Topic — QUEUE — Unit No. 04

★ What is a Queue?

→① Queue is a linear data structure, where the elements are arranged sequentially.

② In queue, the operations - insertion and deletion are performed at two different positions / ends.
The insertion is performed at one end, which is called as "rear" end whereas, deletion is performed at another end which is called as "front" end.

③ Inserting a new element in queue is termed as "enqueue" operation and deletion operation in queue is termed as "dequeue" operation.

④ In queue, enqueue and dequeue operations are performed based on FIFO (First In First Out) Principle.
i.e. Queue follows FIFO structure.

Topic

Unit No.

In short,

"Queue data structure is a collection of similar data items in which insertion and deletion operations are performed based on FIFO principle."

ex-

1. Railway Ticket Counter
2. Car Wash Center, etc.
3. Cashier Line
4. People on escalator, etc.

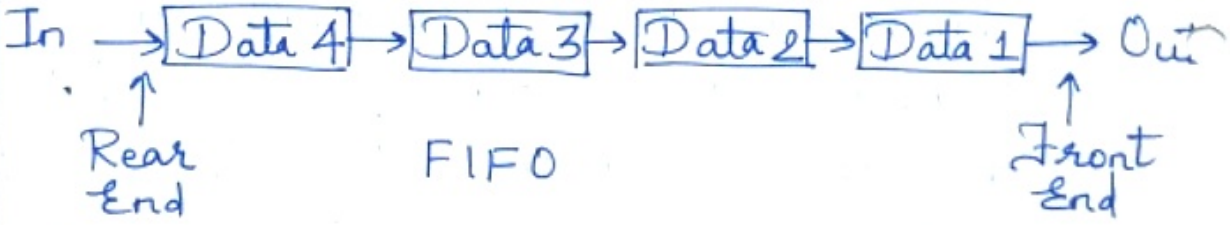


fig. Queue

Topic

Unit No. 04

* Operations performed on queue are :-

1. Queue() :- creates a new empty queue. It needs no parameters and returns an empty queue.

2. enQueue(item) :- Used to add new item to the rear end of the queue. It needs 'item' as its parameter and returns nothing.

3. deQueue() :- Used to remove an item from the front end of the queue. It needs no parameters, but returns the item being removed.

4. isEmpty() :- Used to check whether the queue is empty. Needs no parameters and returns a boolean value.

5. size() :- Returns the number of items in the queue. Needs no parameters and returns an integer.

Topic

Unit No. 04

☆ Implementation of Queue :-

1) Using Arrays.
2) Using Linked List.

* Implementation of Queue using Arrays:-

(1) Queue can be implemented using one-dimensional array (1D).

(2) Queue implemented using array stores only fixed number of data values.

(3) Implementation of queue using array is very simple and easy.
Just define a 1D array of specific size and insert or delete the values into the array using FIFO principle with the help of the variables 'rear' and 'front'.

(4) Initially, both variables rear and front are set to -1.

Topic

Unit No. 04

While inserting a new element into the queue array, increment rear by 1 and then add new element at that (queue[rear]) position.

While deleting an element from the queue array, increment front by 1 and then delete the element which is at front position of the array.

★ Queue Operations using Array :—

First, follow the below steps to create an empty queue.

1) Declare the global variables, the user defined functions, define a constant SIZE with specific value.

2) Create a 1D array with above defined SIZE (int queue[size]).

3) Define two integer variables 'rear' and 'front' and initialize both with '-1'.

```
int rear = -1, front = -1;
```

Topic Unit No.

4) enQueue (value) :-

- Check whether queue is FULL (rear == SIZE - 1)
- If it is FULL, then display "Queue is FULL" and terminate the function.
- If it is NOT FULL, then increment rear by 1 and insert a new element at a location pointed out by rear.

```
rear = rear + 1;
queue[rear] = value;
```

5) deQueue() :-

- Check whether queue is EMPTY. (front == rear)
- If it is EMPTY, then display "Queue is Empty" and terminate the function.

Topic

Unit No. 04

- If it is NOT EMPTY, then increment the front by 1, display the value that is deleted and return that value i.e. queue[front].

Then check whether both front and rear are equal (front == rear), if it is TRUE, then set both to -1.

```
if (front == rear)
   printf ("Queue Empty");
else
{
      front = front + 1;
      value = queue[front];

   printf ("Element deleted is %d", value);
      return(value);
}
```

Topic Unit No.

6) display ():-

- Check whether queue is EMPTY. (front == rear)
- If EMPTY, then display "Queue is Empty" and terminate the function.
- If NOT EMPTY, then define an integer variable 'i' and set it to :-
  i= front + 1;
- Display 'queue[i]' and increment i by 1.
- Repeat the same until i reaches to rear (i<= rear)

```
if( front == rear)
  printf (" Queue Full");
else
{  for(i= front+1; i<= rear; i++)
     printf(" %d", queue[i]);
}
```

Topic

Unit No. 04

★ Types of Queue :-

1) Simple Queue
2) Circular Queue
3) Double - Ended Queue
4) Priority Queue.

★ Circular Queue :-

A circular queue is the extended version of a simple queue where the last element is connected to the first element. Thus forming a circle - like structure.

Circular queue resolves the memory wastage problem generated in simple queue.

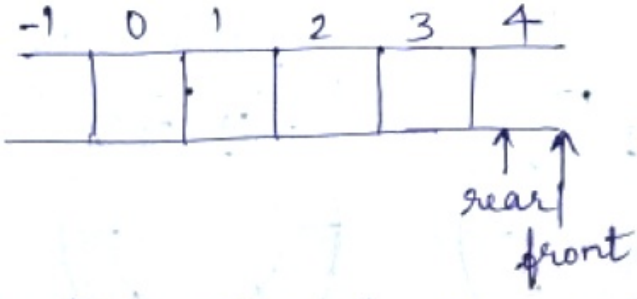


fig. : Simple Queue

Topic

Unit No.

- Its a linear data structure in which the operations are performed based on FIFO principle and last position is connected back to first position to make a circle. Also termed as "Ring Buffer."

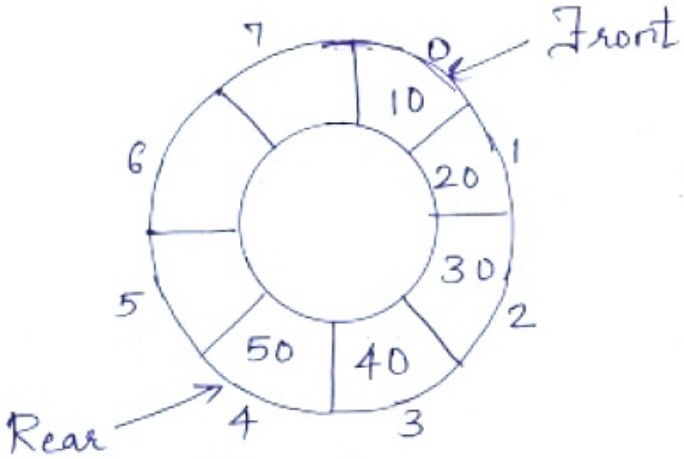
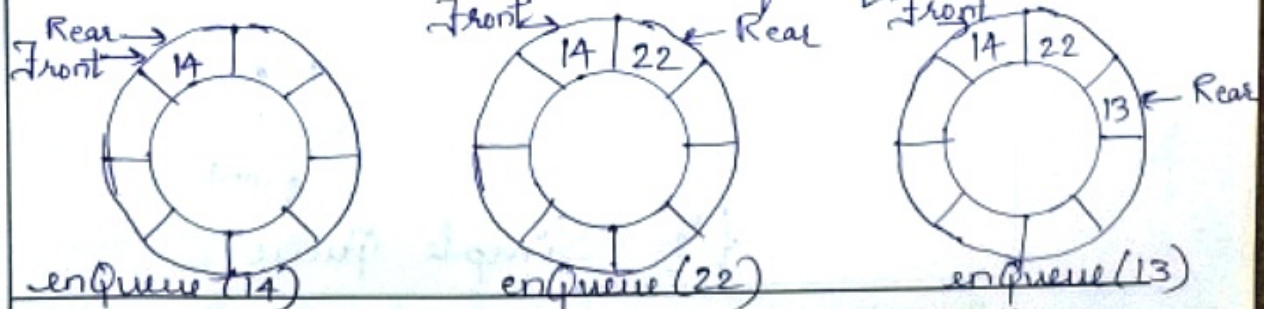


fig. Circular Queue.

- In simple queue, we can insert the elements until queue becomes full. But once queue is full, we cannot insert the next element even if there is a space in front of queue.



★ How a Circular Queue Work :-

- It works by the process of circular increment i.e. while incrementing rear & front variables, if we reach the end of the queue, then we again start from the beginning of the queue.
- The circular increment is performed by modulo division with the queue size.
  i.e. Let size = 5
  If rear reaches the queue end,
  i.e. if (rear + 1 == 5) ⟶ overflow,
  then
  rear = (rear + 1) % size equation will set rear to 0th location.

★ Circular Queue Operations :-

The circular queue works as follows -

- 2 variables :- rear and front.
- rear :- tracks the last element of the queue.
- front :- keeps track of the first element of the queue.
- initially set both rear & front to -1.

1] Enqueue Operation :-

- For the very first element insertion in the circular queue, check if both variables are -1, if yes, set both variables to 0 and assign the element at rear end.
- Circularly increment rear by 1. If it matches front then print message as "Circular Queue Full".
- Else, increment rear circularly and add the new element in the position pointed out by rear.

2] Dequeue Operation :-

- Check if queue is empty. If yes, print "Queue Empty".
- If both rear and front variables are at same position then first delete the element at front and set both variables to -1.
- Else, delete the element at front end and increment front circularly by 1.

```
#define size 5
int queue[size];
int rear = -1, front = -1;

void enQueue (int element)
{
  if (rear == -1 && front == -1)
  {
      rear = 0;
      front = 0;
      queue[rear] = element;
  }
  else if ((rear+1) % size == front)
  {
      printf ("Circular queue full");
  }
  else
  {
    rear = (rear + 1) % size ;
    queue[rear] = element;
  }
}
```

```
void deQueue()
{
  if (rear == -1 && front == -1)
  {
    printf ("Circular Queue Full");
  }
  else if( front == rear)
  {
   printf (" Element dequeued is %d ", queue[front]);
   rear = -1;
   front = -1;
  }
  else
  {
  printf(" Element dequeued is %d ", queue [front]);
   front = (front +1) % size ;
  }
}


void display()
{
   int i;
  if (rear == -1 && front == -1)
     printf (" Circular Queue Empty");
```

```
    else
    {
     for (i= front ; i!= rear ; i= (i+1) % size)
      { printf ("%d\t", queue [i]);
      }
    }
}

int main()
{
    deQueue ();
    enQueue (10);
    enQueue (20);
    enQueue (30);
    display ();
    enQueue (40);
    enQueue (50);
    enQueue (60);
    display ();
```

```
deQueue();
deQueue();
display();
enQueue(70);
display();
enQueue(80);
return(0);
}
```

Topic Unit No. 04

☆ Deque - Double Ended Queue :-

What is Deque in Data Structure?

→ Deque is a more generalized version of a linear queue. In linear queue or simple queue, insertion is done from rear end and deletion from front end only.
But, in deque, both insertion and deletion can be performed at both of its ends. That's why it is called a Double Ended Queue (Deque).

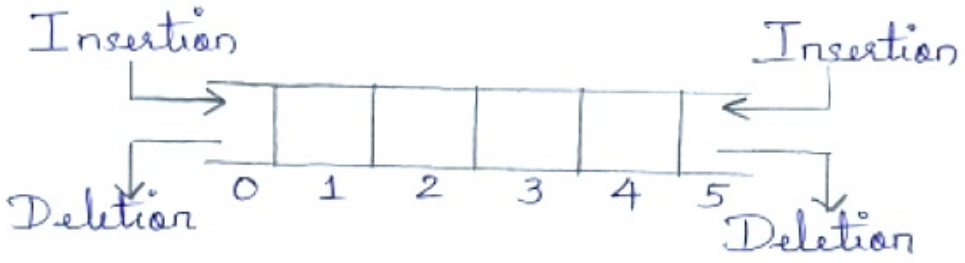


fig. Deque

The above image represents how different operations take place at both ends of the deque.

Topic

Unit No. 04

- Deque is a data structure that inherits the properties of both stack and queue.
- Additionally, the implementation of this data structure (deque) requires constant time, i.e.,
time complexity = O(1).
- Also, inheriting the properties of stack & queue, deque does not follow LIFO and FIFO principles.

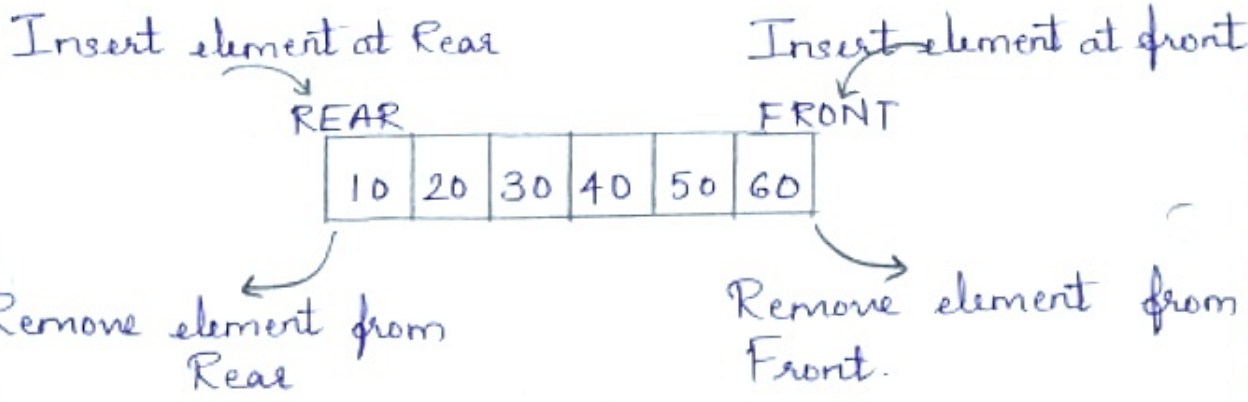


fig. A Deque.

Topic

Unit No. 04

★ The Deque Abstract Data Type :-

- A deque is structured and an ordered collection of items where items are added and removed from either end, either rear or front.
- The deque operations are :-

a) Deque() :- creates a new empty deque. It needs no parameters and returns an empty deque.

b) addToFront(item) :- adds a new item to the front of the deque. It takes one parameter as item and returns nothing.

c) addToRear(item) :- adds a new item to the rear of the deque. It takes one parameter as item and returns nothing.

d) removeFromFront() :- removes item from front end of the deque. It needs no parameters but returns the item deleted. The deque is modified.

e) removeFromRear() :- removes item from the rear end of the deque. It needs no parameter and returns the item deleted. The deque is modified.

f) isEmpty() :- tests to see whether the deque is empty. It needs no parameters and returns a boolean value.

g) isFull() :- tests to see whether the deque is full.

h) size() :- returns the number of items in the deque. It needs no parameters and returns an integer.

★ Types of Deque :-

① Input - Restricted Deque :-

In this deque, insertion is done at only one end whereas deletion is performed at both ends.

Topic

Unit No. 04

fig. Input Restricted Deque ..

(2) Output - Restricted Deque :-
In this deque, deletion is done at only 1 end whereas insertion is performed at both ends.

fig. Output Restricted Deque ..

★ Implementation of Deque :-
The deque can be implemented using either —
(1) Circular Queue
(2) Doubly - Linked List.

Topic Unit No.

Here, we use circular queue to implement deque.

① addToFront (item) :-

Before starting the actual code, we need to initialize both the variables to -1.
i.e

```
int front = -1;
int rear = -1;

void addToFront (int item)
{
/* as we are implementing deque using circular queue, we before inserting any new element, we always have to check for queue full condition. So, here we check for circular queue full condition
*/
if ((front == 0 && rear == size-1) ||
        (front == rear+1))
{   pf("Deque full"); }
```

Topic Unit No.

```
/* if the deque is empty, which means
both variables are -1, then in this
case, first initialize both variables
to 0 and then add new element
at 0th position from front end */.

else if ((front == -1) && (rear == -1)) //initially
 {
     front = 0
     rear = 0
     deque[front] = item

 }

/* if front is pointing at 0th location,
then reinitialize it to point at
last location and insert element at
front */
else if( front == 0)
 {      front = size-1
       deque[front] = item
 }
```

Topic Unit No.

```
/* if front is pointing to some other
location except 0th and last, then just
decrement front by 1 and insert element
at front */

else
{
    front = front - 1
    deque[front] = item
}
} // end of addToFront(item)
```

(2) addToRear(item) :-

```
/* check for circular queue full condition
before inserting any new element */

void addToRear(item)
{
if((front == 0 && rear == size - 1) ||
        (front == rear + 1))
    pf("Deque Full")
```

Topic

Unit No. 04

```
else
{
    rear ++
    deque[rear] = item
}
}// end of addToRear(item)
```

(3) displayDeque() :-

```
void displayDeque()
{
/* display deque elements from front to
rear */

    int i = front

while( i is not equal to rear)
{
    ~~pf~~ print the elements of deque
and increment 'i' using circular
incrementation.
}// end of while
```

Topic

Unit No.

```
/* first time insertion from rear end */
else if ((front == -1) && (rear == -1))
   {
        rear = 0
        deque[rear] = rear      /* initialize rear to 0 and add element to rear end */
   }

/* if rear is at last location then reinitialize it to 0 and insert new element at rear */

else if (rear == size -1)
 {
      rear = 0
      deque[rear] = item
 }

/* if rear is pointing to any location except 0th and last, then first increment rear by 1 and add element at rear */
```

P.T.O

Topic Unit No. 04

```
else
{
    rear ++
    deque[rear] = item
}
}// end of addToRear(item)
```

③ displayDeque() :-

```
void displayDeque()
{
/* display deque elements from front to rear */
  int i = front
  while( i is not equal to rear)
  {
      print the elements of deque
  and increment 'i' using circular
  incrementation.
  }// end of while
```

Topic

Unit No. 04

printf the element at rear end

}// end of displayDeque()

(4) removeFromFront() :-

```
void removeFromFront()
{
// check for deque empty
 if((front == -1) && (rear == -1))
     print "Deque Empty"

/*check if both variables are pointing
to same location. If yes, then delete
element from front and reinitialize
both variables to -1 */

 else if(front == rear)
 {
     print the deleted element from front end.

     front = -1
     rear = -1
 }
```

Topic Unit No. 04

```
/* if front is at last location, then delete
the element from front end and initialize
front to 0. */

else if (front == (size-1))
{
  print the deleted element front end.
     front = 0
}

/* if front is pointing to some other location
except first and last then, simply delete
the element from front end and increment
front by 1 */

else
{
   print the deleted the element from front
     front++
}
}//end of removeFromFront()
```

Topic

Unit No.

(5) removeFrontRear :—

```
void removeFromFront()
{
//check for deque empty
if((front == -1) ff(rear == -1))
    print "Deque Empty"
else if(front == rear) // both at same location
{
  print element deleted from rear end
    front = -1
    rear = -1
}
/*if rear is at 0th position, then delete the
rear end element and traverse back to
last position */
else if(rear == 0)
{  print the deleted element from rear end
    rear = size-1
}
// if rear points to some other location except
// 0th and last then, delete rear end element
//and decrement rear by 1.
```

Topic Unit No.

```
else
{
  printf("print the deleted element from rear")
  rear--
}
}//end of removeFromFront()
```

* Now, in main() function, call the functions.

```
int main()
{
    addToFront(10);
    addToFront(20);
    addToRear(30);
    addToRear(40);
    addToRear(50);
    displayDeque();
    removeFromFront();
    removeFromRear();
    displayDequeue();
    return 0;
}
```

* Applications of Deque :—

① Palindrome Checker

② Multiprocessor Scheduling

Topic

Unit No. 04

★ Priority Queue :-

- A priority queue is a special type of queue in which each element is associated with a priority value.
- Elements are served on the priority basis. That is, higher priority elements are served first (removed first).
- However, if elements with the same priority occur, they are served according to their order in the queue.

★ Assigning Priority Value :-

- Generally, the value of the element itself is considered for assigning the priority.
- For ex - The element with the highest value is considered the highest priority element.

P.T.O

Topic Unit No.

However, in other cases, we can assume the element with the lowest value as the highest priority element.

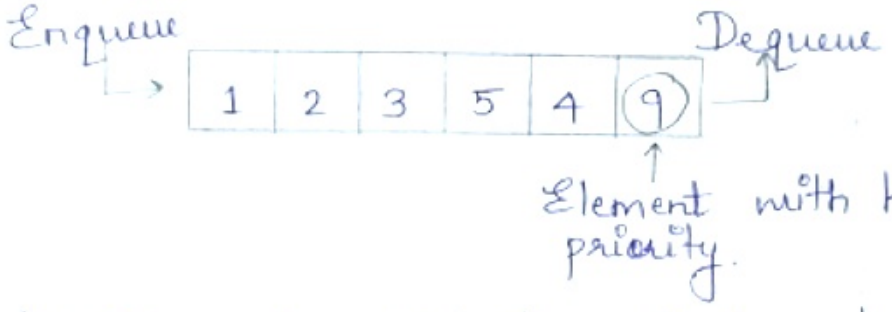


fig. Removing highest priority element from Priority Queue.

☆ Properties of Priority Queue :—

1. Every element has a priority associated with it.
2. An element with high priority is dequeued before an element with low priority.
3. If two elements have same priority, they are served according to their order in the queue.

Topic

Unit No. 04

★ Types of Priority Queue :-

- Ascending order priority queue.
- Descending order priority queue.

1) Ascending Order Priority Queue :-

- Here, the element with lower priority value is given a higher priority in the priority list.
- ex- Let the numbers in the priority queue be :- 2, 6, 8, 9, 11, arranged in ascending order.

Here, 2 is the smallest number, therefore it will get the highest priority in a priority queue, hence will be served (removed) first.

2) Descending Order Priority Queue :-

- Here, the element with higher priority value is given a higher priority in the priority list.
- From the above example, element 11

Topic

Unit No.

has highest priority. Hence, 11 will be served (removed) first.

★ Difference between Normal Queue (Linear) queue and Priority Queue :-

	Normal Queue	Priority Queue
①	No priority attached to the elements in a normal queue.	Elements have priority.
②	Follow FIFO rule.	Elements with higher priority are served first.

★ Implementation of Priority Queue :-

Priority queues can be implemented using:-

1) Arrays
2) Linked List
3) Heap data structures
4) Binary Search Tree (BST).

Topic

Unit No. 04

★ Implementation of Priority Queue using Ordered Array :-

① Enqueue :-
- Insert item according to their priority, lowest priority at the start and highest priority at the end.
- Items are arranged in ascending sorted order of their priority value.

② Dequeue :-
Dequeue item from the end in O(1) time.

③ Peek :-
Return item with highest priority. Last item in the array itself will have highest priority.

★ Pseudo Code :-

```
#include<stdio.h>
#define max 10
```

P.T.O

Prepared by :

Page No. 35

Topic

Unit No.

```
int id = -1;
int elementValue[max];
int elementPriority[max];
int i;
int isEmpty()
{
    if (id == -1)
        return;
}
int isFull()
{
    if (id == max-1)
        return;
}

void enqueue(int data, int priority)
{
  if (!isFull())
  {
    if (id == -1) // first insertion
    {
        id++
        elementValue[id] = data;
    elementPriority[id] = priority;
    return; }
```

Topic Unit No.

```
else
{
   id++;
for(i = id-1 ; i>=0; i--)  //reverse order.
{
if(elementPriority[i] >= priority)
{   elementValue[i+1] = elementValue[i];
elementPriority[i+1] = elementPriority[i];
}
else
{
/* insert item just before where lower
priority index was found */
 elementValue[i+1] = data;
elementPriority[i+1] = priority;
   break
}
}//end of for
}//end of else
}//end of if
}// end of function
```

shift all items rightwards with higher priority than the item we are trying to insert.

Topic

Unit No.

```
int peek()
{
   return id;   // returns highest priority item
}

void dequeue()
{
    id--;
}

void display()
{
  for(i=0; i<=id; i++)
     printf("%d %d", elementValue[i],
                elementPriority[i]);
}

int main()
{
    enqueue(25, 1);
    enqueue(10, 10);
    enqueue(15, 50);
    enqueue(20, 100);
    enqueue(30, 5);
    enqueue(40, 7);
```

Topic Unit No.

```
    printf (" Before Dequeue : ");
    display();
    dequeue();
    dequeue();
    printf (" After Dequeue : ");
    display();
    return 0;
}
```

O/P :-

Before Dequeue :

```
25  1
30  5
40  7
10  10
15  50
20  100
```

After Dequeue :

```
25  1
30  5
40  7
10  10
```

Topic

Unit No.

★ Applications of Priority Queue :-

1. Dijkstra's algorithm
2. Load balancing and interrupt handling in OS
3. Data compression in Huffman code.
4. Stack implementation
5. CPU scheduling.

★ Applications of Queue :-

1. Job Scheduling
2. Multiprogramming
3. Access to shared resources
4. Multilevel Queue scheduling.
5. In simulation
6. As a buffer space.