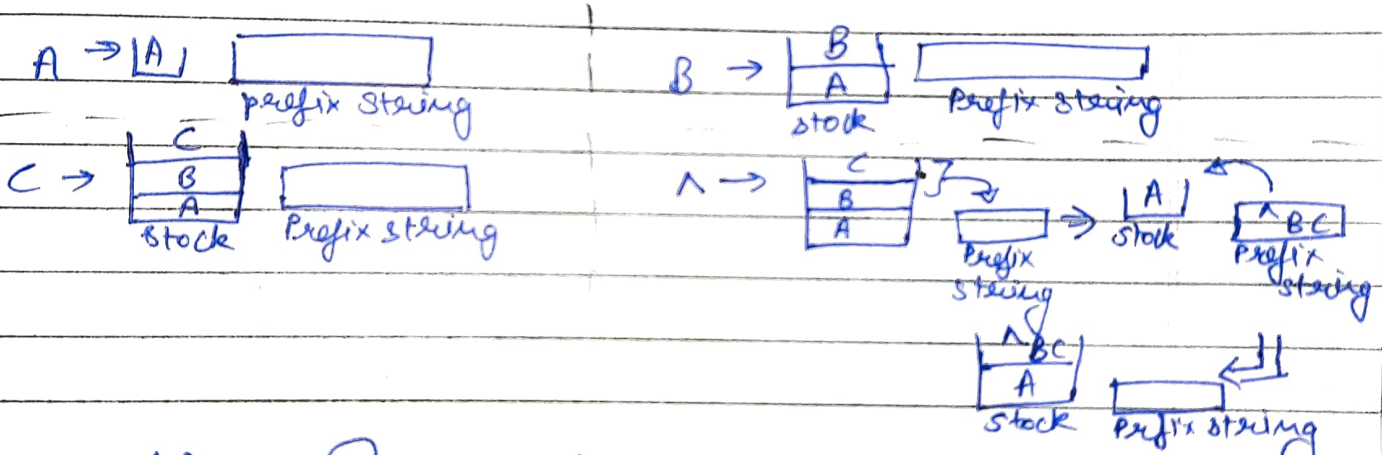
End term Exam.

BCS-204

Q-1(i)

Ans 1(i) Now, for the conversion we will scan the character one at a time from left to right. If it is an operand, we will push it to stack. And if it is an operator, we will pop two operands from stack, form the prefix string and then push the prefix string to the stack. Now,

Given → ABC^+D* ES^+



Since, scanning - remaining element → +^+A^BCD^ES

becomes result of Postfix to Prefix

Q1.(ii)
Ans 1.(ii)

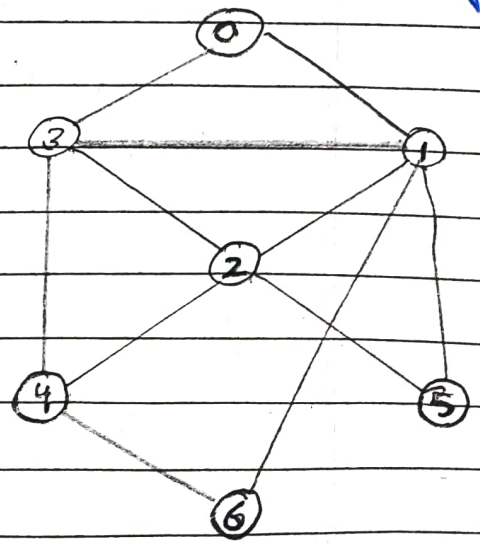
Breadth first search (BFS) is an algorithm that is used to graph data or searching tree or traversing structures. The full form of BFS is the Breadth first search. The algorithm efficiently visits and marks all the key nodes in a graph in an accurate breadthwise fashion. This algorithm selects a single node (initial or source point in graph) and then visits all the nodes adjacent to the selected node. BFS accesses these nodes one by one.

Once the algorithm visits and marks the starting node then it moves towards the nearest unvisited nodes and analyses them. Once visited, all nodes are marked. These iterations continues until all the nodes of the graph have been successfully visited and marked.

* Example.

- Step-1 → You have taken a graph paper of seven number ranging from 0-6

1) Marked 0 as visited
2) Insert 0 to the queue
3) Transverse the un-visited adjacent nodes which are 3 and 1



- Step-2 → 0 has been marked as a root node
  1) Root node = 0

- Step-3 → 0 is visited, marked, and inserted into the queue data structures.

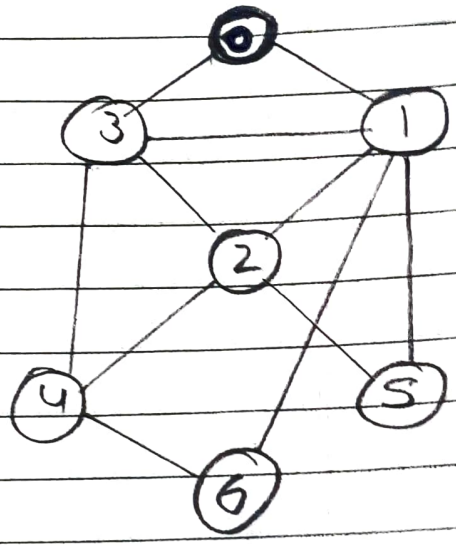
Queue :

0	1	2	3	4	5	6

~~0~~	~~1~~	~~2~~	~~3~~	~~4~~	~~5~~	~~6~~

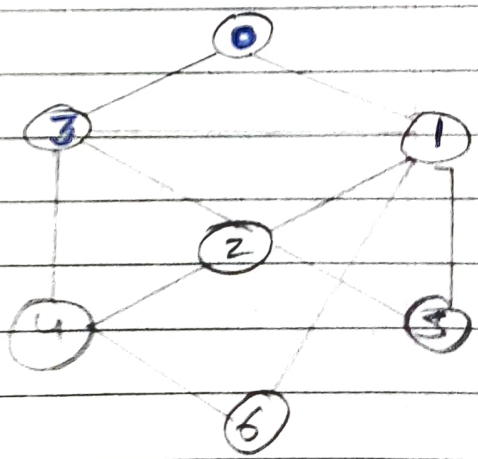
Delete values from queue and print as result.

Result = 0, 1, 2, 3, 4, 5, 6



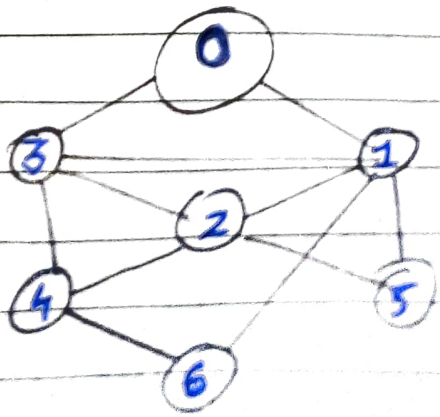
- Step 4 → Remaining 0 adjacent and unvisited nodes are visited marked, and inserted into the queue

» Visited 3 and 1 in any sequence and mark them as visited and add them to the queue



- Step 5 → Traversing iterations are repeated until all nodes are visited.

» Visit all adjacent and un-visited nodes of the previous node and iterate until all visited.



Q.2(i)

Ans2(i) // program to insert a node at given position in circular linear list.

```
#include <stdio.h>
#include <stdlib.h>
struct node {
  int data;
  struct node *next;
} *head;

void insert (int data, int pos);

int main ()
{ int n, data, pos, node data;
  head = NULL;
  printf ("Enter the total number of values in list: ");
  scanf ("%d", &n);
  struct node *prev node, *New node;
  if (n >= 1)
     head = (struct node*) malloc (size of (struct node));
printf ("Data at node 1 : ");
  Scanf ("%d", &data);
  head -> data = data;
  head -> next = NULL;
  prev node = head;
  for (int i = 2, i <= n, i++)
  {
     new node = (struct node*) malloc (size of (struct node));
  printf ("data at node %d", i);
  scanf ("%d", &data);
```

```
new node -> data = data;
    new node -> next = NULL;
    prev node -> next = New node;
    prevnode = newnode
  }
  prevnode -> next = head;
  }
}
printf ("Enter position at which node is to be inserted");
  Scanf ("%d", & pos);
  printf ("Enter data to be inserted as %d position:");
  Scanf ("%d"; & node data);
  insert (node data, pos);
  return 0;
  }
void insert (int data, int pos);
{       struct node * new node, * current;
        int i;
          if (head = NULL)
          printf ("list is empty");
else.
{ new node = (struct node *) malloc (size of (struct node));
  new node -> data = data;
  printf ("Element %d is inserted at index %d", data, pos);
  current = head;
        for (i = 2; i <= pos; i++)
  current = current + next;
  new node -> next = current, next;
  current -> next = newnode; }}
```

Q-2(ii)

Ans(ii) $f(s) = 0$, $\max(f(6), 0) = 0$, $i = 2$
$f(3)_{new} = \max(f(s), 0) + i = 0 + 2 = 2$

$f(s) = 2$, $\max(f(s), 0) = 2$ $i = -3$
$f(s)_{new} = \max(f(s), 0) + i = 2 - 3 = -1$

$f(s) = -1$, $\max(f(s) = 0) = 0$, $i = 2$
$f(s)_{new} = \max(f(s), 0) + i = 0 + 2 = 2$

$f(s) = 2$, $\max(f(s), 0) = 0$, $i = 2$
$f(s)_{new} = \max(f(s), 0) + i = 2 + 1 = 1$

$f(s) = 1$, $\max(f(s), 0) = 1$ $i = 4$
$f(s)_{new} = \max(f(s), 0) + i = 1 + 4 = 5$

* So $f(s)$ is 5

Q3(i)

Ans(i) Sorted → 5, 6, 8, 10, 12, 13, 15, 50] → sorting after algorithm
Pass -3 → 5, 8, 6, 10, 12, 13, 16, 50]

Algorithm

```
Begin bubble sort (list)
for all elements of list
  if list [i] > list [i+1]
swap (list [i], list [i+1])
  End if
  end for
return list
end bubble sort.
```

	10	14	10	4	50	8	6	13	12
Pass 1:	10	4	14	8	6	13	12	50	
Pass 2:	4	10	8	6	13	12	14	50	
* Pass 3:	4	8	6	10	12	13	14	50	
Pass 4:	4	6	8	10	12	13	14	50	
Pass 5:	—— sorted ——								
Pass 6:	—— sorted ——								
Pass 7:	—— sorted ——								

Q3(ii)

Ans3(ii) Given Array :-

A[0] = 8
A[1] = 18
A[2] = 18
A[3] = 35
A[4] = 45
A[5] = 70
A[6] = 86
A[7] = 90
A[8] = 99
A[9] = 120

So, there are total 10 numbers,

so, low = 0

High = 9

$$Mid = low + \frac{High}{2} \Rightarrow Mid = \frac{0+9}{2} = 4$$

A[4] = 45 is the mid Now, the key = 86

But Key > mid

So, mid + 1 checking now

So, low = 5

High = 9

$$Mid = \frac{low + High}{2} = \frac{5+9}{2}$$

Mid = 7

A[7] = 90 which is mid

Now, mid > key

so mid - 1 checking now

low = 5

High = 7

$$Mid = \frac{low + High}{2} = \frac{5+7}{2}$$ Mid = 6

A[6] = 86

Now mid is 86, which was required to find, Hence 4 comparisions are required to find 86 in given array

Algorithm for Binary searches

```
#include <stdio.h>
int main()
{ int i, low, high, mid, n, key, array
[100]; printf("Enter no. of element ->\n");
scanf("%d", &n);
printf("Enter %d integers ->\n", n);
for(i=0; i<n; i++)
scanf("%d", &array[i]);
printf("Enter the value to find ->\n");
scanf("%d", &key);
low = 0;
high = n-1;
mid = (low + high)/2;
```

```
while (low <= high)
{
  if (array[mid] < key)
        low = mid + 1;
      else if (array mid == key)
  {
    printf ("%d found at array position -> %d\n
key mid);
break;
}
else
    high = mid - 1;
     mid = (low + high) / 2;
}
if (low > high)
  Print ("Not found ! %d isn't present in
the array \n", key) return 0;
}
```

0-4(i)
dig4(i)

```
** include <conio.h>
** include <stdio.h>
** include <stdlib.h>
typedef struct node.
  {
    int info;
      struct node * next;
} node type;
node type* insert (node type *);
      void sort (node type*);
  void main ()
{
```

```
Node type * left = Null
Node type * right = NULL;
   int ch, c;
printf ("Enter 1 for insert \n Enter 2 for
sort \n
   Enter 3 for exit);
do
{
   printf ("\n choice : ");
   scanf ("%d" & ch);
   switch (ch)
   {
```

Case I

```
      right = insert (right);
         if (left == NULL)
      { left = right
      }
break;
```

case 2:

```
sort (left).
```

case 3

```
   exit (1):
   default:
   printf ("invalid choice");
    break;
}
{ int n;
   Node type * p;
   P = (node type *) malloc (sizeof (node type));
   printf ("\n Enter no. : ");
```

```
        scanf ("%d", &x);
        if (p != NULL)
    {
        p -> info = x
        p -> next = NULL;
if (r == NULL)
    {   r = p;
    }
    else
    {  r -> next = p;
        r = p;
    }
    return (r)
    } else
    {
     printf ("Not enough memory");
    }
    }
    void sort (node type = 1)
{
    node type *t;
    node type *s;
     int x;
    t = p;
    while
        (t != NULL)
    {   s = t -> next;
        while (s != NULL
{   if(t -> info > s -> info)
    {  x = t -> info ;
```

```
        t -> info = s -> info;
        s = info = x;

    s = s -> next
}
t = t -> next
}
t = l;
while (t != null)
{
printf ("%d -> t -> info);
        t = t -> next
}
}
```

Q 4(ii)

Ans 4(ii) Given elements are →

35, 50, 40, 25, 30, 80, 78, 20, 28

So, to construct an AVL tree, we will be adding each element one by one and will be checking balanced factor of each node before adding another one. If the node is imbalanced then we will balance that node and then add the next element. Now, the Balance factor is given by:

Balance factor = height of left sub tree − height of right sub tree

$$bf = h_l - h_r = \{-1, 0, 1\}$$

or

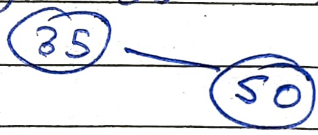
$$|bf| = |h_l - h_r| \leq 1$$

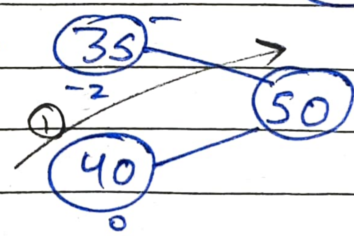
If these equations is not satisfied by a node then it is imbalanced and we have to balance it using rotations.

Note → Balance factor is only applicable to 3 nodes at a time.

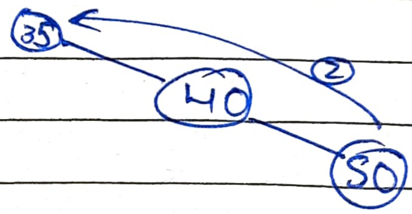
Inserting elements

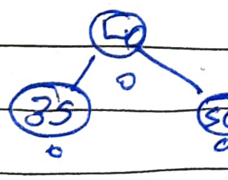
1) 35 → 

2) 50 → 

3) 40 → 

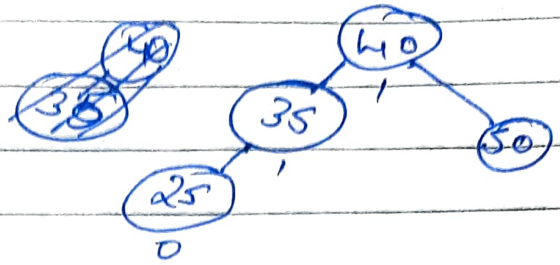
Now here node 35 is imbalanced so we will perform rotations



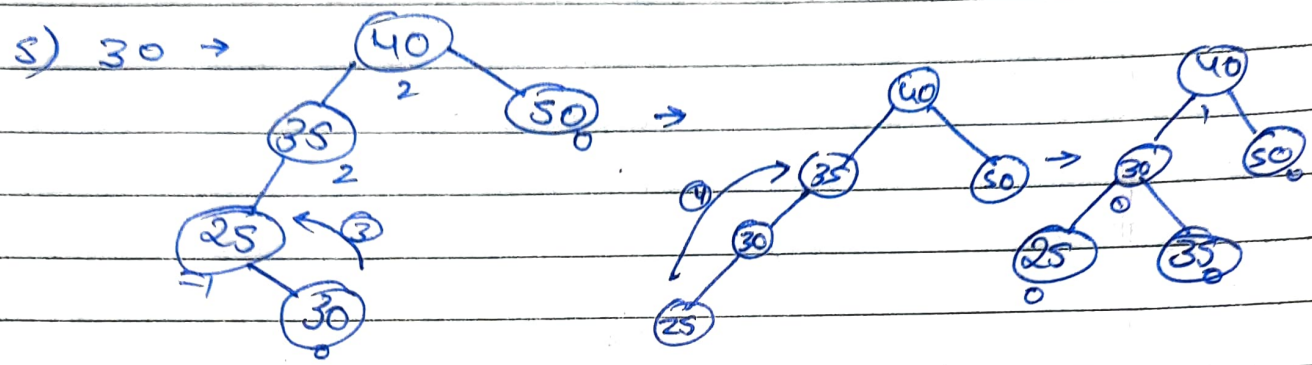
⇒ 

Now this one is balanced

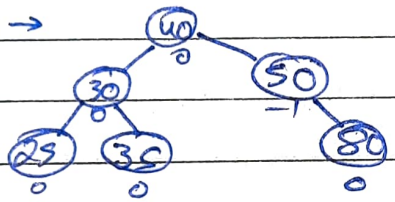
4) 25 →



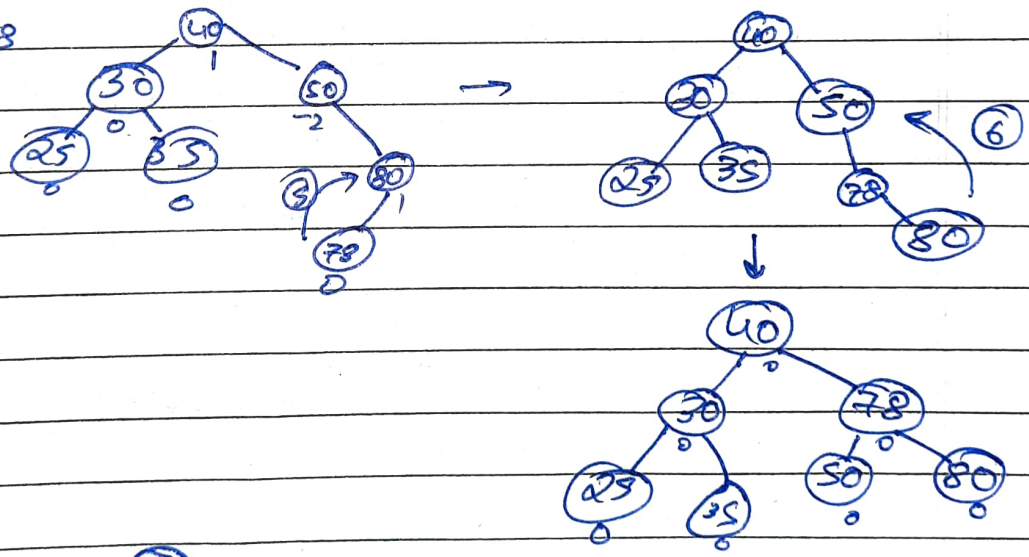
5) 30 →



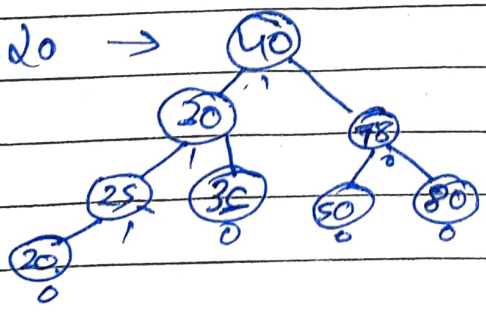
6) 80 →



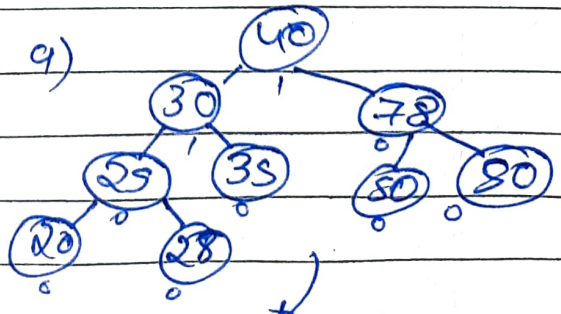
7) 78



8) 20 →



9)



Required Balanced AVL tree

⇒ It took 6 rotations to construct the AVL tree for the given element

Now, for post - order traversing we follow left right Root so post order for the above AVL tree transformed is.

Post order → 20, 28, 25, 35, 30, 50, 80, 78, 40

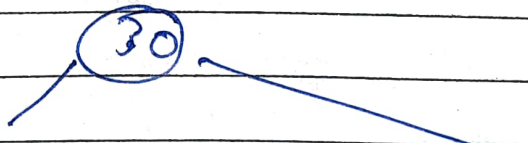
0.5 ; diss; j)

POT 15, 10, 23, 25, 20, 35, 42, 39, 30
(L & Root

IOT → 10, 15, 20, 23, 25, 30, 35, 39, 45
(L and R root).

↳ Just arranging in ascending order.

From POT → Root is 30,



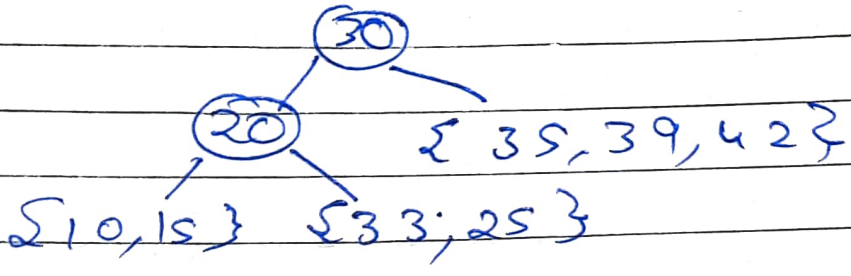
{10, 15, 20, 23, 25} {35, 39, 42}

First we will construct left subtree so.

POT — 15, 10, 23, 25, 20, 35, 42, 39, 30

IOT — 10, 15, 20, 23, 25, 30, 35, 39, 42

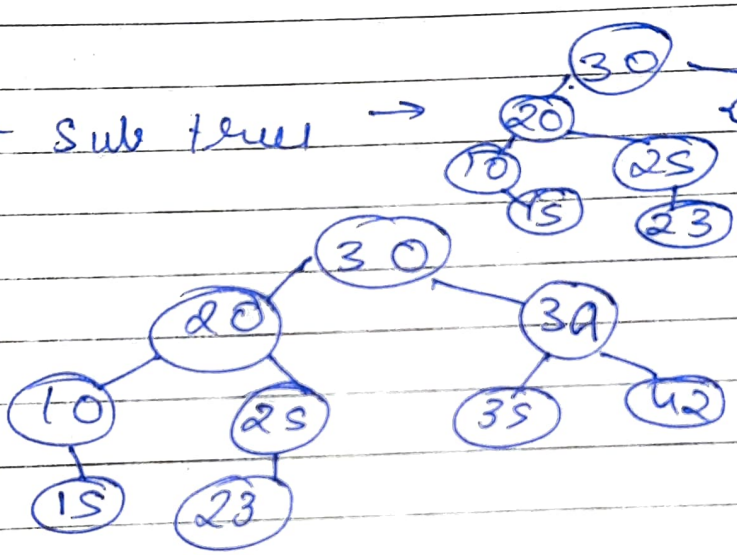
L R



POT - 15, 10, 23, 25, 20, 35, 42, 39, 30.

IOT - {(10, 15) 20 (23, ∞)} 30 {35, 39, 42}

Right Sub tree →

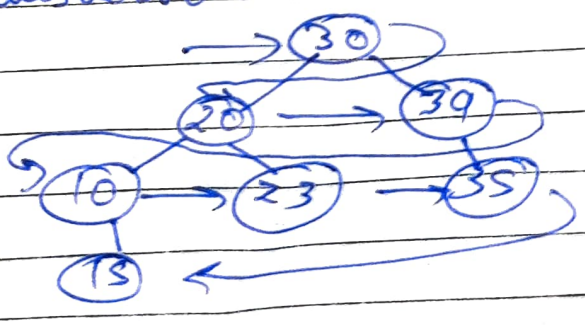


↳ Required binary tree

By deleting nodes
↳ Pre order of tree is.
Pre order → 30, 20, 10, 15, 23, 25, 35.

Level order transversal.

30, 20, 39, 10, 23, 35, 15



Q5(ii)

Ans5(ii) Advantages of using circular queue over linear queue are.

•> Easier for insertion-detection :- In the circular queue, elements can be inserted easily if there are vacant locations until it is not fully occupied, whereas in the case of a linear queue insertion is not possible once the rear reaches the last index even if there are empty locations present in the queue

•> Efficient utilization of memory :- In circular queue, there is no wastage of memory because it uses the unused space, and also there is proper use of memory circular queue as compared to a linear queue.

•> Performing operations are easy :- Now in linear queue, First in first out is followed, so the element that is inserted first is to be deleted firstly But in circular queue, the rear and front are not fixed. So, the order of insertion and deletion can be change, which turns out very useful.

Function to transversed the element of circular queue

```
void transverse()
{   int i = front;
    if (front == -1 && rear == -1)
{   printf ("\n queue is empty.");
 } else
{   printf ("\n Elements in the queue are:");
    while (i <= rear)
 {   printf ("%d", queue[i]);
     i = (i+1) % max;
 } }
```