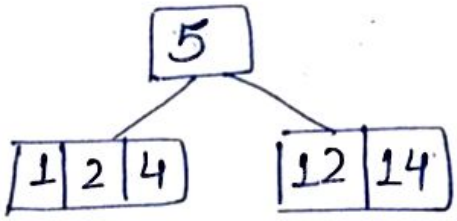
Insert 5, 12, 14, 1



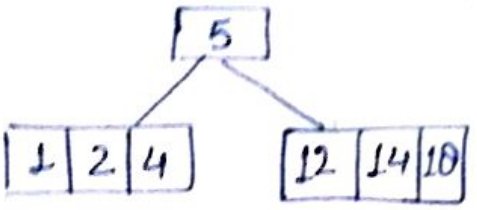
Insert 2 :-



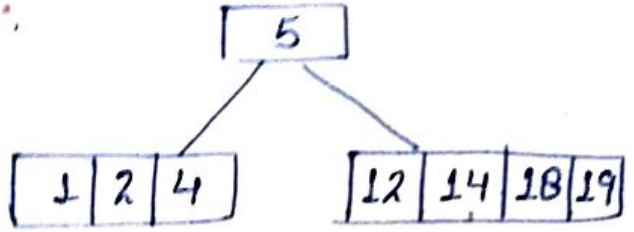
Insert 4 :



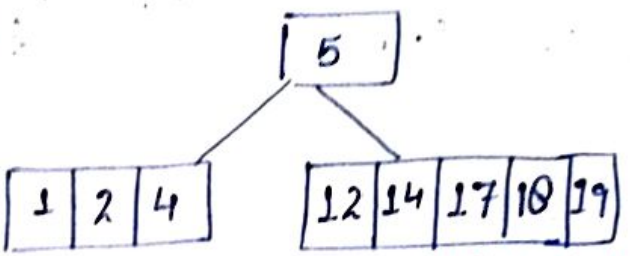
Insert 18:



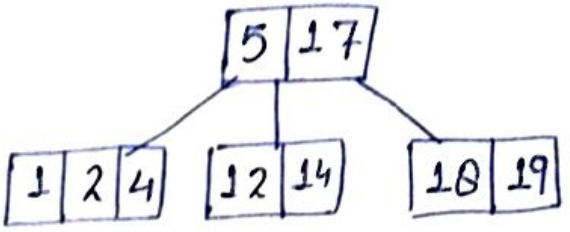
Insert 19:



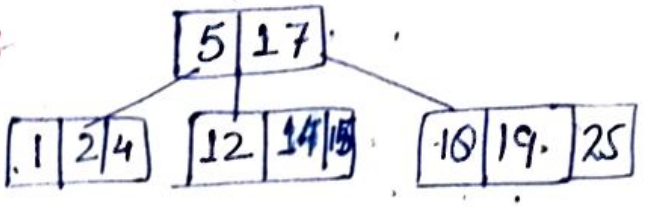
Insert 17:



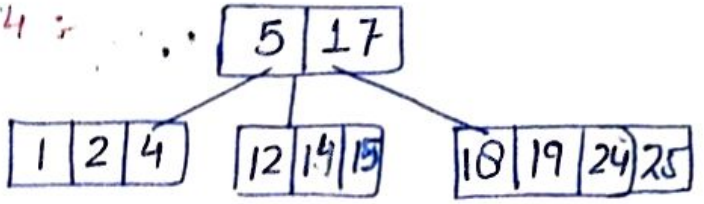
Split up



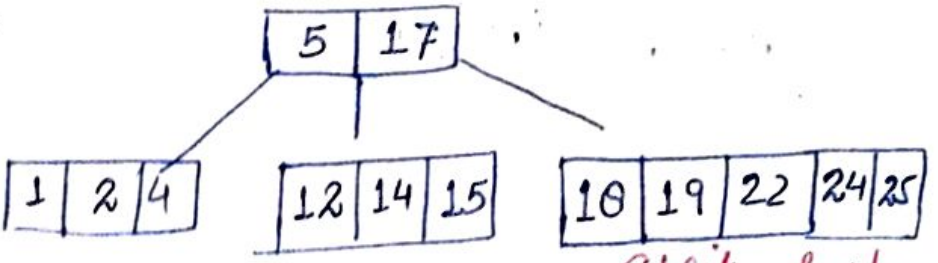
Insert 15 & 25:



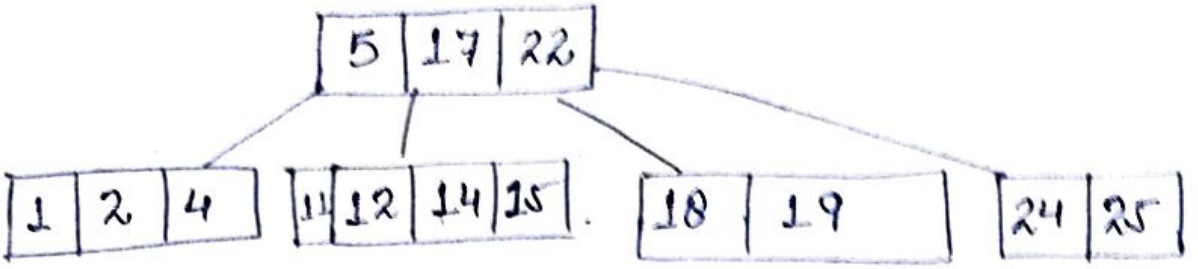
Insert 24:



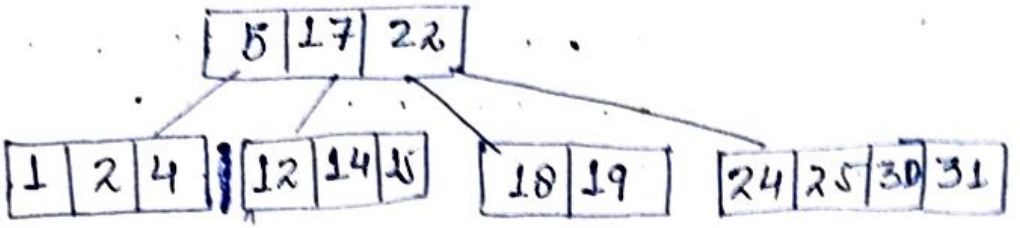
Insert 22:



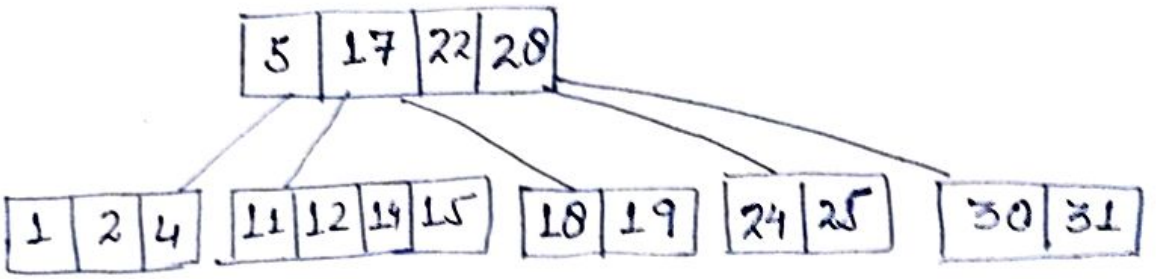
Split up



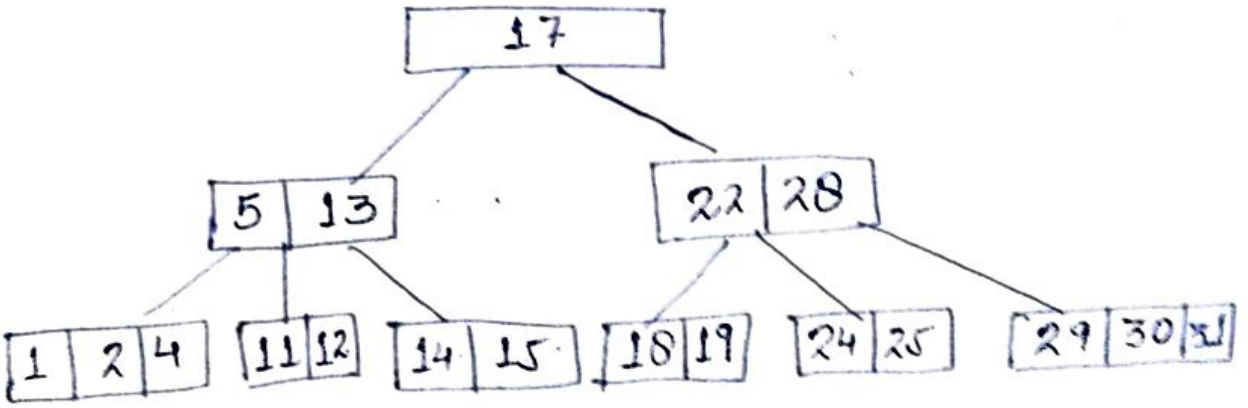
Insert 30, 31



Insert 28



Insert 29 & 13

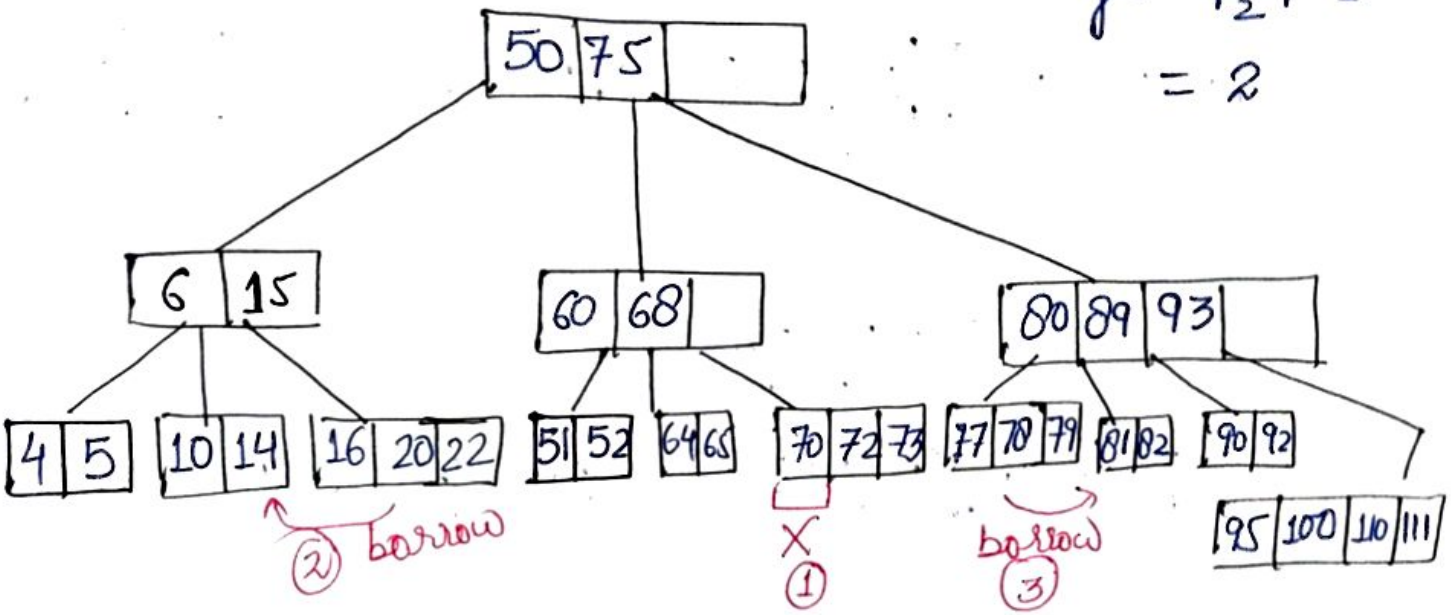


B-Tree Deletion :-

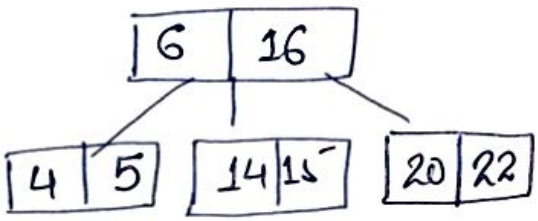
delete 70, 10, 81, 65, 75
① ② ③ ④ ⑤

$m = 5$

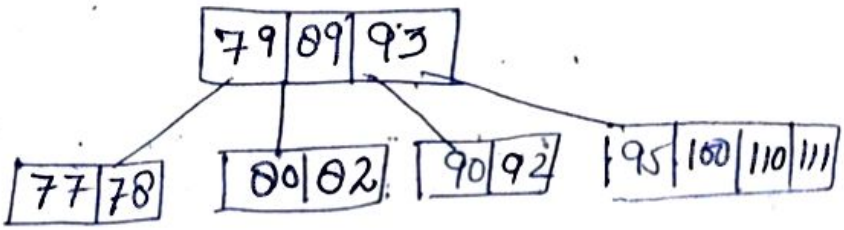
Min. Keys $= \lceil \frac{5}{2} \rceil - 1$
$= 2$



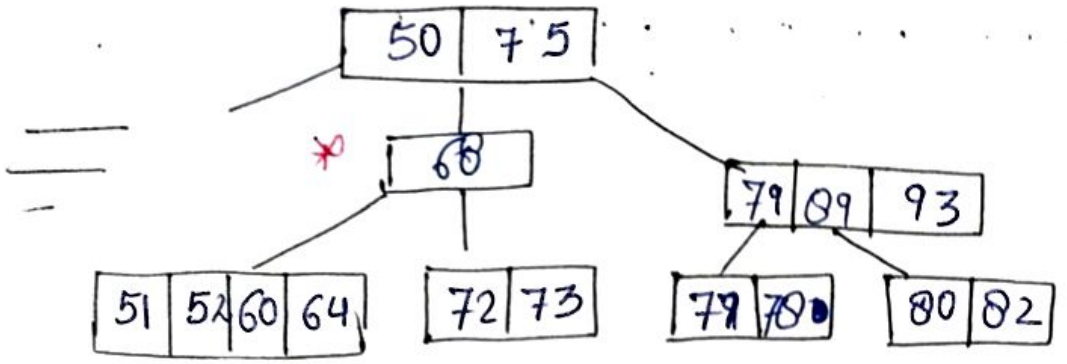
delete 10 :- Borrow from Sibling (if extra)
②

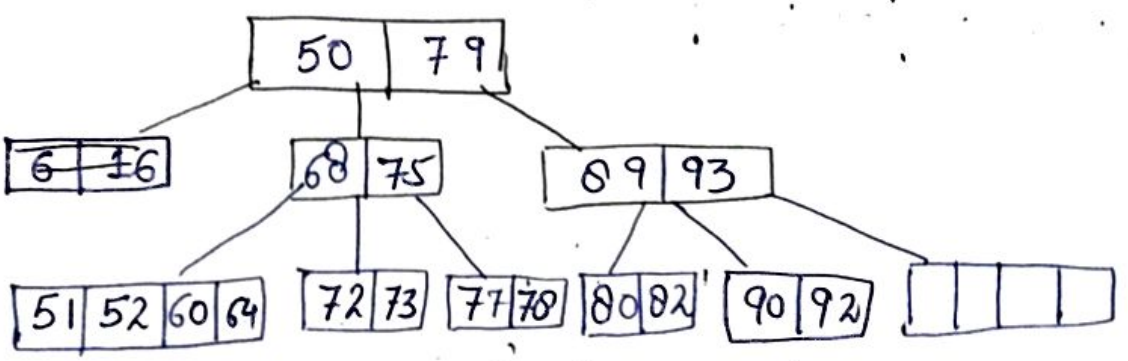


delete 81 :-
③

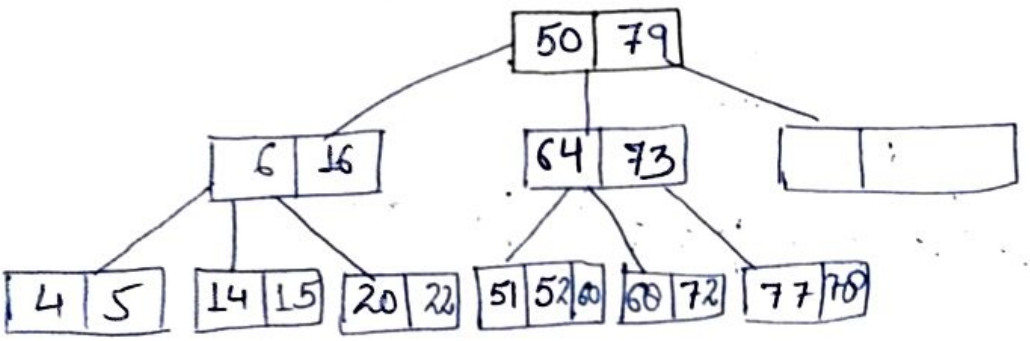


delete 65 :- (4) as can't borrow from L or R sibling, Thus we'll merge



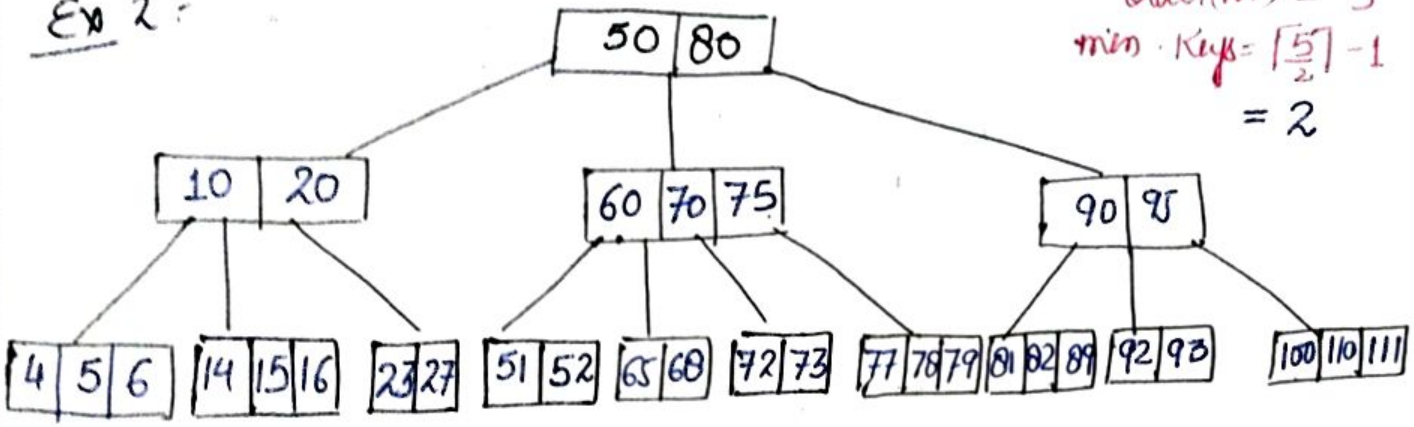


delete 75 :- (2) Replace 75 by its inorder Predecessor 73

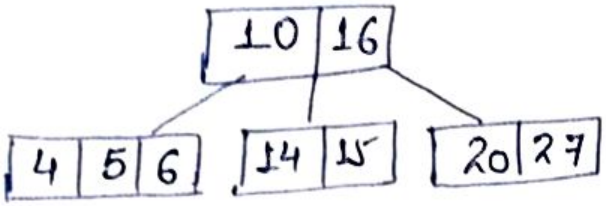


Ex 2:

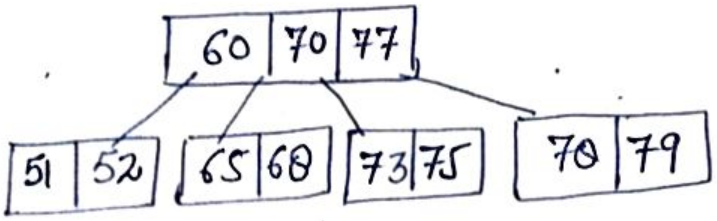
order(m) = 5
min. Keys = $\lceil \frac{5}{2} \rceil - 1$
= 2



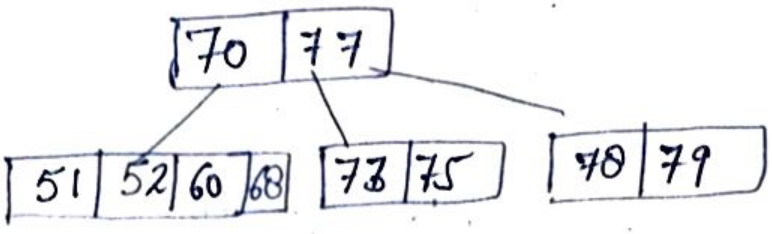
delete 23: Borrow from left sibling



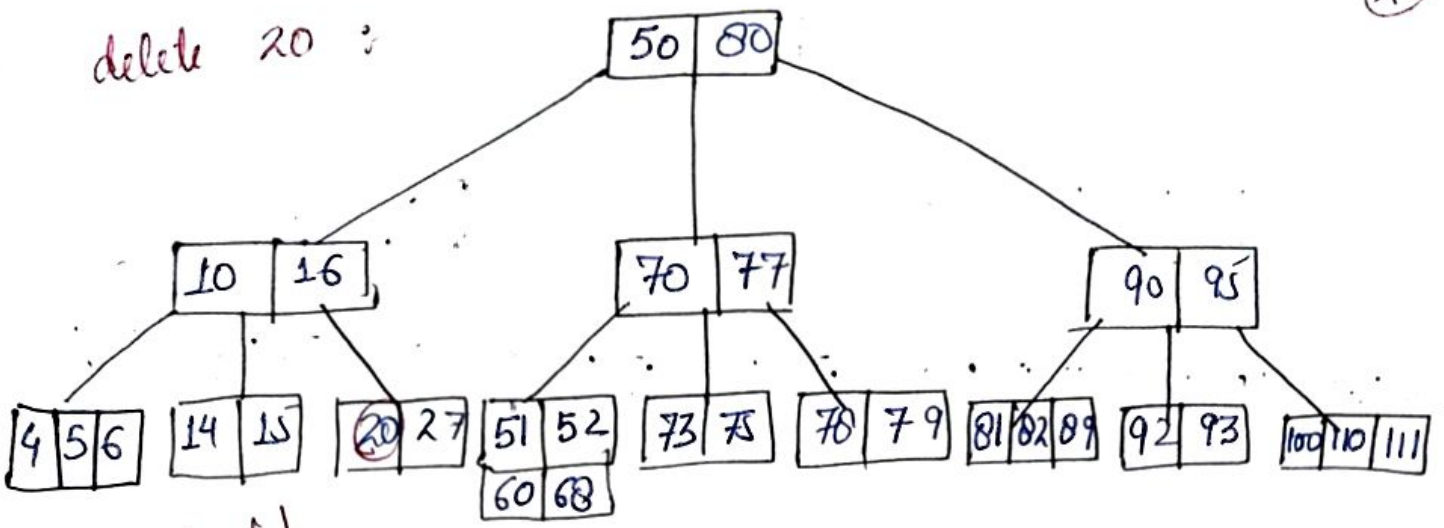
delete 72: Borrow from right sibling



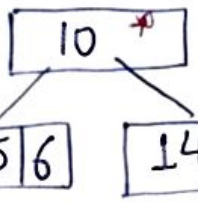
delete 65: can't borrow either from L nor R.
then merge:



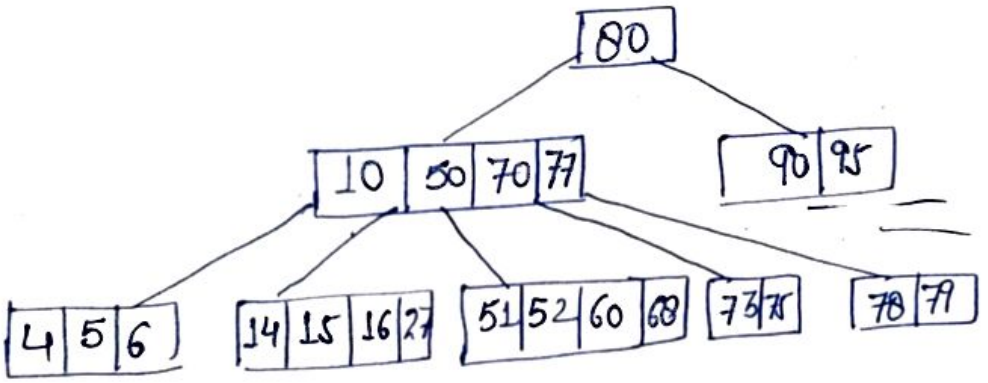
delete 20 :



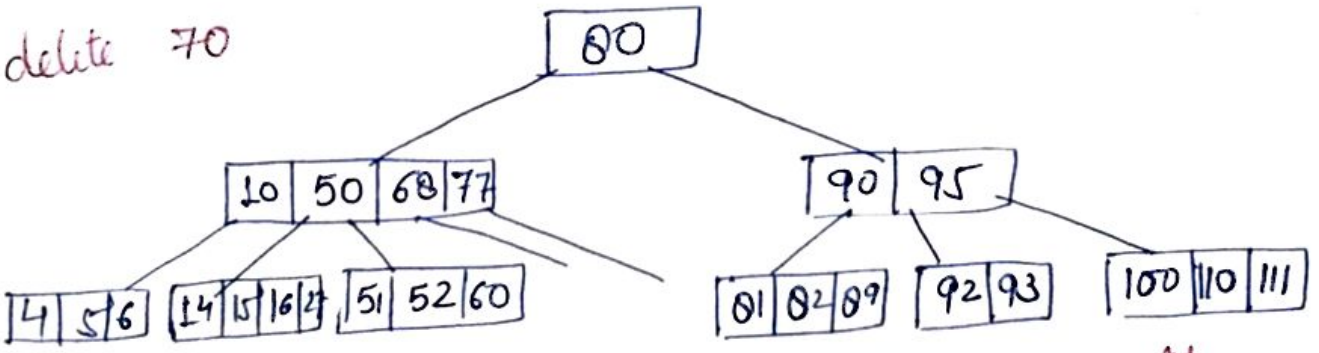
merge



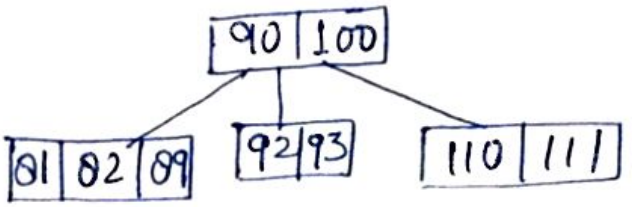
violate ⇒ can't borrow from Sibling than Merge



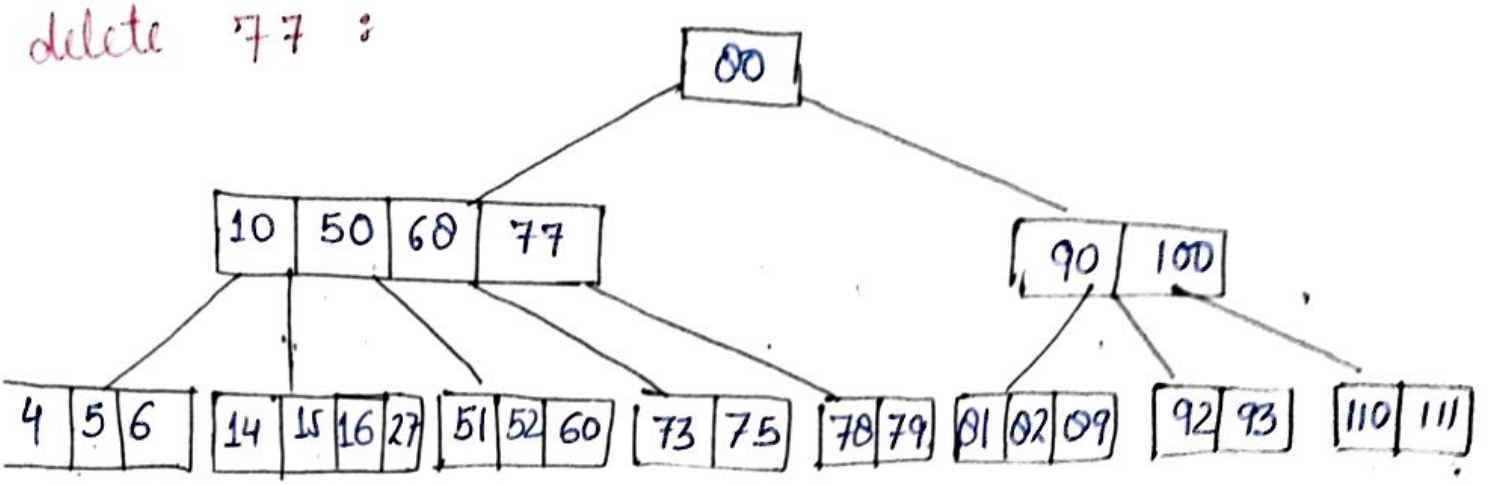
delete 70



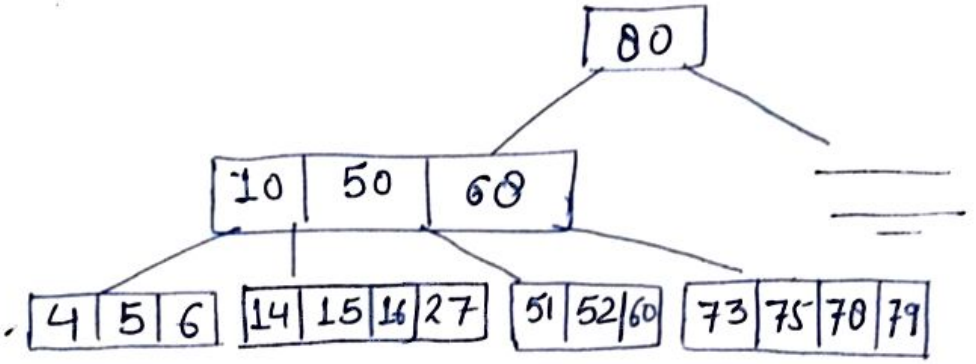
delete 95 ; can't replace Replace with Predecessor than with Successor :



delete 77 :



* can't Replace with inorder successor or predecessor, than Merge



* delete 80 : Replace with 79 (Inorder Pred.)

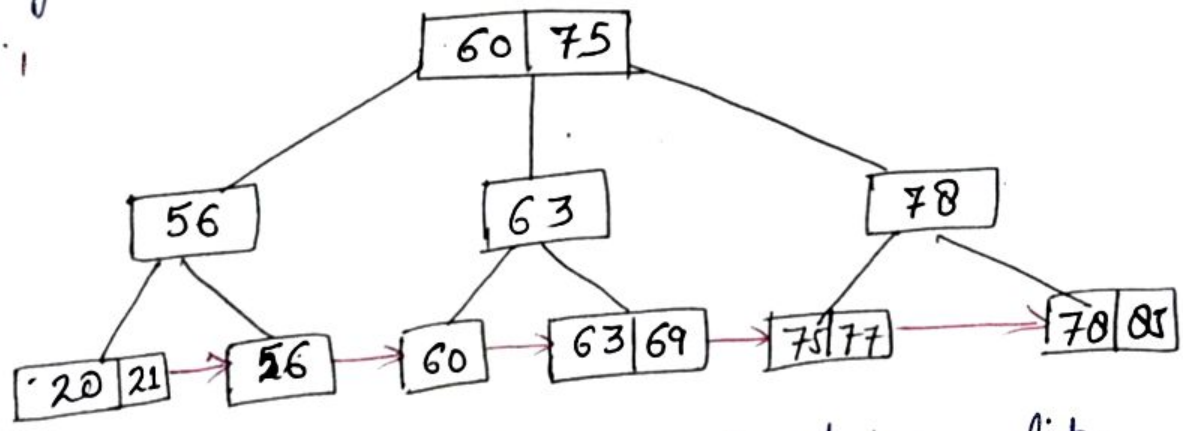
B^+ Trees :-

* B+ trees are extended version of B-Trees. This tree supports better insertion, deletion and searching over B-tree.

* In B^+ tree records can be stored at the leaf node; internal nodes will store key values only.

* The leaf nodes of B+ tree also linked together as linked list.

e.g.:



* This supports basic operations like searching, insertion & deletion. In each node, items will be sorted.

Advantages over B-Tree :-

* As leaf are connected like linked list, we can search elements in sequential manner also.

* Searching is faster.

* Records can be fetched in equal no. of disc accesses.

Insertion :-

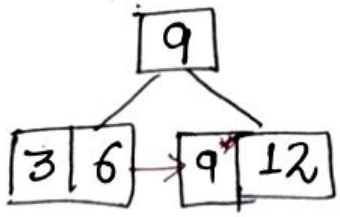
Q. Insert the following values into an empty B+ tree of order 4.

3, 6, 9, 12, 19, 23, 33, 27, 21, 22, 30, 44

order m = 4, max. child = 4
min " = $\lceil \frac{4}{2} \rceil = 2$
Max Keys = 3
Min Keys = $\lceil \frac{4}{2} \rceil - 1 = 1$

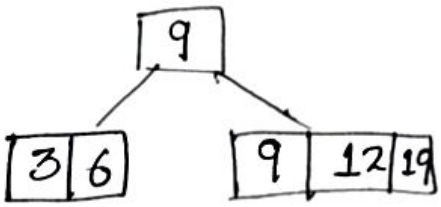
Insert 3, 6, 9 :-

Insert 12 Split at second middle

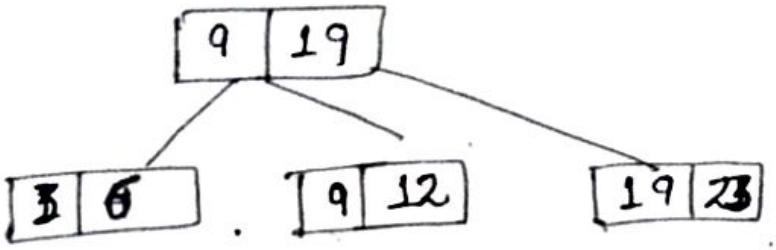


* all data must present in leaf node.

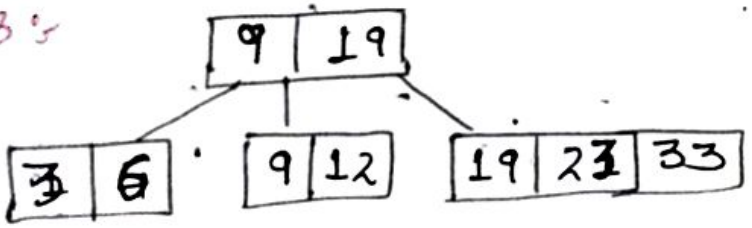
Insert 19



Insert 23:

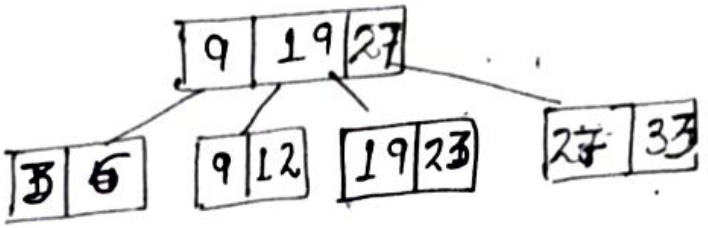


Insert 33:

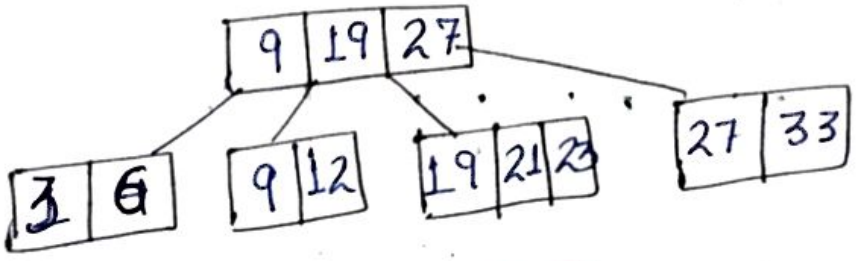


Insert 27:

Split at 27 & up

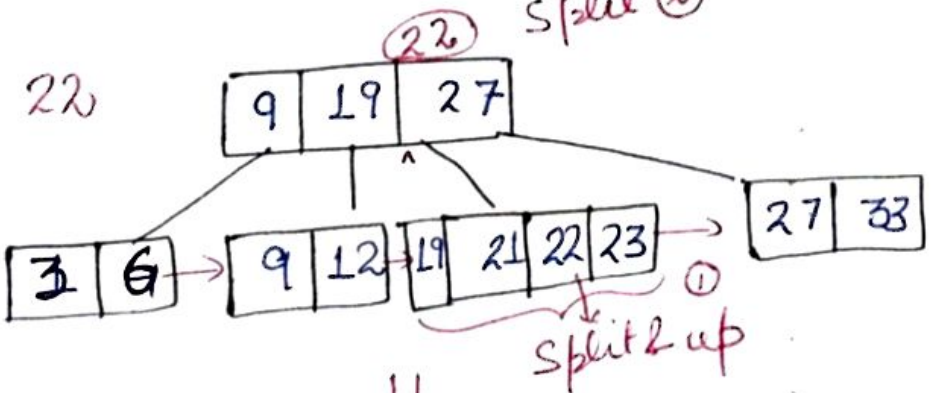


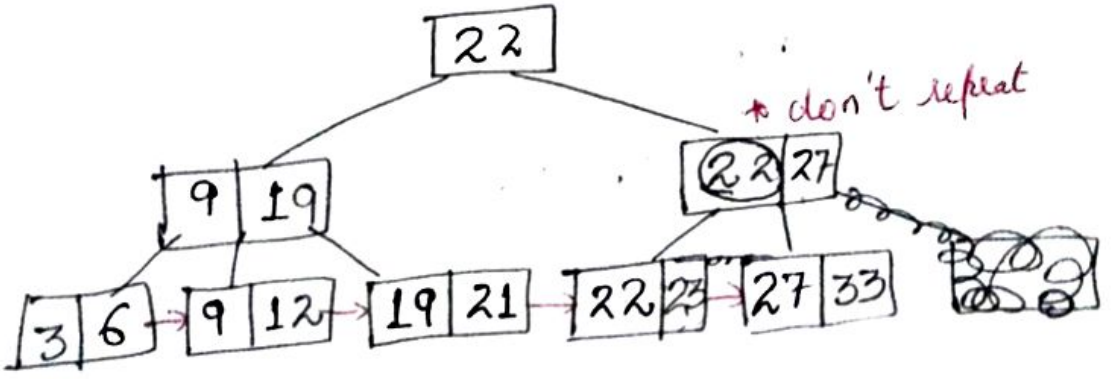
Insert 21



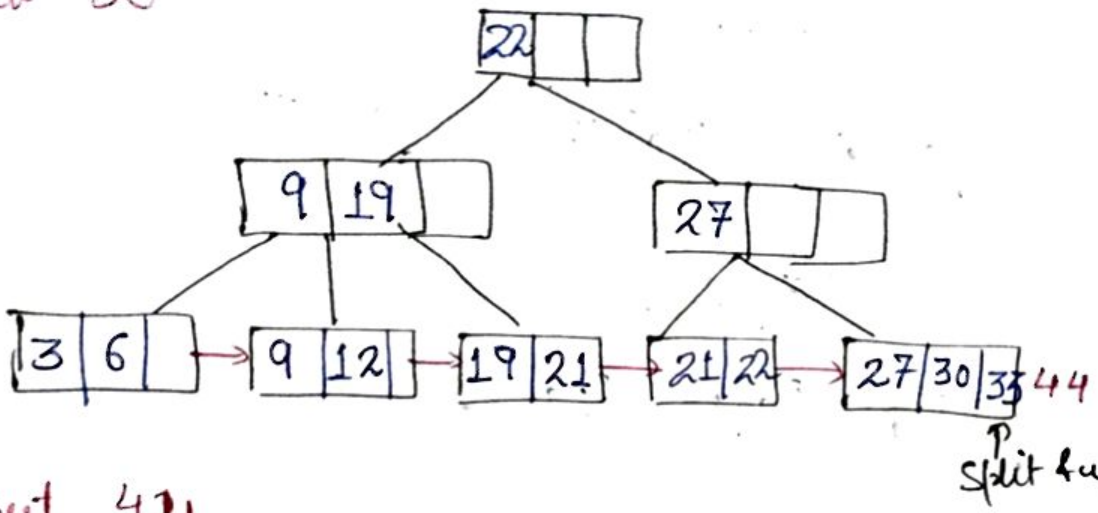
Insert 22

Split ②

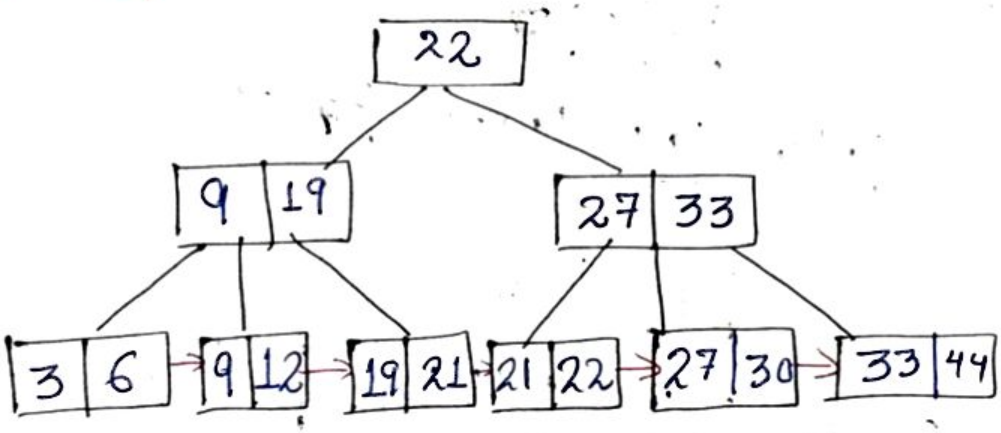




Insert 30



Insert 44

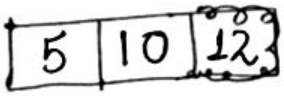


Q 5, 10, 12, 13, 14, 1, 2, 3, 4 · m = 3

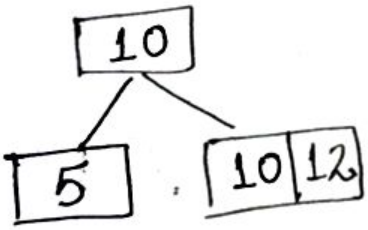
k = 2

Insert 5 & 10 :-

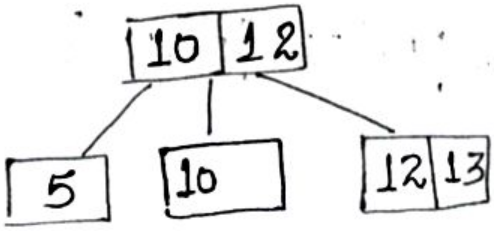
Insert 12 :-



split & up

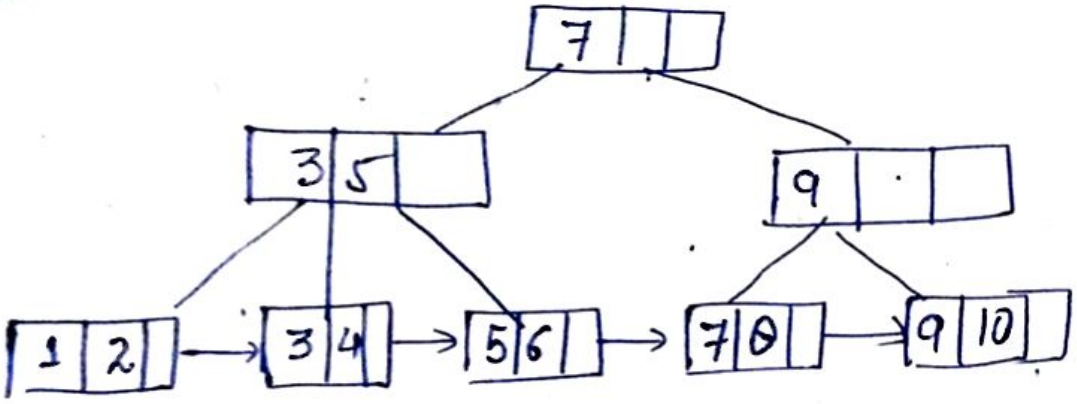


Insert 13 :-



Q :- Insert 1, 3, 5, 7, 9, 2, 4, 6, 0, 10 into an empty B+ tree of order 4.

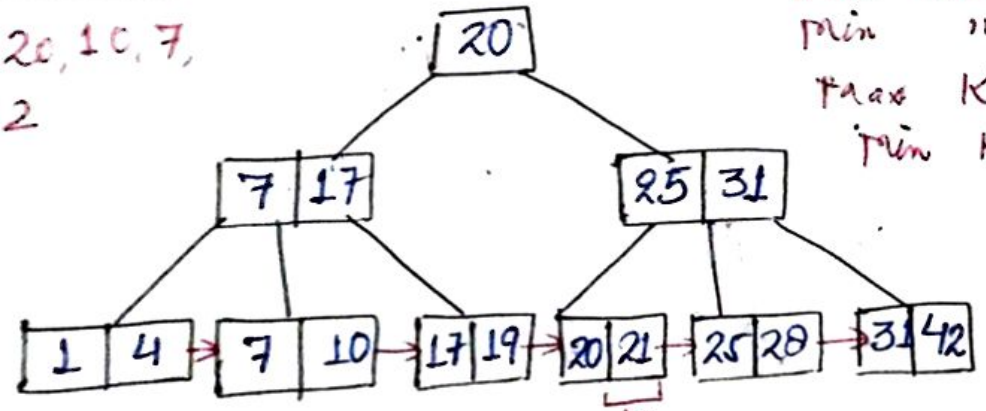
Ans



Deletion in B+ Tree :-

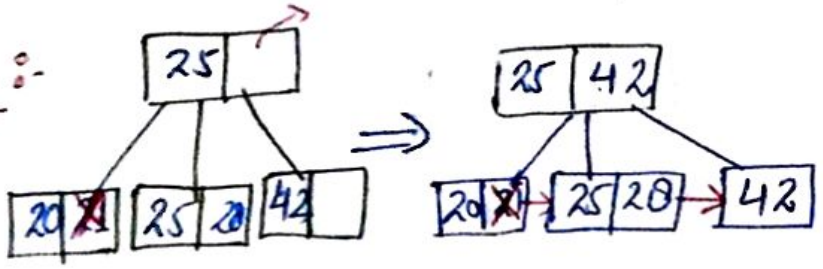
Order (m) = 4
max child = 4
min " = $\lceil \frac{4}{2} \rceil = 2$
max Key = 3
min Key = $\lceil \frac{4}{2} \rceil - 1 = 1$

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

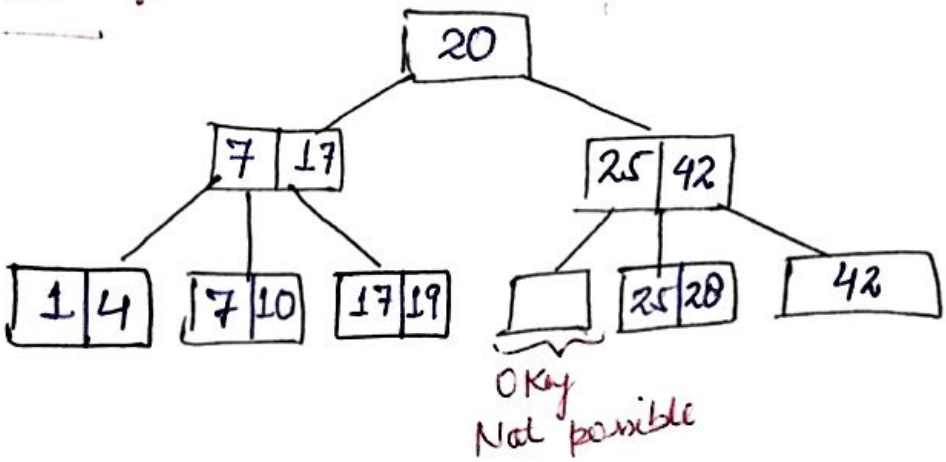


Delete 21 :- No Problem, as min. 1 Key will remain. copy min-value from right child ①

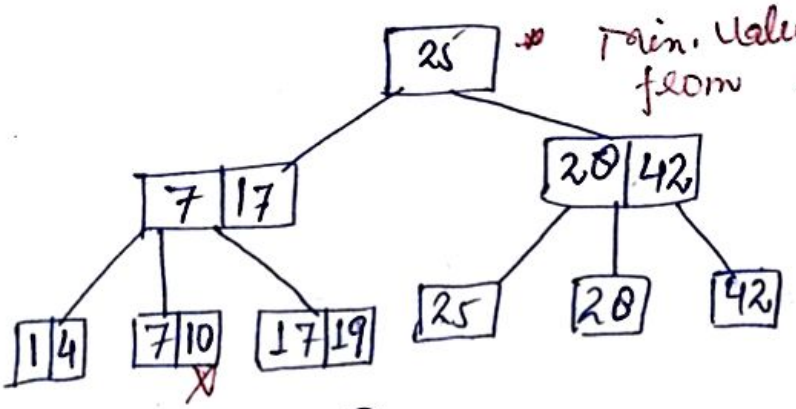
Delete 31 :-



Delete 20 :-



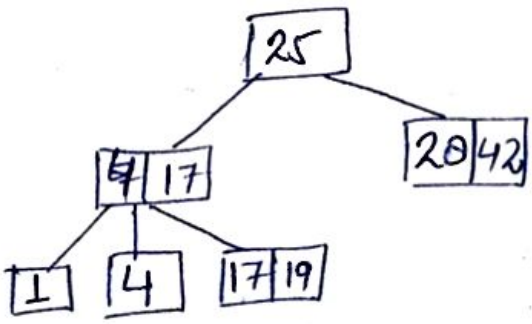
* Borrow from L or R Thus borrow smallest element from immediate right sibling, i.e, 25, & then shift new min. value from 20 . . . . . .



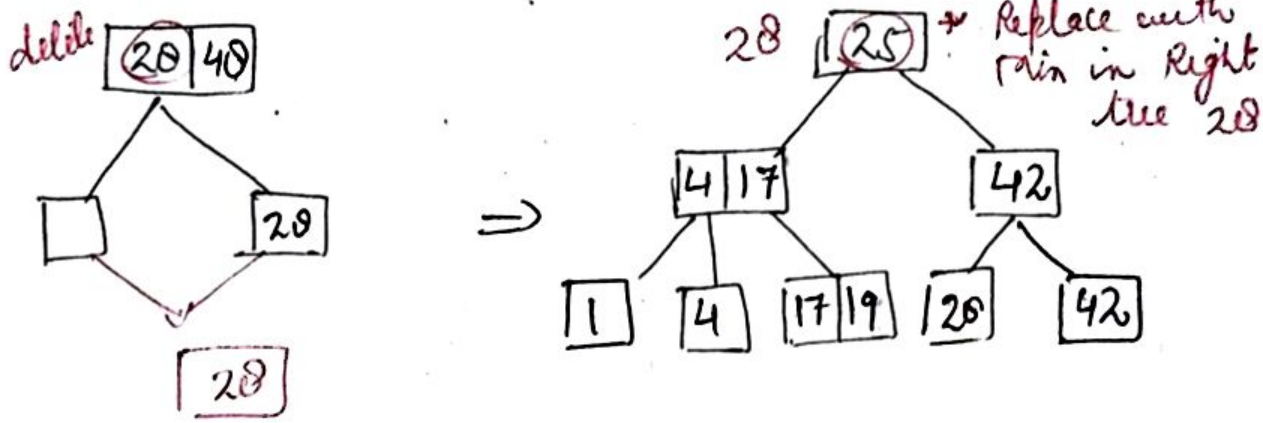
* Min. Value at leaf nodes from right subtree.

Delete 10 :- No Problem

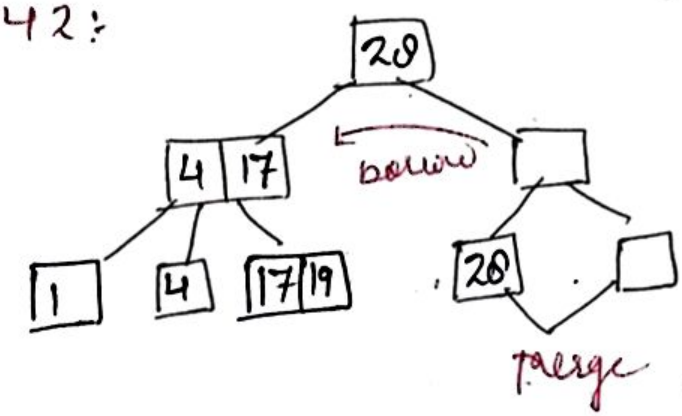
Delete 7 :-

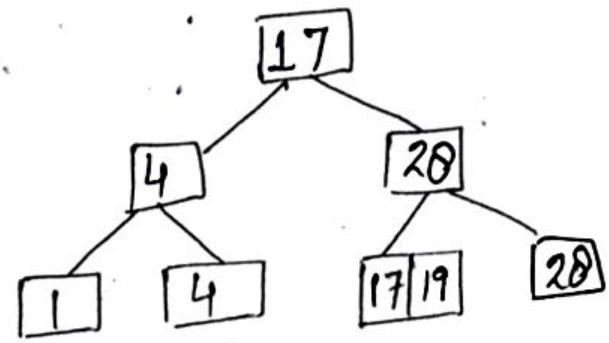


delete 25: Can't borrow from any of the sibling, then merge

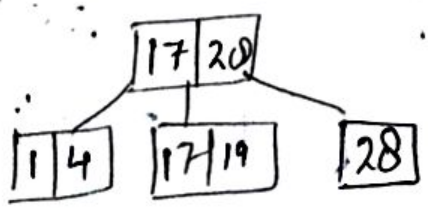


delete 42:





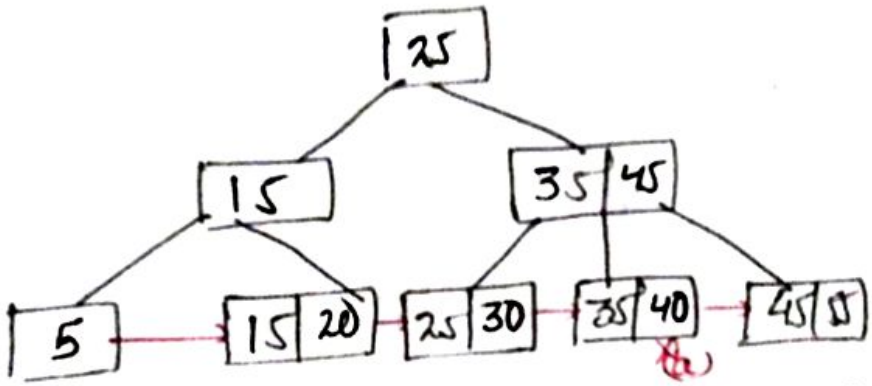
delete 4:-



delete 9 :

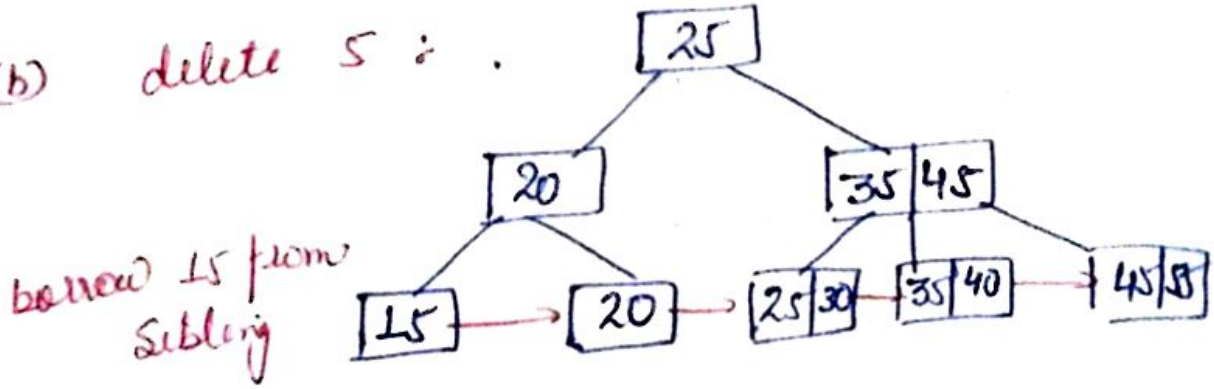
Ex 2: delete the following elements from the given B+ tree
40, 5, 45, 35, 25, 15

order = 3
min = $\lceil \frac{3}{2} \rceil - 1 = 1$

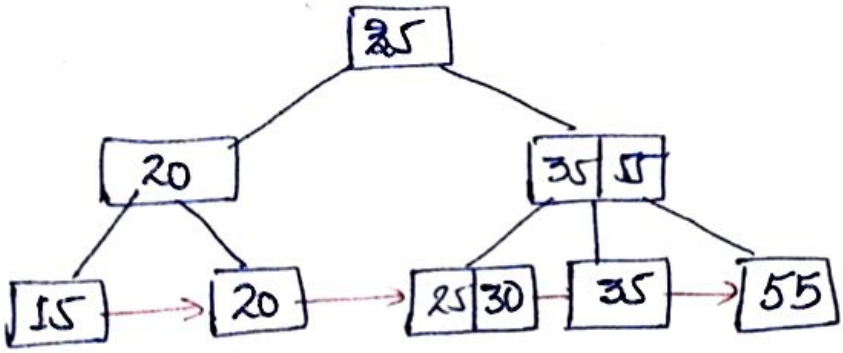


(a) delete 40 :- No changes required, as min. no. of keys are 1 & 35 remain there.

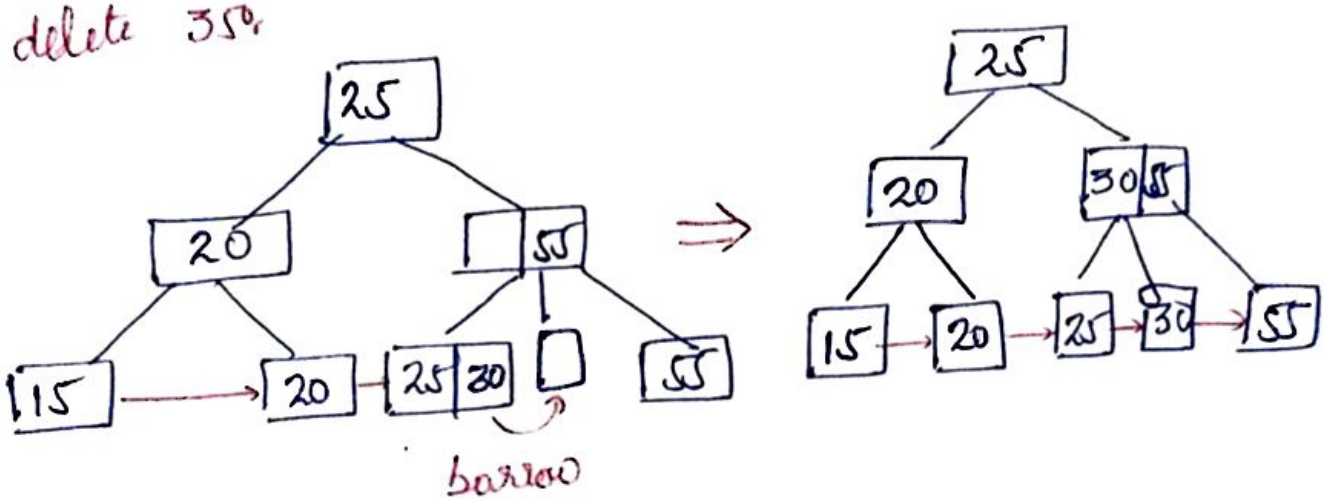
(b) delete 5 :



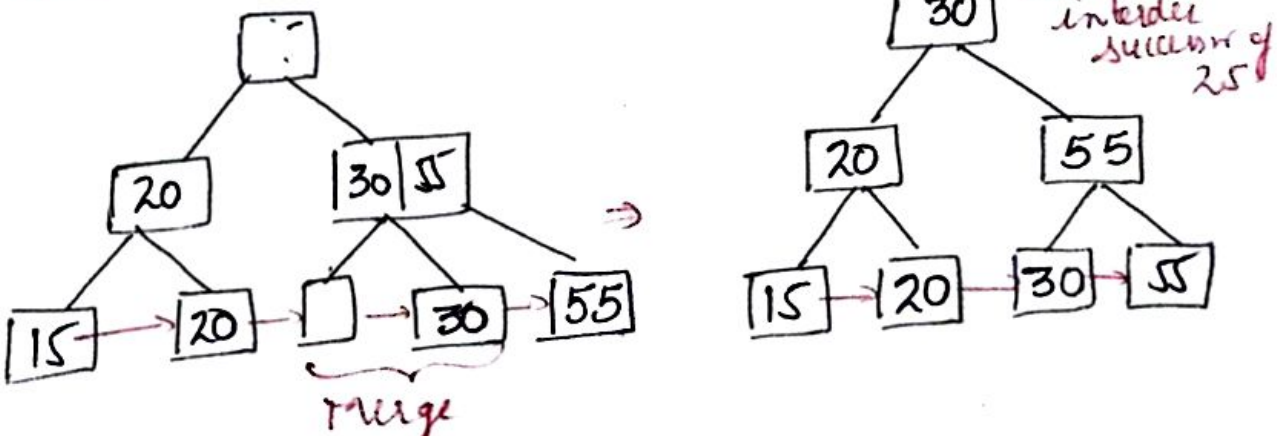
(c) delete 45 : 45 is in internal node.



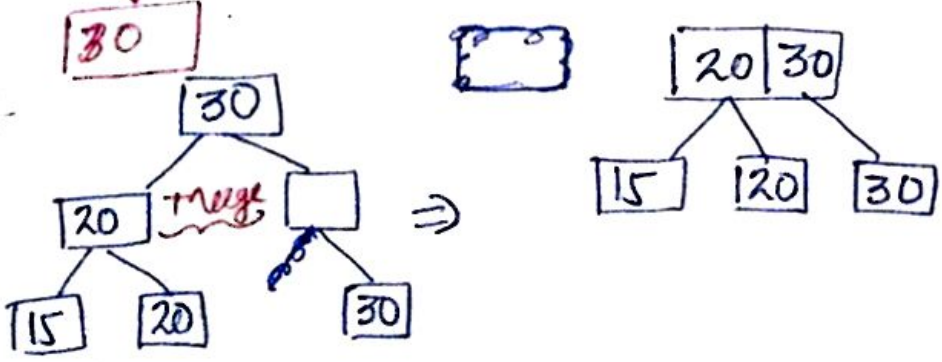
(d) delete 35:



(e) delete 25:



(f) delete 55 :-



Threaded Binary Tree

* As in case of ~~memory~~ binary tree, memory is not efficiently utilised. For storing 'n' values, '(n+1) values' are null values (leaf nodes).
* Hence, we have the concept of threaded binary tree.
* A binary tree with threaded pointers is called a threaded binary tree.
* In the linked representation of any binary tree if there are n no. of nodes, then there'll be (n+1) no. of null pointers. So in order to efficiently utilise the memory those null pointers can be replaced by some special pointers called as threads, which will point to the nodes higher in the tree.
* A binary tree with these thread pointers is called a threaded binary tree. Diagramatically, a thread can be represented by a dotted line

* There are different ways to thread a binary tree & each way corresponds to a particular type of traversal of the binary tree.

Types of Threads :-

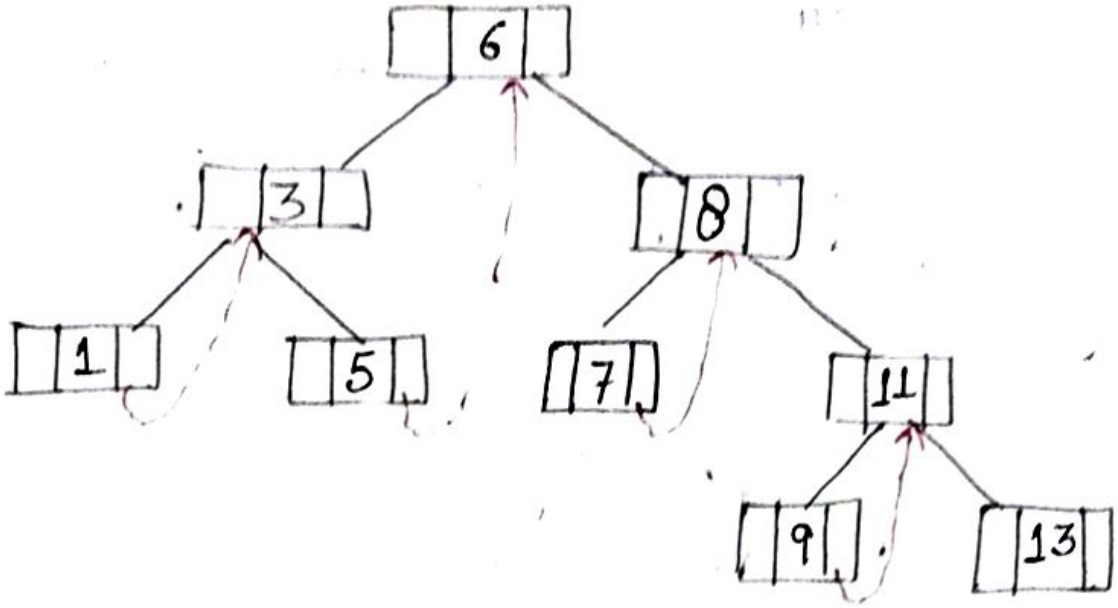
1. One-way threading
2. two-way threading

① One-way Threading :-

* ~~In one way threading a thread will appear in the right / null pointer field~~

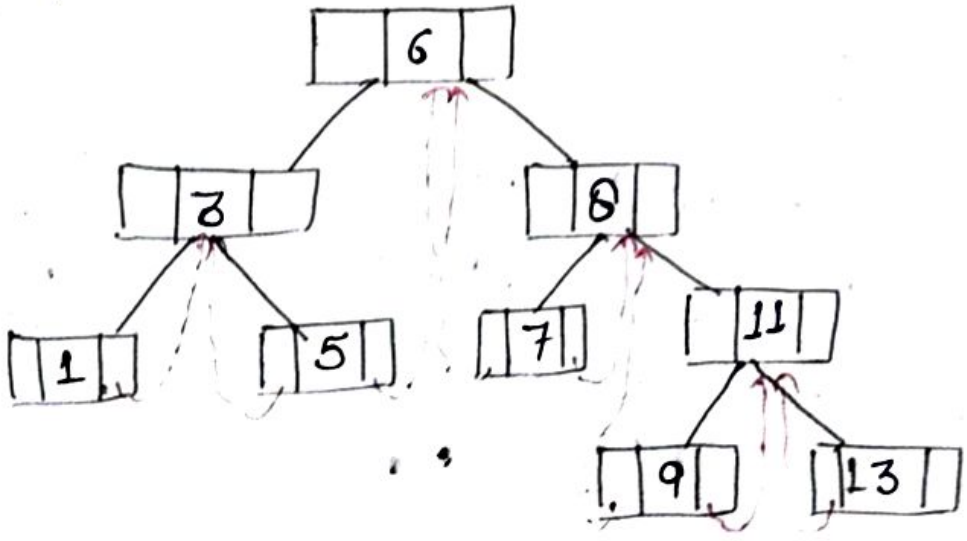
* Each node is threaded towards either the in-order predecessor or successor (left OR right).

* All right NULL pointers will point to inorder successor OR all the left NULL pointers will point to inorder predecessor.



② Two-way Threading :-

Each node is threaded towards both in-order predecessor & successor (left & right). Thus all right NULL pointers will point to inorder successor AND all left pointers will point to inorder predecessor.



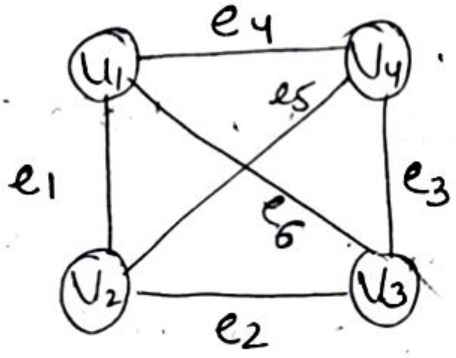
Graph:

* A Graph G=(V,E). consists of two sets.

1) A set V of elements called as nodes or vertices.

2) A set E of edges such that each edge $e \in E$ is identified by a unique pair (u, v) of nodes in V and is denoted as $e = [u, v]$.

* Graph is a non-linear data structure



$V(G) = \{V_1, V_2, V_3, V_4\}$

$E(G) = \{e_1, e_2, e_3, e_4, e_5, e_6\}$

$= \{(V_1, V_2), (V_2, V_3), (V_3, V_4), (V_4, V_1) - - \}$

* Edge e_1 connects the vertex V_1 & V_2. Hence. V_1 and V_2 are the end points of edge e_1 and V_1 & V_2 are called as the adjacent nodes or neighbour.

Degree of a node (u) :

degree (u) is the no. of edges connected to the node u.

$deg(v_4) = 3$, $deg(v_3) = 3$

* if any node's degree is zero, then that node is called as isolated node.
* A path P of length n from a node to another node is the sequence of (n+1) nodes

e.g., $P = \{(v_1, v_2), (v_2, v_3)\}$

length = 2.

Note :- The path from one node to another is not unique.

Simple Path :- A path P is said to be simple if all the nodes in that path are distinct. (No Repetation)

closed Path :- A path P is said to be closed if starting node & destination nodes are same.

$v_4 \rightarrow v_2 \rightarrow v_3 \rightarrow v_4 \rightarrow v_1$

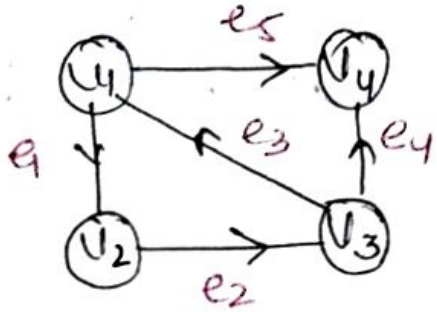
Cycle :- cycle is a closed simple path of length 3 or more.

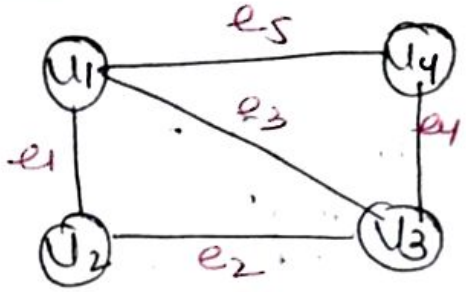
$v_1 \rightarrow v_2 \rightarrow v_3 \rightarrow v_4 \rightarrow v_1$

$v_1 \rightarrow v_2 \rightarrow v_3 \rightarrow v_1$

Types of Graphs :-

① Directed Vs Undirected Graph :-





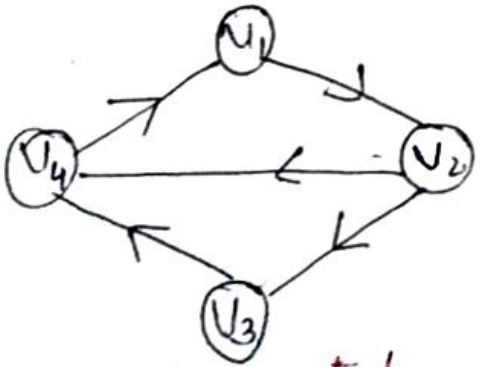
* e_5 is an edge from v_1 to v_4 but there is no edge from v_4 to v_1

* e_5 is an edge from v_1 to v_4 as well as from v_4 to v_1.

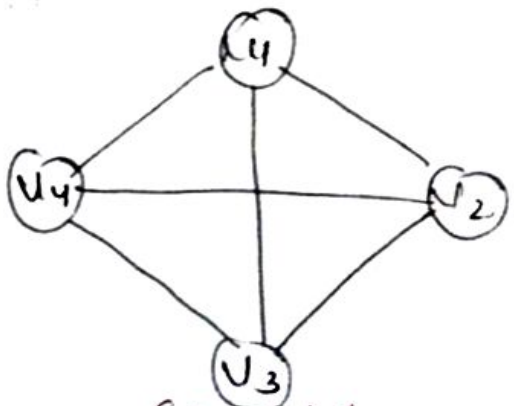
② Connected Vs Complete Graphs :-

⇒ A graph G is said to be connected if and only if there exists a ~~single~~ simple path between any two nodes of the graph.

⇒ A graph G is said to be a complete graph if each node u is adjacent to all other nodes v of G.



Connected

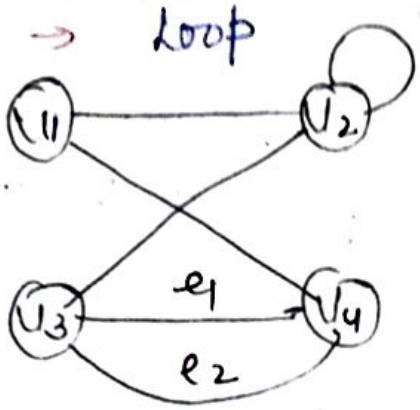


Complete

* If a complete graph contains n nodes, then there will be $\frac{n(n-1)}{2}$ no. of edges

Multigraph :-

* A multigraph consists of the following two things :
  - → Parallel edges
  - → Loop



$e_1 = (V_3, V_4)$, $e_2 = (V_3, V_4)$

Parallel edges :- Two different edges e_1 & e_2 are said to be parallel edges if they have same end points. e.g., e_1 & e_2

Loop :- An edge e is called a loop if it has identical end points e.g., e_3

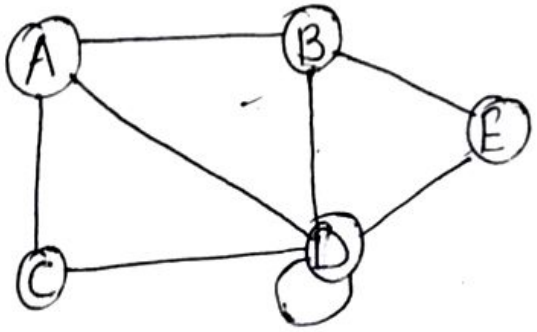
Pendant Vertex :- A vertex with degree one is known as pendant vertex.

- Representation of Graph :-

1. Sequential representation using adjacency matrix.
2. Linked representation using linked list and adjacency list

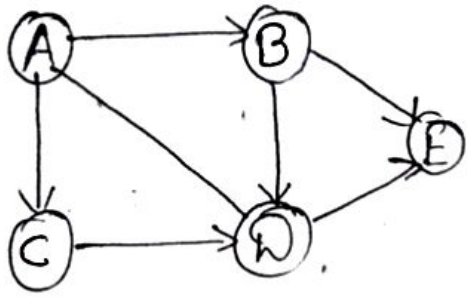
Adjacency Matrix :-

* In this representation the graph is represented using a matrix of size total no. of vertices by total no of vertices.
* Thus a graph with 4 vertices is represented using a matrix of size 4×4.
* The matrix is filled with either 1 or 0.
  - → 1 represents that there is an edge from row vertex to column vertex.
  - → 0 represents that there is no edge from row vertex to column vertex.



	A	B	C	D	E
A	0	1	1	1	0
B	1	0	0	1	1
C	1	0	0	1	0
D	1	1	1	1	1
E	0	1	0	1	0

Directed graph representation :-



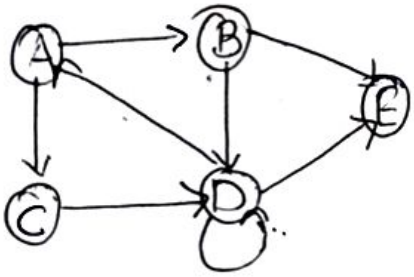
$\Rightarrow$

	A	B	C	D	E
A	0	1	1	0	0
B	0	0	0	1	1
C	0	0	0	1	0
D	1	0	0	1	1
E	0	0	0	0	0

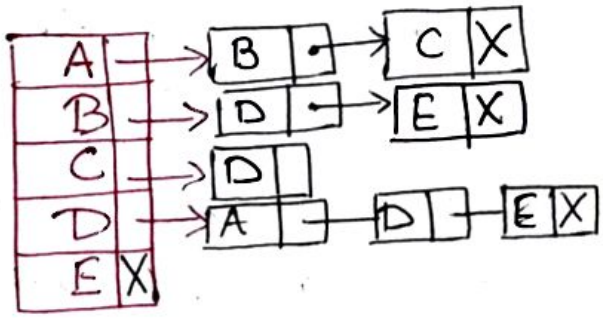
Adjacency List :-

* In this representation, every vertex of a graph contains list of its adjacent vertices.

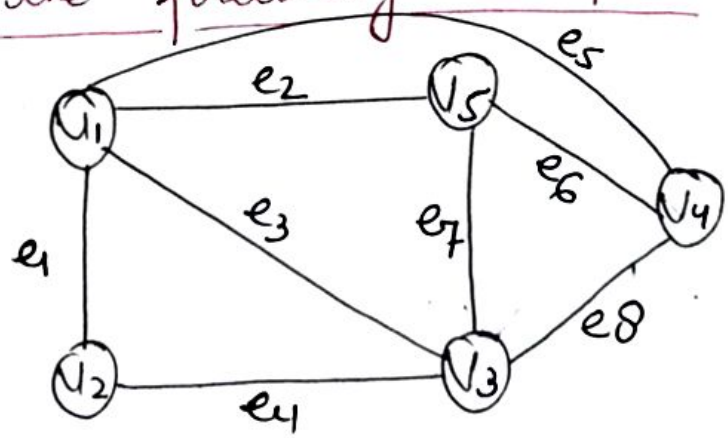
e.g.,



$\Rightarrow$



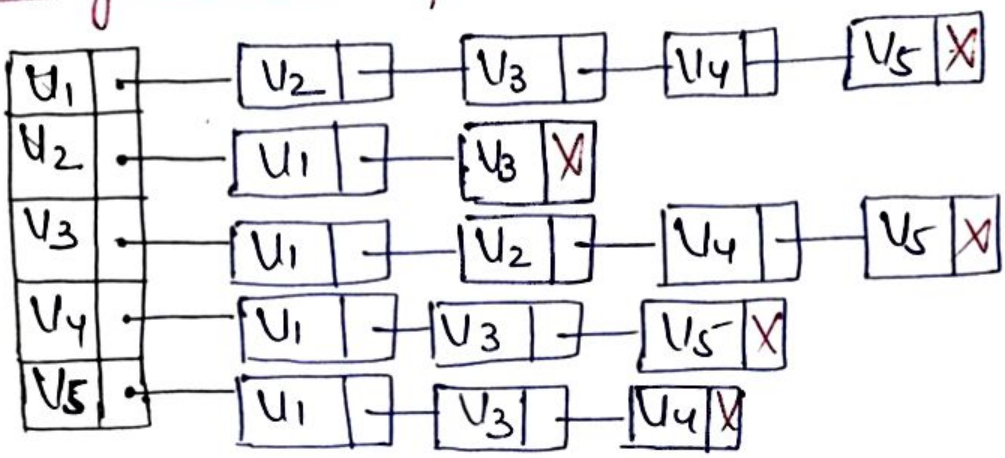
Q. Consider the following examples :-



Adjacency Matrix Representation :-

	v_1	v_2	v_3	v_4	v_5
v_1	0	1	1	1	1
v_2	1	0	1	0	0
v_3	1	1	0	1	1
v_4	1	0	1	0	1
v_5	1	0	1	1	0

Adjacency List Representation :-



Incidence Matrix Representation :-

	e_1	e_2	e_3	e_4	e_5	e_6	e_7	e_8
v_1	1	1	1	0	1	0	0	0
v_2	1	0	0	1	0	0	0	0
v_3	0	0	1	1	0	0	1	1
v_4	0	0	0	0	1	1	0	1
v_5	0	1	0	0	0	1	1	0

Traversing of a Graph :-

* There are two standard ways to traverse any graph :

(1) Breadth First Search (BFS)

(2) Depth First Search (DFS)

* Many graph algo require one to systematically examine the nodes and edges of a graph G.

* The Breadth first search (BFS) will use a queue as an auxiliary structure to hold nodes for future processing. and depth first search will use a stack.

* During the execution of our algo, each node N of G will be in one of three states, called the status of N, as follows :

Status = 1 : (Ready state) The initial state of N

Status = 2 : (Waiting State) The node N is on the queue or stack, waiting to be processed.

Status = 3 : (Processed State) The node N has been processed.

Breadth First Search

* The general idea behind a BFS beginning at a starting node A is as follows
  - → First examine the starting node A
  - → Then examine all the neighbours of A
  - → Then examine all the neighbours of neighbours of A and so on
  - → We need to guarantee that no node is processed more than once.
  - → This is accomplished by using a queue to hold nodes that are waiting to be processed & by using a field STATUS which tells us the current status of any node.

Algorithm (BFS) :-

1. Initialize all nodes to the ready state (STATUS = 1)
2. Put the starting node A in the QUEUE & change its status to waiting (STATUS = 2).
3. Repeat step 4 & 5 until QUEUE is empty.
4. Remove the ~~first~~ front node N of QUEUE Process N & change the status of N to processed STATUS = 3.

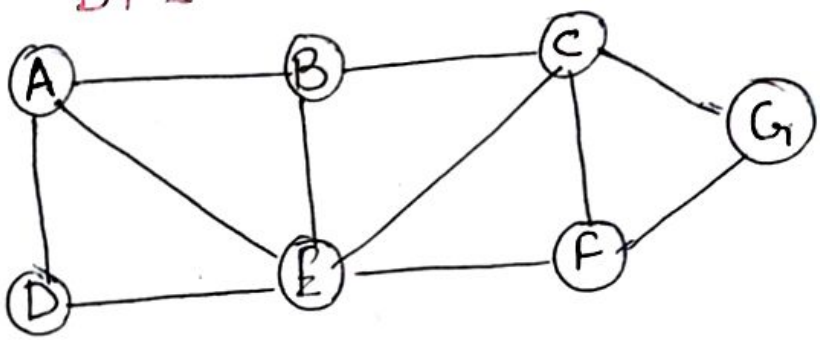
5. Add to the rear of QUEUE all the neighbours of N that are in steady state (STATUS = 1) & then change their status to the waiting state (STATUS = 2)
[End of step 3 loop]

6. Exit

Q Consider the following example to perform BFS traversal

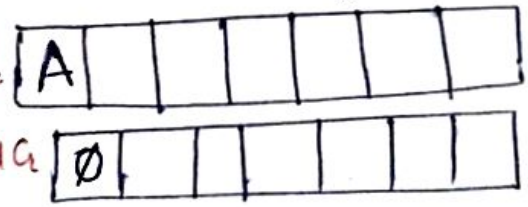
List

A:- B, D, E
B: A, C, E
C: B, E, F, G
D: A, E
E: A, B, C, D, F
F: C, E, G
G: C, F



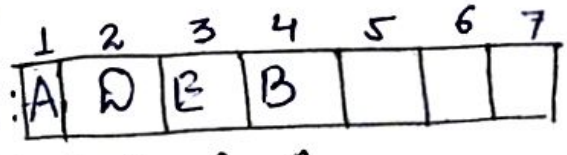
(i) Initially add A to QUEUE and add NULL to ORIG as follows:-

FRONT = 1
REAR = 1



(ii) Remove front element A from QUEUE
Set front = front + 1 & add to QUEUE the neighbours of A

Front = 2
Rear = 4



* Origin A of all three edges is added to ORIG!

(iii) F = 3
R = 4

Q | A | D | E | B | | | |

Orig ∅ A A A

(iv) F = 4
R = 6

Q | A | D | E | B | C | F | |
1 2 3 4 5 6 7

Orig ∅ A A A E E

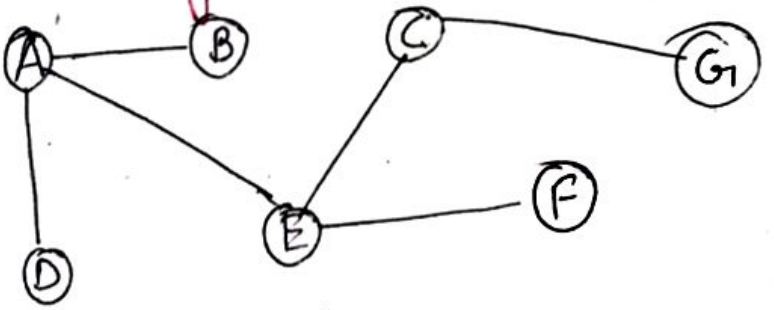
(v) F = 5
R = 7

Q | A | D | E | B | C | F | G |

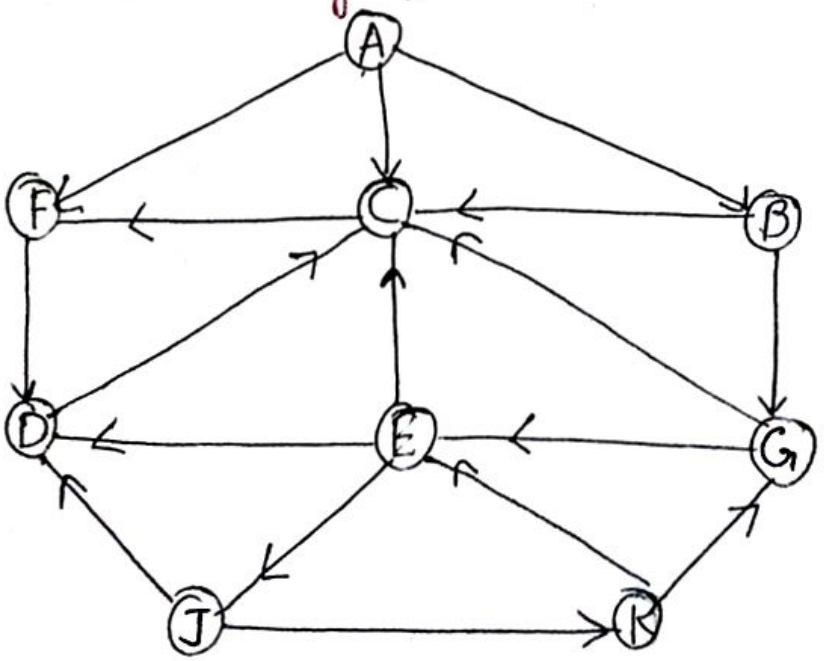
Orig ∅ A A A E E C

(vi) F = 6
R = 7

Final Result of BFS is :-



Q :- Consider the graph below



Adjacency list
A : F, C, B
B : G, C
C : F
D : C
E : D, C, J
F : D
G : C, E
J : D, K
K : E, G

(a) Initially, add A to QUEUE & add NULL to ORIG as follows:-

FRONT = 1	QUEUE = A
REAR = 1	ORIG = ϕ

(b) Remove the front element A from QUEUE by setting FRONT = FRON + 1 & add to QUEUE the neighbours of A as follows:-

FRONT = 2	QUEUE = A, F, C, B
REAR = 4	ORIG = ϕ, A, A, A

(c) Remove the front element F from the Q by setting F = F + 1 & add neighbours of F.

F = 3 QUEUE A, F, C, B, D

R = 5 ORIG: ϕ A, A, A, F

(d) F = 4, R = 5 — Q : A, F, C, B, D; O : ϕ A, A, A, F

neighbour F of C can't be added as F is not in the ready state

(e) F = 5, R = 6 — Q : A, F, C, B, D, G; O : ϕ, A, A, A, F, B

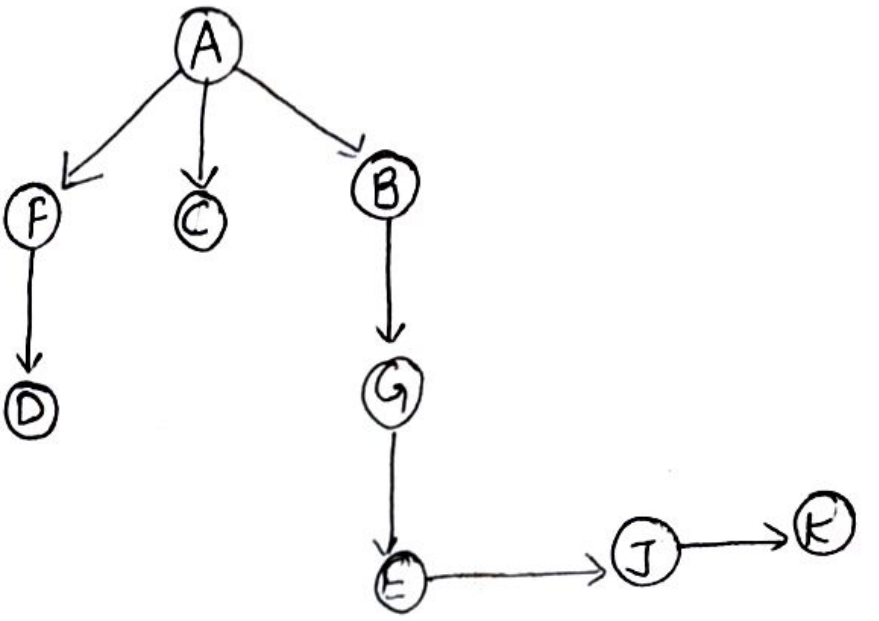
(f) F = 6, R = 6 — Q : A, F, C, B, D, G; O : ϕ, A, A, A, F, B

(g) F = 7, R = 7 — Q : A, F, C, B, D, G, E; O : ϕ, A, A, A, F, B, G

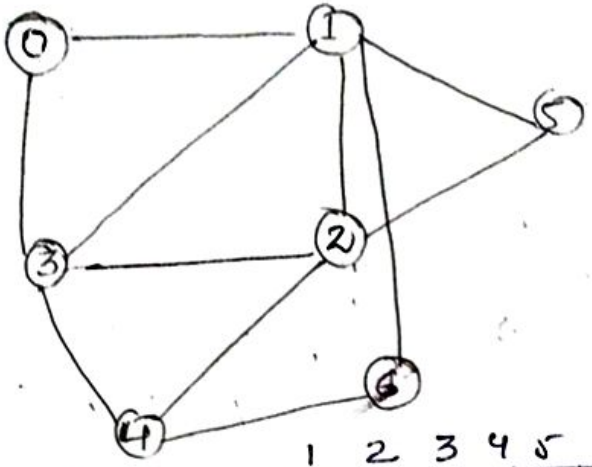
(h) F = 8, R = 8 — Q : A, F, C, B, D, G, E, J; O : ϕ A, A, A, F, B, G, E

(i) F = 9, R = 9 — Q : A, F, C, B, D, G, E, J, K; O : ϕ, A, A, A, F, B, G, E, J

(j)

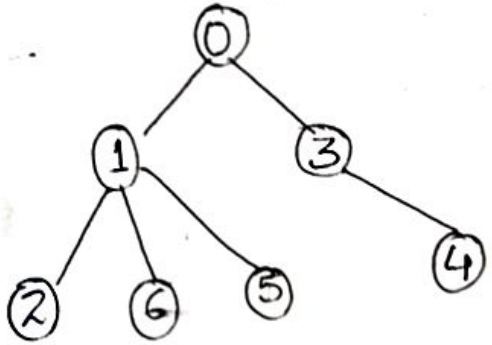


Q.



0: 1, 3
1: 0, 3, 2, 6, 5
2: 3, 1, 4, 5
3: 0, 1, 2, 4
4: 2, 3, 6
5: 1, 2
6: 1, 4

Start with '0'



1	2	3	4	5	6	7
0						
ϕ						

0	1	3				
ϕ	0	0				

0	1	3	2	6	5	4
ϕ	0	0	1	1	1	3

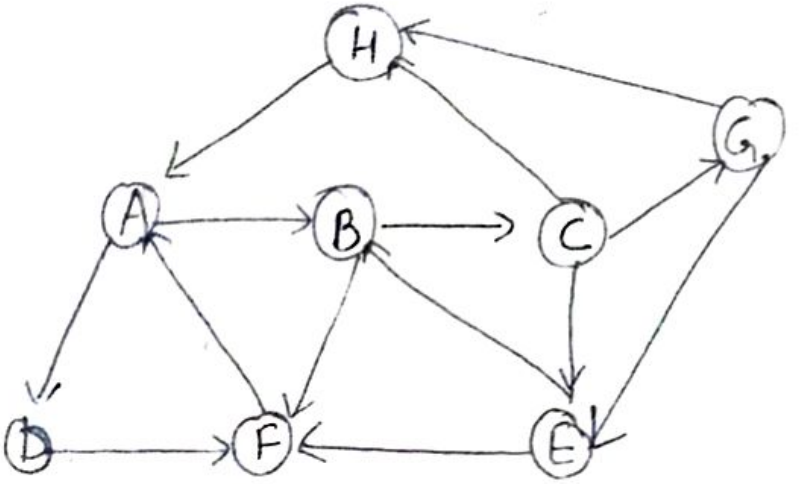
0	1	3	2	6	5	4

After coming to a dead end, that is, to the end of path P, we backtrack on P until we can continue along another path

Algorithm (DFS) :-

1. Initialize all nodes to the ready state (STATUS = 1).
2. Push starting node A onto stack.
3. Repeat step 4 & 5 until STACK is empty.
4. Pop the top node N of STACK. Process N & change its status to the processed (STATUS = 3)
5. Push onto the STACK all the neighbours of N that are in ready state & Change their status to waiting (STATUS=2) [End of step 3 loop]
6. Exit

Example : consider the following graph :-

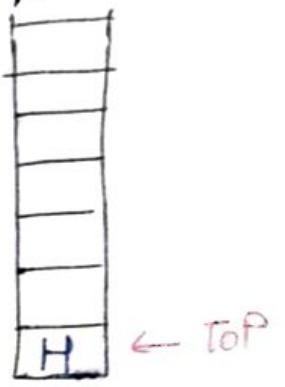


Adjacency list

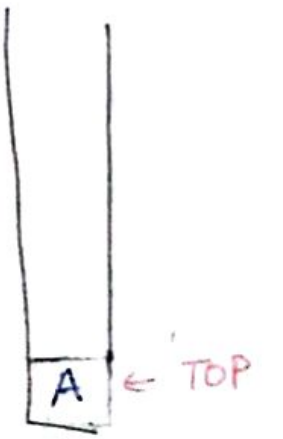
A: B, D
B: C, F
C: E, G, H
D: F
E: B, F
F: A
G: E, H
H: A

Step 1 :- Lets Take initial vertex as H

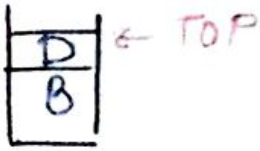
Now pop the top element of the stack i.e; H, print it & push all the neighbours of H onto the stack that are in ready state.



Step 2 :- Pop the top element i.e., A, print it & push all the neighbours of A that are in ready state



Step 3 :- Pop the top element i.e, D, print it & push all the neighbours of D i.e, F.

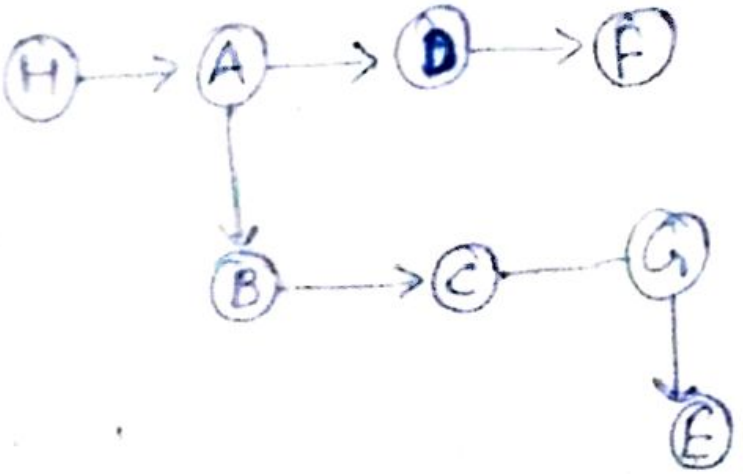


Pop D & push its neighbours but there is no neighbour in ready state now pop B & push all its neighbours in ready state.

Pop C & push E, G

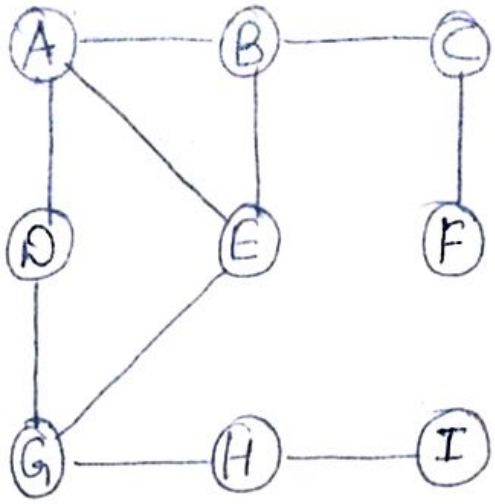
Pop G & push all its neighbours but neither E nor H is in ready state

Thus the final spanning tree will be :-



DFS

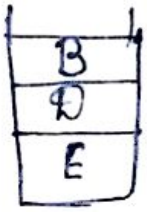
Example :-



A: B, E, D
B: A, C, E
C: F
D: A, G
E: A, B, G
F: C
G: D, E, H
H: G, I
I: H

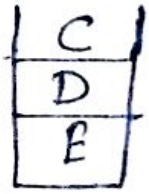
Step 1 : Start with A :

Step 2 : Pop A & push E, D, B



Step 3 : Pop B & push ~~E~~
as A & ~~E~~ status is 3 now not 1.

Step 4 :- Pop C & Push F

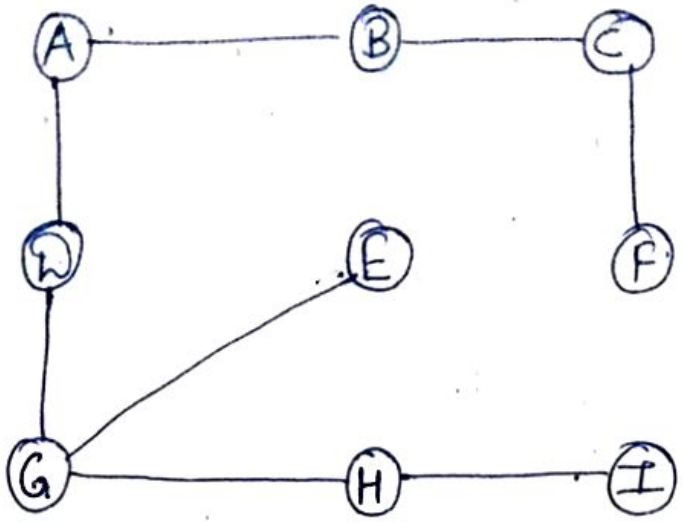


Step 5 : Pop F, Now nothing is left to push as neighbours of F as C is already visited.

Step 6 : pop D and Push G

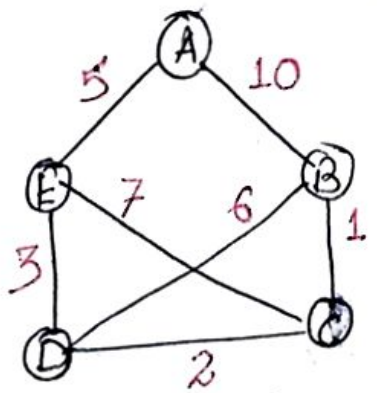
Step 7 : pop G & push H

Step 8 : pop H & push I

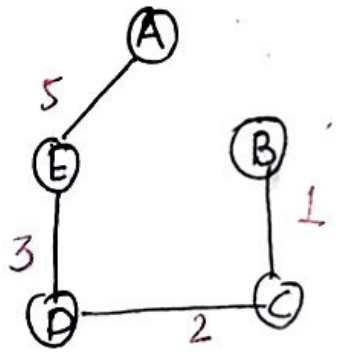


Minimum Spanning Tree :-

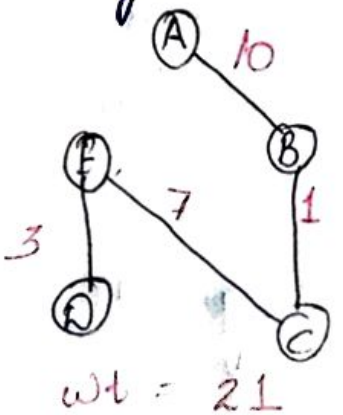
* A spanning tree of a graph G is a subgraph which is basically a tree & containing all the vertices of G but no circuit.
* A minimum spanning tree of a weighted connected graph G is a spanning tree with minimum or smallest weight.
* Weight of the tree is defined as the sum of weights of all its edges.



Graph G.



wt = 11



wt = 21

* To solve the problem of finding minimum spanning tree, few algo. have been designed. Two of them are :-

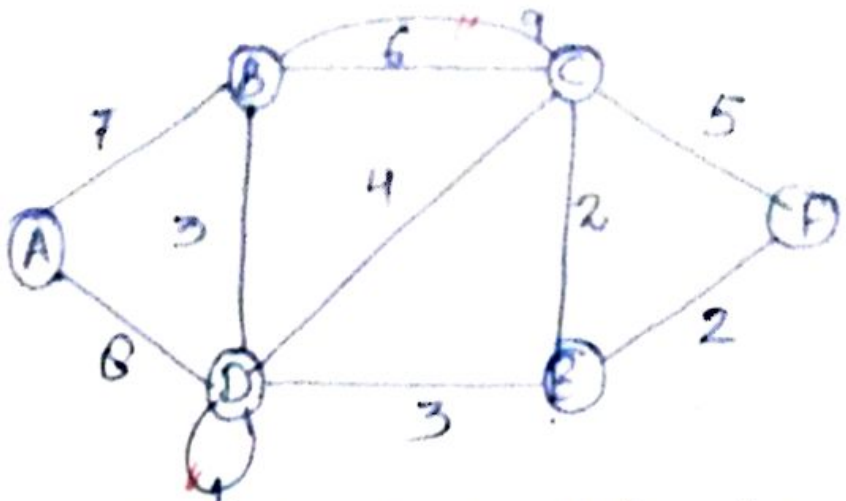
1) Kruskal's algorithm
2) Prim's algorithm

* Kruskal's algo. to find the minimum cost spanning tree uses the greedy approach.
* In this algo. an edge is selected in such a manner that it contains a min. weight & upon adding that it doesn't include any cycle.

Steps for Kruskal's algo. :-

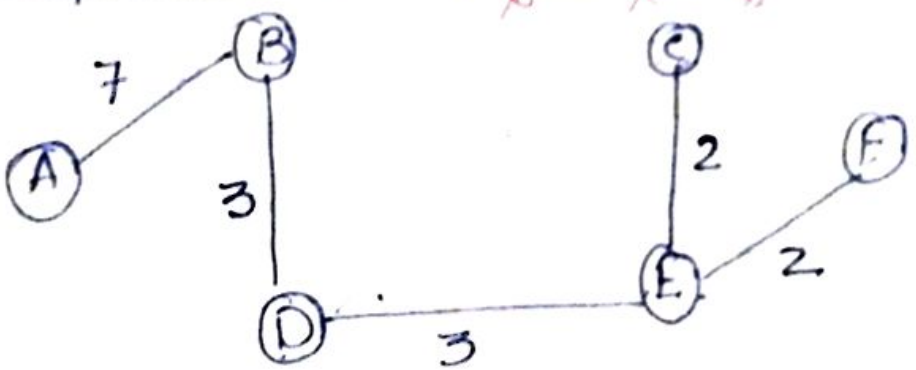
* Remove all the self loops.
* Remove all the parallel edges. Keep the one which has the least cost associated and remove all others.
* Arrange all edges in their increasing order of weight.
* Add the edge which has the least weightage

Example :



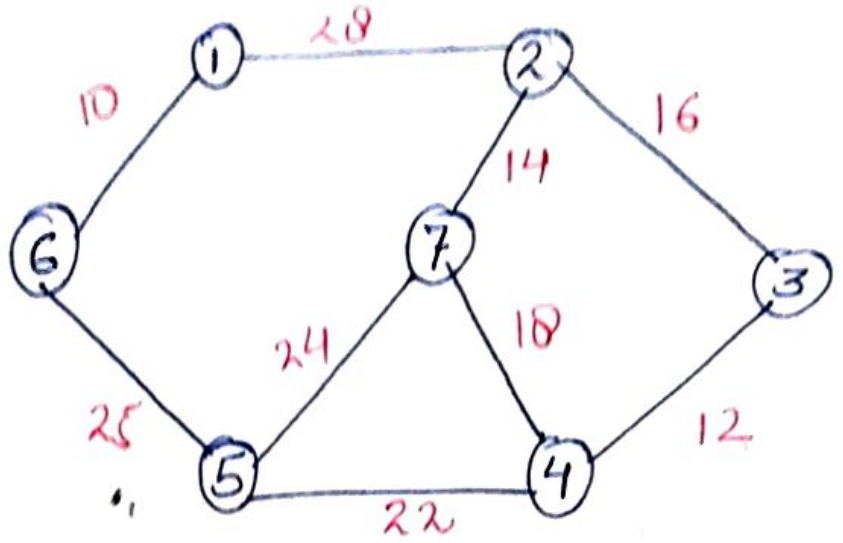
Remove all self loops & parallel edges.

C,E	E,F	D,E	D,B	C,D	C,F	B C	A,B	A,D
2	2	8	3	4	5	6	7	8
				X	X	X		X

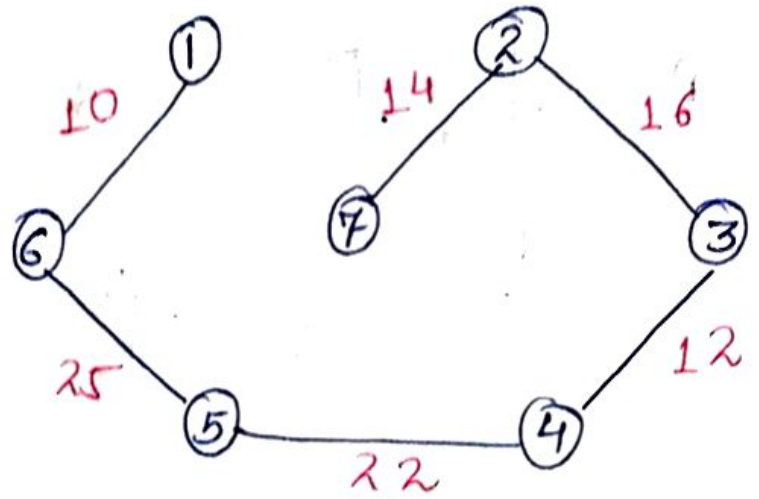


Practice Questions on Kruskal's / Prim's

Q(1)

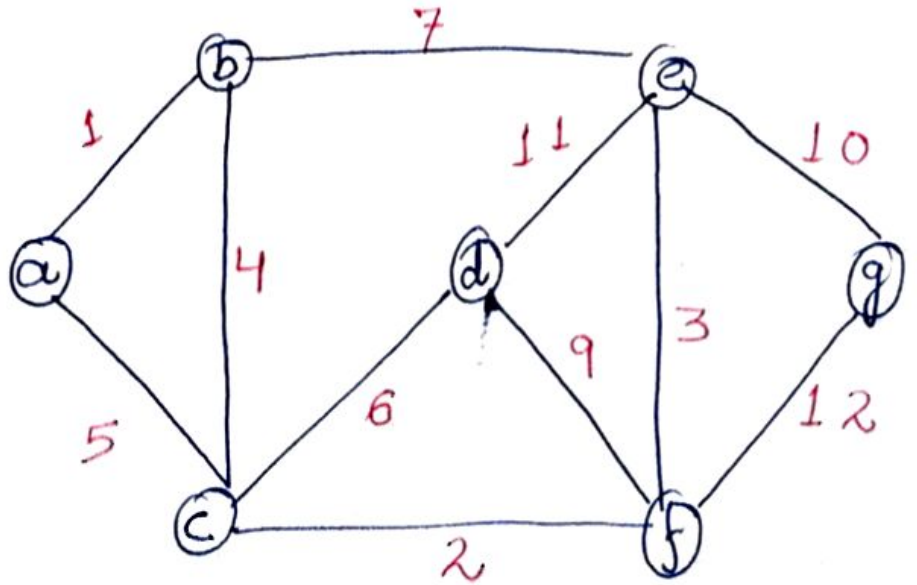


Ans

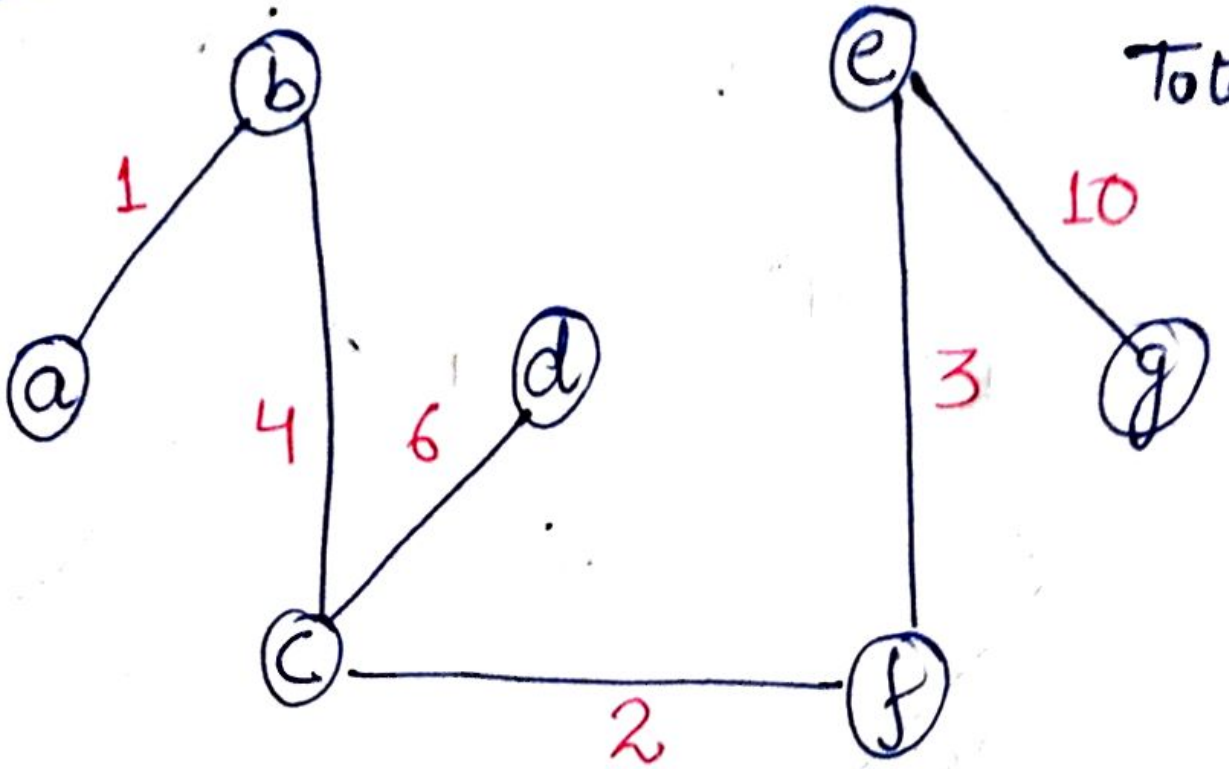


Total = 99

Q(2)



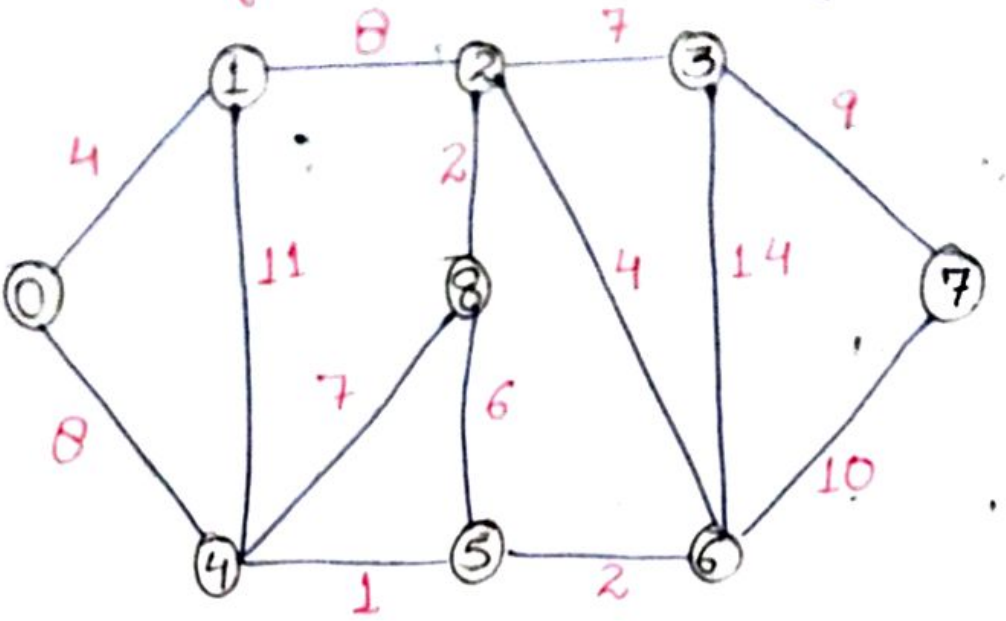
Ans



Total = 26

Dijkstra Algorithm

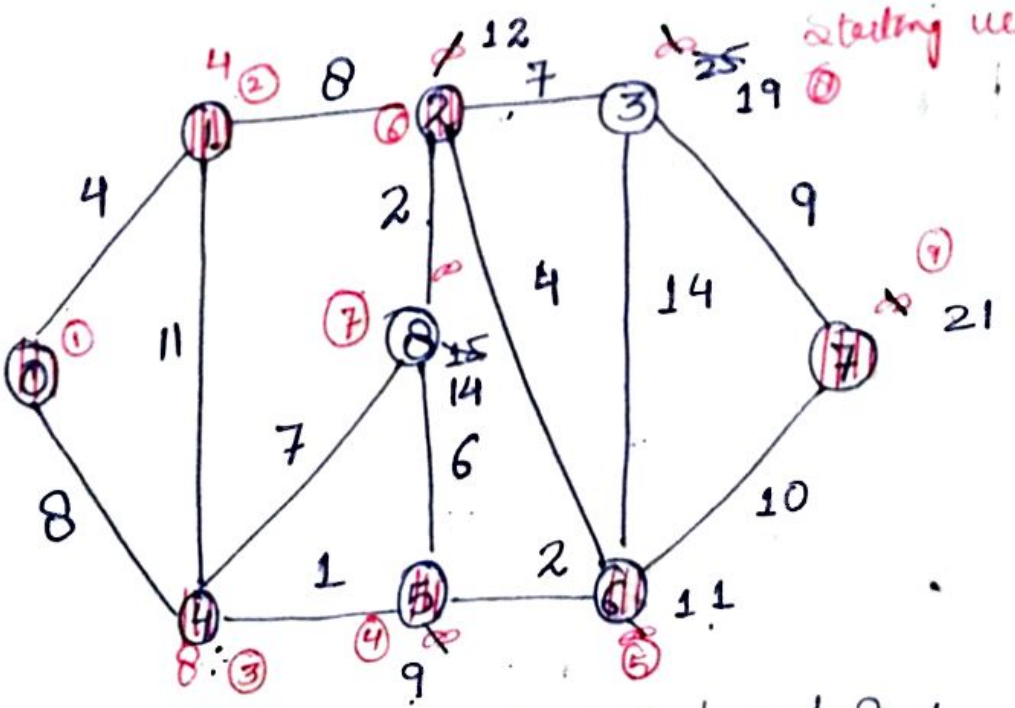
Q1



Q2

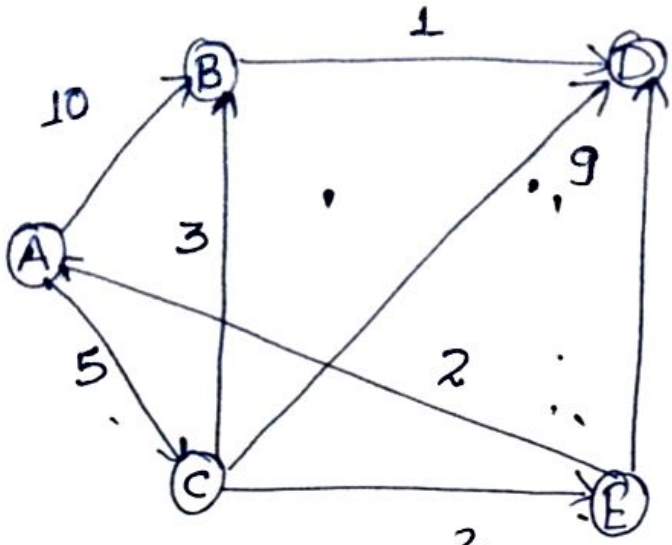
Ans

Starting vertex ⓪



	1	2	3	4	5	6	7	8
0	4	12	19	8	9	11	21	14

Q 2 :-



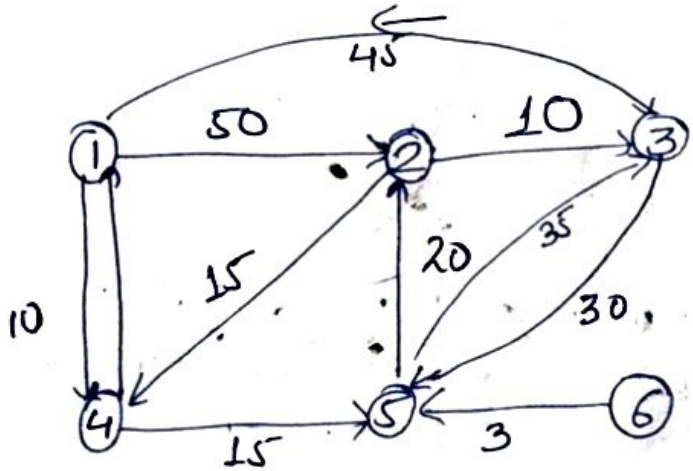
$d(u) + c(u,v) < d(v)$
then $d(v) = d(u) + c(u+v)$

A	B	C	D	E
C	10	5	∞	∞
E	8		14	7
B	8		13	
D			9	

Path :- A to D

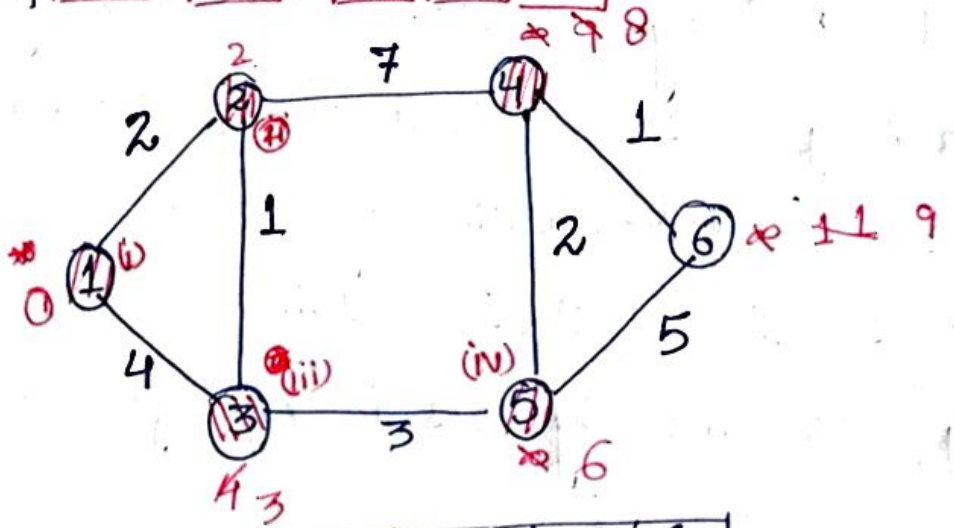
D B C A

Q 3 :-



1	2	3	4	5	6
4	50	45	10	∞	∞
5	50	45	10	25	∞
2	45	45	10	25	∞
3	45	45	10	25	∞
6	45	45	10	25	∞

Q 4:-

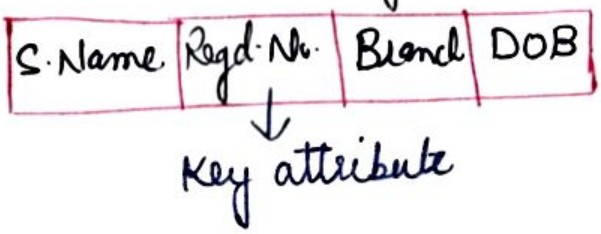


	2	3	4	5	6
1	2	3	8	6	9

Hashing :-

* This searching technique is independent of the no. of elements.
* As in searching, linear search takes O(n) time and Binary Search takes O(lg n) time but hashing is a searching technique which is independent of the number of elements. i.e; O(1) time.
* Hashing is mainly used in file management.

Note :- File : collection of several similar records
Records :- Collection of attributes or fields.



* Suppose F is a file of n records with a set K of keys which can uniquely identify the records in F.
* Let the file F is stored in the memory by means of a table T of m memory locations & L is the set of memory addresses for the locations in T.

* K is the set of n Keys.

⇒ The hash function H is a function from set K of Keys to the set L of memory addresses.

$$H : K \rightarrow L$$

Hash Function :-

Two major criteria to select a hash function are:

(i) It should be very easy and quick to compute.

(ii) It should uniformly distribute the hash addresses throughout the set L so that there are a minimum no. of collisions.

* There are three popular hash function :-

1) Division method
2) Midsquare "
3) Folding "

① Division Method :- Choose a number m larger than the number n of keys of K. (The no. m is usually chosen to be a prime no. or a no. without small divisors, since this frequently minimizes the no. of collisions)

* The hash function is defined by;

$$H(k) = k \pmod m \quad \text{or}$$

$$H(k) = k \pmod m + 1$$

* Here $k \pmod m$ denotes remainder when k is divided by m. The second formula is used when we want the hash addresses to range from 1 to m rather than 0 to m-1.

② Midsquare Method :- The key k is squared. Then the hash function H is defined by;

$$H(k) = l$$

* where l is obtained by deleting digits from both ends of k^2.

* We emphasize that the same positions

of k^2 must be used for all of the keys.

③ Folding Method :- The key k is partitioned into a no. of parts $k_1, k_2 \text{ —— } k_r$, where each part, except possibly the last has the same no. of digits as the required address. Then the parts are added together ignoring the last carry. i.e.,

$$H(k) = k_1 + k_2 + \text{ — } k_r$$

where the leading digit carries, if any, are ignored. Sometimes for extra "milling", the even no. parts, k_2, k_4 — are each reversed before addition.

Example :-

Consider the company in following example, each of whose 68 employees is assigned a unique 4-digit employee number. Suppose L consists of 100 two-digit addresses 00, 01, 02 — 99. We apply the above hash functions to each of the following employee numbers.

3205, 7148, 2345

(a) division method :- lets choose $m = 97$ then

$H(3205) = 4$ [3205 mod 97 = 4]

$H(7148) = 67$ [7148 mod 97 = 67]

$H(2345) = 17$ [2345 mod 97 = 17]

Note :- In case the memory addresses begin with 01 rather than 00, we choose the fn $H(k) = k \pmod m + 1$ to obtain

$H(3205) = 5$, $H(7148) = 68$, $H(2345) = 18$

(b) Mid Square Method :-

K :	3205	7148	2345
K^2 :	10272025	51093904	5499025
H(K) :	72	93	99

Note :- counting from R to L, 4th & 5th digits are chosen.

(c) Folding Method :- Chopping the key k into two parts & adding yields the following hash addresses.

$H(3205) = 32 + 05 = 37$

$H(7148) = 71 + 48 = 19$ ("1 is ignored)

$H(2345) = 23 + 45 = 68$

alternatively one may want to reverse the second part before adding:

$H(3205) = 32 + 50 = 82$
$H(7148) = 71 + 84 = 52$
$H(2345) = 23 + 54 = 77$

Collision Resolution :-

* Suppose we want to add a new record R with key k to our file F. but suppose the memory location address H(k) is already occupied. This situation is called collision.

* Collisions are almost impossible to avoid. e.g., Suppose a student class has 24 students & suppose the table has space for 365 records.

* One random hash function is to chose the students birthday as the hash address. although load factor $\lambda = 24/365 \approx 7\%$. is very small.

Note :- The ratio of no. n of keys in K (which is the no. of records in F) to the no. m of hash addresses in L. $\lambda = n/m$ is called as load factor.

* The efficiency of a hash function with a collision resolution procedure is measured by the average no. of probes (key comparisons) needed to find the location of the record with a given key k.

* Specifically we are intrested in two quantities :-

$S(\lambda)$ = Avg. no. of probes for successful search.

$U(\lambda)$ = Avg. no of probes for unsuccessful search.

(1) Open Addressing : (a) Linear Probing

* Suppose that a new record R with key k is to be added to the memory table T, but the memory location with hash address $H(k) = h$ is already filled, One way to resolve the collision is to assign R to the first available location following $T[h]$.

Note :- We assume the table T with m locations is circular, so $T[1]$ comes after $T[m]$.

Accordingly, with such a collision procedure we'll search for the record R in the in the table T by linearly searching the locations T[h], T[h+1] - — until finding R or an unsuccessful search. This procedure of collision resolution is called linear probing.

e.g., A table has 11 memory locations

T[1], T[2] - — T[11]

Suppose F consists of 8 records & Hash addresses

A,	B,	C,	D,	E,	X,	Y,	Z
4,	8,	2,	11,	4,	11,	5,	1

Suppose 8 records are entered into the table in the above order. Then file F will appear in memory as follows:

Table T :-

X	C	Z	A	E	Y		B		D	
1	2	3	4	5	6	7	8	9	10	11

Address :-

Q:- 3, 2, 9, 6, 11, 13, 7, 12 (use division method)

$h(k) = 2k + 3$

$m = 10$

0	13
1	9
2	12
3	
4	
5	6
6	11
7	2
8	7
9	3

Key	location		probes
3	(2×3+3) % 10	9	1
2	(2×2+3) % 10	7	1
9	(2×9+3) % 10	1	1
6	(2×6+3) % 10	5	1
11	(2×11+3) % 10	5	2
13	(2×13+3) % 10	9	2
7	(2×7+3) % 10	7	2
12	(2×12+3) % 10	7	6

(b) Quadratic Probing :-

The main disadvantage of linear probing is that the records tend to cluster i.e., appear next to one another. To minimize clustering quadratic probing is used.

$(a + i^2)$ % n where i = 0 to m-1

0	13
1	9
2	
3	12
4	
5	6
6	11
7	2
8	7
9	3

for 11 :-
(5 + 1) % 10 = 6

for 13 :-
(9 + 1²) % 10
= 0

for 7 :-
(7 + 1²) % 10 = 8

Key	location		probes
3	(2×3+3) % 10	9	1
2	(2×2+3) % 10	7	1
9	(2×9+3) % 10	1	1
6	(2×6+3) % 10	5	1
11	(2×11+3) % 10	5	2
13	(2×13+3) % 10	9	2
7	(2×7+3) % 10	7	2
12	(2×12+3) % 10	7	5

(c) **Double Hashing** :- Here a second Hash function H' is used for resolving a collision. As;

Suppose a record R with Key k has the Hash addresses $H(k) = h$ and $H'(k) = h' \neq m$. Then we linearly search the location with addresses.

$$h,\ h+h',\ h+2h',\ h+3h' \text{ ---}$$

Note : if m is a prime no., then the above sequence will access all the locations in the table T.

Chaining

* Chaining involves maintaining two tables in memory. First of all, as before, there is a table T in memory which contains the records in f except that T now has an additional field LINK, which is used so that all records in T with the same Hash address h may be linked together to form a linked list

Example :- Suppose we have 8 records with the following hash addresses.

Record :	A	B	C	D	E	X	Y	Z
H(k) :	4	8	2	11	4	11	5	1