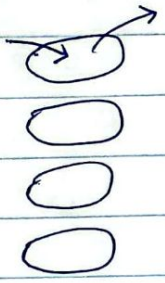
QUEUE

LIFO - Last In First Out



FIFO - First In First Out

Insert an element in to the queue →

step 1 : check if the queue is full
step 2 : If the queue is full, Produce overflow error & exit
Step 3 : If the queue is not full, increment rear pointer to point the next empty space
step 4 : Add data element to the queue location, where the rear is pointering
Step 5 : Return success.

* Queue maintain two data pointers, front and rear.

Algorithm

```
begin Procedure enqueue (data)
if queue is full
   return null
Overflow   endif

rear ← rear + 1
Queue [rear] ← data
return true

end procedure
```

rear position ... element = ... rear = rear + 1 ...

Implementation.

```
Void enqueue (int element) {
  if (isfull() ) {
  printf (" couldn't insert
                data \n" );
  }
  else {
     rear = rear +1;
     queue [rear ]= element;
  // queue [++rear ]= element;
  }
```

Remove an element from the queue.

End (insert) enqueue	(remove) dequeue
queue is full ↳ return overflow	queue is empty ↳ return underflow
++ rear or rear = rear + 1 (insert)	(remove) Front ++ Front = Front + 1

① First Removing element
② Increasing front

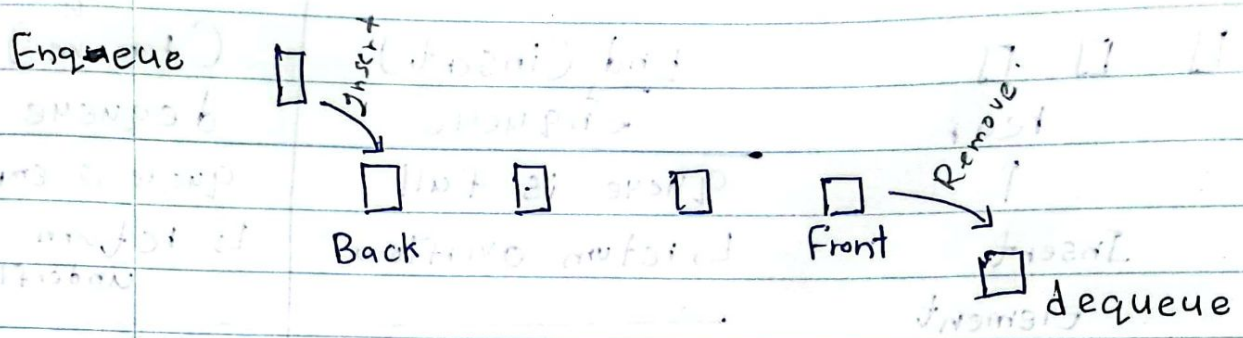
Algorithm

```
begin Procedure dequeue
if queue is empty
  return
underflow endif

data ← queue (front)
front ← Front + 1

return true
return end Procedure
```

Implementation.

```
int dequeue ( ) {
  if (isempty ( )) {
    printf ("Couldn't receive data\n");
  }
  else {
    data = queue [front];
    front = front + 1;
    return data;
    // return stack [top - 1];
  }
}
```

Endqeue & Dequeue



* data මුකුත් නැති වුනාම Front එක 0. Rear එක minus (-) වල
* element (insert) enqueue කලාම front එක 0 ම තියෙනවා Rear එක - වල ඉඳන් + වලට එනවා.

Peek() කියන්නෙ මුලින්ම ඉඳ්ම කරන්න ඉන්න කොට්ටා

Algorithm

```
begin Procedure peek
  return queue [front]
end Procedure
```

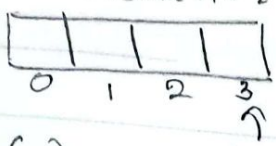
Implementation

```
int peek()
{
  return queue (front)
}
```

```
int peek(){
```

```
int peek () {
  int display;
  display = queue (front);
  return display;
}
```

No of element = 4



IsFull ( )

Algorithm

begin Procedure isFull (No of elements)
 is rear equal to Maxsize - 1
  return true
 else
  return false
 endif
end Procedure

IsEmpty ( )

Algorithm

begin Procedure isempty
 if rear less than 0
  return true
 else
  return false
 endif
end Procedure

```
Public boolean isFull()
{ //size == maxsize
  return (rear == maz size - 1)
}
```

Implementation

```
boolean isfull()
{
  if (rear = Maxsize - 1)
    return true;
  else
    return false;
}
```

Implementation

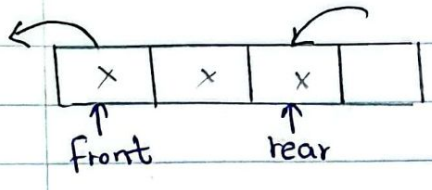
```
boolean isEmpty()
{
  // return rear = -1
  if (rear == -1)
    return true;
  else
    return false;
}
```

```
Public boolean isEmpty()
{ // size == 0
  if (front < 0 || front > rear)
    return true;
  else
    return false;
}
```

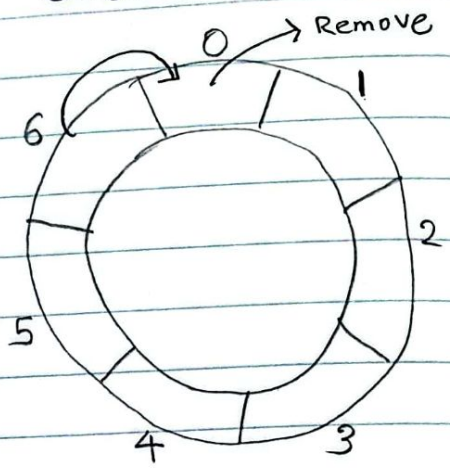
06 th May 2024

Chapter 03.
Circular Queue.

Queue



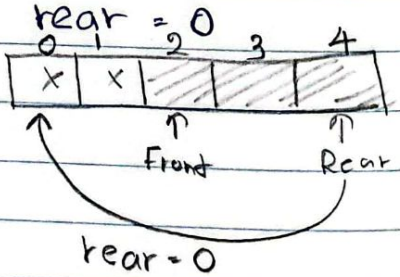
Circular Queue



Front = 0
rear = 6

After Removing,
Front = 1
rear = 6

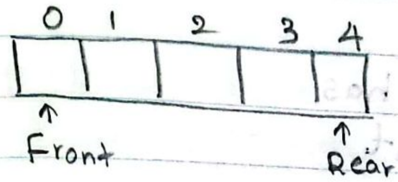
After Inserting,

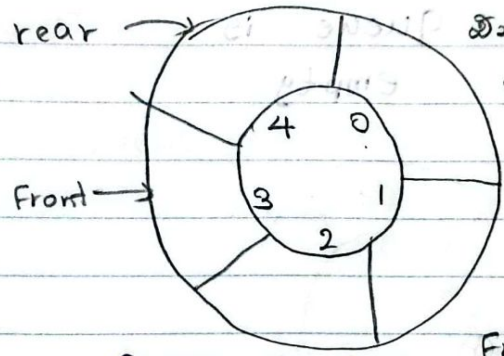


- Circular queue is a linear data structure.
- It follows FIFO principal. It's called "Ring Buffer".
- In circlar Queue rear and front are connecting.
- Element added → rear, Element Deleted → front.
- Double ended → Insert and delete we can do both ended.

09th May 2024

Inserting value into the Circular Queue





rear, front ස්ථාන මාරු වෙන්න පුළුවන්.

rear = size - 1 }
front = 0 } Queue is full

front = rear + 1 ← only Circular queue - Queue is full

Front සහ rear අතර ඉඩක් නැති නිසා.

Algorithm.

step 1: Check whether queue is Full

((rear == size - 1 && front == 0) || (front == rear + 1))

step 2: If it is Full

display " Queue is full! Insert is not possible.

step 3: If is Not full.

then check rear == size - 1 && front != 0

if it is TRUE

then set rear = -1

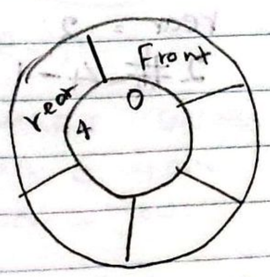
step 4: Increment rear value by one (rear++)

set queue [rear] = value

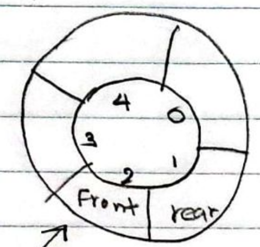
check front == -1

if it is TRUE

then set front = 0



rear = size - 1
front = 0



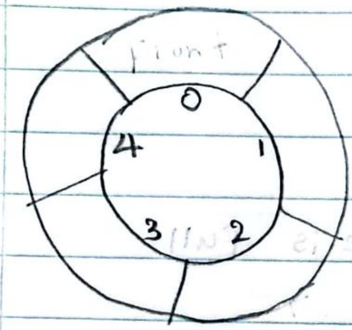
front = rear + 1 ← මේ condition එක

ඇතුළු වෙන්න මේ වගේ ...

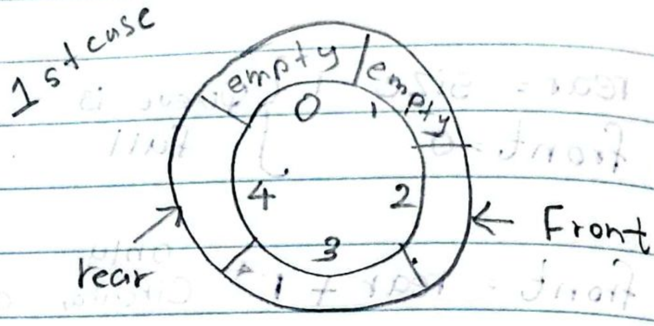
If is it not full.

queue is empty

queue has element

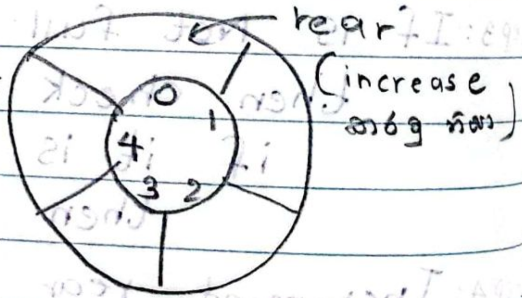


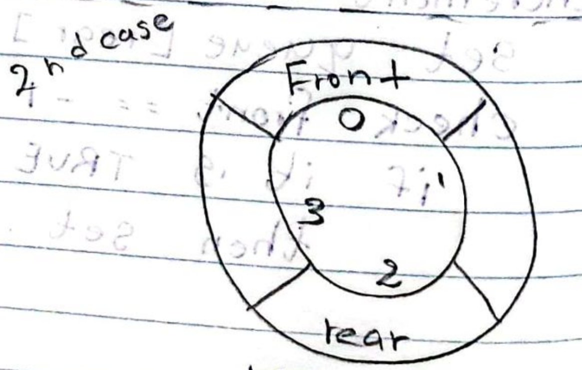
Front = Rear = -1
↳ set 0



ඉස්සෙල්ලාම rear == -1
කරන්න ඕන. මොකද එතකොට
විතරයි 0 index එකට යන්න
පුලුවන්.

rear = -1
↳ rear + 1





rear = 2
2 ≠ 4 - 1 ← size

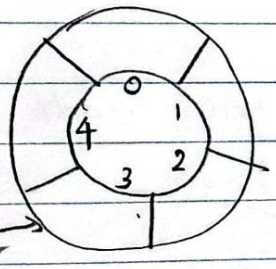
මෙහෙම තියෙනවනම් නැවත
step 3 ඉඳන් step 4
එකට යනවා.

Implementation.

```
public void enQueue (int e)
{   if (isFull())
        System.out.println ("Queue Over flow");
    else
    {
        if (front == -1)
            front = 0;
        rear = (rear +1) % maxSize;
        CirQueueArray [Array] = e;
    }
}
```

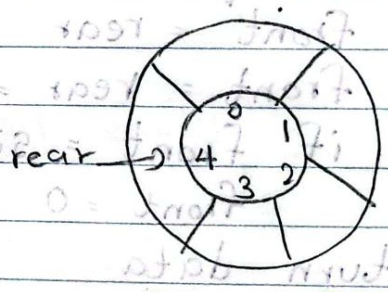
eg:-

If rear = 3



rear = (rear +1) % maxSize
= (3 +1) % 5
= 4

If rear = 4



rear = (rear +1) % maxsize
= (4+1) % 5
= 5 % 5
= 0

Removing a value from the Circular Queue.

In a circular queue,

Algorithm

```
begin Procedure dequeue
if queue is empty
   return
underflow endif
data← queue[front]
if front = rear
    front = rear = -1
    if front = size
        front = 0
return data
end procedure
```

Implementation.

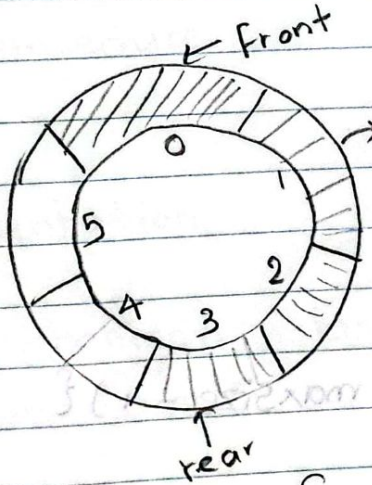
```
public int dequeue() {
   if (isEmpty()){
      System.out.println("Queue
```

Check whether queue is EMPTY
(front == -1 && rear == -1)

If it is Empty
then display "Queue is EMPTY!! Deletion is not possible!!"

If it is NOT EMPTY
Then display queue[front] as deleted element
increment the front value by one (front++)
Then check whether front == SIZE,
if it is TRUE,
the set front = 0
Then check whether both
front -1 and rear and equal (front -1 == rear)
if it TRUE
then set both front and rear to '-1' (front

More than one element (Not empty)

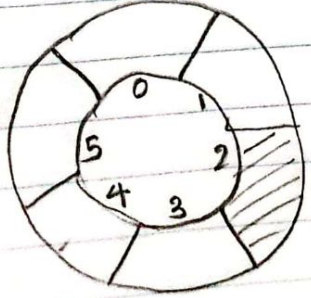


After removing (front)

front = rear (front+1)

Single element (NOT empty) - element in the begin/middle of the array

Before removing,
front = 2, rear = 2
~~front = rear = -1~~



← rear/front

After removing, front = ~~front+1~~ = 3
rear = 2

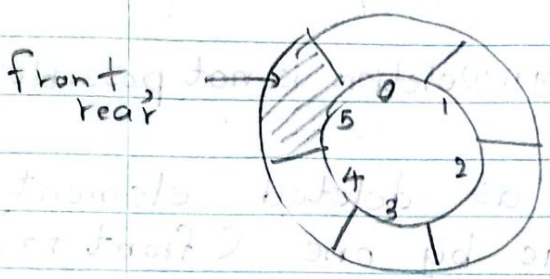
front = rear = -1

1 case NOT satisfied

II case → One element is end of the array.

Before Removing,
front = 5 , rear = 5

After removing,
front = 5 + 1 (front + 1)
= 6



Size = front
∴ then set front = 0

20 th May 2024

IsFull()

Algorithm

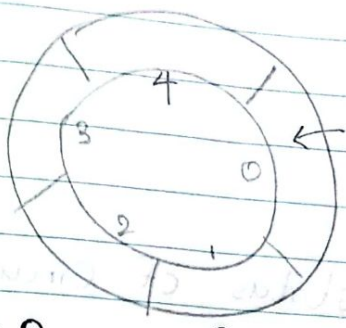
```
public boolean isfull{
   if rear equals to MAXSIZE -1
        return true
   else
        return false
   endif
  end procedure
```

Implementation.

```
public boolean isfull(){
  if(front == 00 && rear == maxsize - 1){
       return true;
  }
  if (front == rear + 1){
       return true;
  }
  return false;
}
```

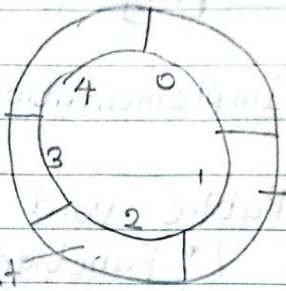
IsFull( )

I case



f = 0 R = -1

II case



Front = 2 Rear = 1

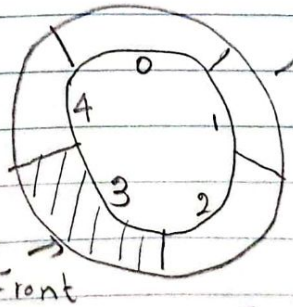
IsEmpty ( )

Algorithm

```
begin procedure isempty
   if front less than 0
      return true
   else
      return false
   endif
end procedure.
```

Implementation.

```
public boolean isEmpty() {
   return ( false == -1 || rear == front );
}
```



After Removing → Rear = 3 , front = 4

R < F ← does not satisfy condition

↳ R = F = -1

No :

Date:

Display

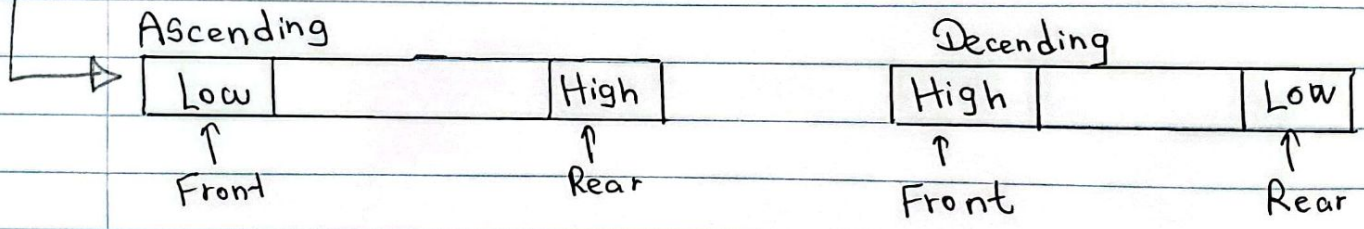
Implementation.

```
Public void display() {
  /* Function to display status of Circular Queue */
  int i;
  if (isEmpty()) {
     System.out.println("Empty Queue");
  }else {
     System.out.println("Front ->" + front);
     System.out.println("Items ->");
     for(i= front; i!= rear; i=(i+1) % SIZE)
        System.out.print(items[i] + " ");
     System.out.println(items[i]);
     System.out.println("Rear ->" + rear);
  }
}
```

Chapter 03 :

Priority Queue.

- Priority Queue is more specialized data structure than queue.
- Like ordinary queue, priority queue has same method but with a major difference.
- In Priority queue items are ordered by key value so that item with the lowest value of key is at front and item with the highest value of key is at rear or ~~vice~~ vice versa.
- So we're assigned priority to item based on its key value
- Lower the value, higher the priority.



- This is a type of queue is an extended version of the normal
- The element with the higher priority will be served first, and if two elements share the same priority then they will be served on the basis of their occurrence, which means the element who entered the queue first will have more priority over others.
- A priority queue is an extended version of the normal queue with the only difference being that its elements have a priority value associated with them.

- So the priority queue has a few properties, which are as follows:

1. Every element of the queue has a priority assigned to it.
2. The element with the highest priority is served first or will be dequeued before the element having low priority.
3. Two elements with the same priority appear then they will be served on the basis of their occurrence. The one who occurred first will be dequeued first.

- In the priority queue, generally, the value of an element is assigned as the priority of that element.

- The element with the largest value will be assigned the largest priority, and similarly, the smallest value will ~~be~~ have the last priority.

- Also the lowest value can have the highest priority and the highest value is having the lowest priority.

- Priority Queue is of two types.
  1. Ascending order Priority
  2. Descending order Priority.

Ascending Order Priority.

- In Ascending order priority queue, the element with least value has the highest priority.

5 ⟶ 7 ⟶ 8 ⟶ 12 ⟶ 19

- In the above queue, elements are arranged as per their priority in ascending order.
- So, if we need to deque (Remove) an element from the above queue 5 will be first element to be dequed.

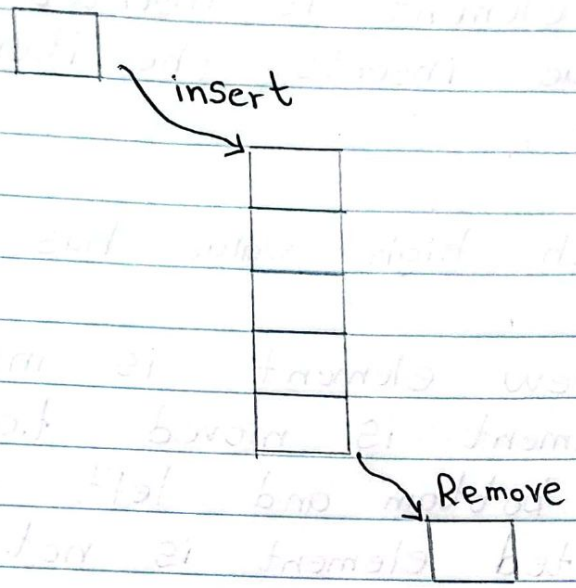
Descending order Priority.

- In descending order priority queue, the element with the highest value has the highest priority

25 ⟶ 17 ⟶ 12 ⟶ 9 ⟶ 5

- In the above queue, elements are arranged as per their priority in desecending order.
- So, if we need to deque an element from the above queue 25 will be the first element to be dequeued.

Priority Queue Representation.



Basic Operations

- Following are the principal methods of a priority Queue.

insert / enqueue - add an item to the rear of the queue.

remove / dequeue - Remove an item from the front of the queue.

- There is few more operations supported by queue which are following.

Peek - returning either maximum or minimum element of the array. It is just returning the value of highest or lowest element without deleting that node.

isFull - check if queue is full

isEmpty - check if queue is empty.

Inserting Value

- Whenever an element is inserted into queue, priority queue inserts the item according to its order.

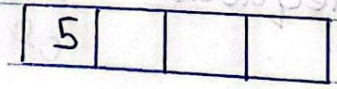
- Here data with high value has low priority.

- Whenever a new element is inserted,
  - The new element is moved to an empty slot top to bottom and left to right.
  - If the inserted element is not in the correct position. It will be compared with the parent element.
  - And if the element is not in the correct order, then elements are swapped until the inserted element gets to the correct position.

29 th May 2024

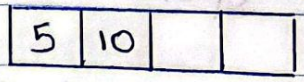
[5, 10, 3, 12]

* insert (5) index = 0



* insert (10)

insert 10 into the empty slot



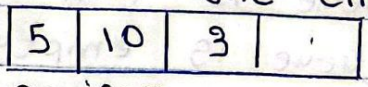
Do the comparison.

5 (current element) < 10 (data has to insert)

No need to swap

* insert (3)

insert 3 into the empty slot



Do the comparison

10 (current element) > 3 (data has to insert)

Need swap

nto queue,
accor ding

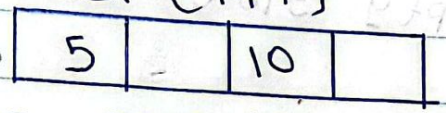
low priority.

rted,
an empty
right.
in the
red with

the correct
ed until the
orrect position.

May 2024

Shift the current element (index = i) 10 into the next index (i+1)

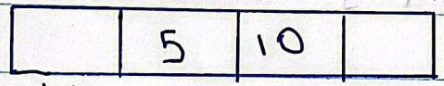


Do the comparison

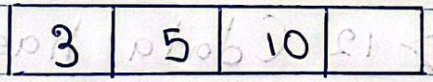
5 (current element) > 3 (data has to insert)

Need swap

Shift the current element (index = i) 5 into the next index (i+1)

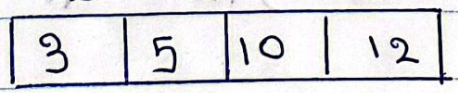


Insert the 3 in the current location



* insert (12)

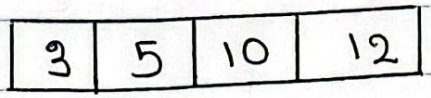
insert 12 into the empty slot



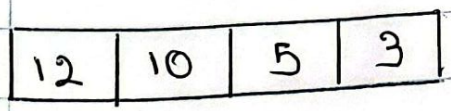
Do the comparison

10 (current element) < 12 (data has to insert)

No need swap



← Ascending order priority Queue



← Descending order Priority Queue

No :

Date :

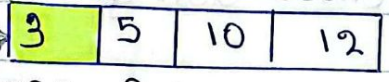
■ - Highest Priority

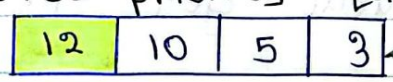
Case 1 :

Highest Priority in the front of the array

Case 1.1 : Ascending order priority queue (Highest Priority = small value)

Case 1.2 : Descending order priority queue (Highest priority = High value)



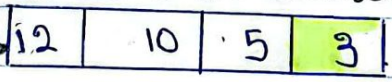


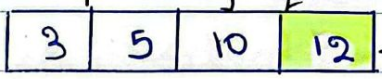
Case 2 :

Highest priority in the end of the array

Case 2.1 : Ascending order Priority queue (Highest priority = small value)

Case 2.2 : Descending order priority queue (Highest priority = High value)





Implementation

```
Static void insert (int data){
  int i = 0;
  int itemcount = 0;
  if (! isFull ()) {
     // if queue is empty, insert the data
  if ( itemcount == 0 ) {
   //
   intArray [0] = data;
   itemCount ++;
  }
  else {
    // Start from the right end of the queue
     for (i = temCount -1; i >= 0 ; i-- ) {
  // if data is larger, shift existing item to
     right
    if (data > intArray [i]) {
       intArray [i+1 ] = intArray [i];
    } else {
       break ; }
    }
```

No :

Date: / /

Remove Value.

- Whenever an element is to be removed from queue, queue get the element using item count.

- Once element is removed. Item count is reduc by one.

Implementation.

```
static int removeData() { //remove from the end
  return intArray[--itemCount]; // highest Priority value
  }                             is at the end
static void print() {
  for (int i=0 ; i< itemCount ; i++) {
      System.out.print(i + "\t");
  }
  System.out.println();
  System.out.println("_ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _");
  for (int i=0 ; i< itemCount ; i++) {
      System.out.print(intArray[i] + "\t");
  }
  System.out.println();
  System.out.println();
}
```

Insert and Remove data in Priority Queue

Highest Priority is in the front of the array.

1. Ascending order priority Queue.

```
static void insert (int data) {
    int i = 0;
    if (! isFull() ) {
        // if queue is empty, insert the data
        if (itemCount == 0) {
            // intArray [itemCount++] = data;
            intArray [0] = data;
            itemCount++;
        }
        else {
            for (i = itemCount - 1; i >= 0 ; i--) {
                if (data < intArray [i]) {
                    intArray [i+1] = intArray [i]; // shift the
                }                                  currant element to the next index
                else {
                    break;
                }
            }
            // insert the data current location
            intArray [i+1] = data;
            itemCount ++;
        }
    }
}
```

```
Static int removeData() { //remove from the front
  if (isEmpty()){
    System.out.println("Queue is Empty");
  }
  int lowestPriorityElement = intArray[0]; // removing the first index 0 which is having the highest priority
  for (int i=1; i<itemCount ; i++){
    intArray[i-1] = intArray[i]; // other elements will be shift to the before index after removing Operation
  }
  itemcount--;
  return lowestPriorityElement; // Remove the lowest value / small value which is having the highest priority
}
  // lowestPriorityElement means lowest value which is the heighest priority element
```

2. Descending order Priority Queue.

```
Static void insert (int data) {
int i=0;
  if (isFull()){
    //if queue is empty, insert the data.
    if (itemCount == 0 ){

  //intArray [itemCount++] = data;
    intArray[0] = data;
    itemCount++;
  }
  else {
    for(i= itemCount-1 ; i>=0 ; i--){
      if (data > intArray[i]){
        intArray[i+1] = intArray[i];
      }
      else {
      } break;
```

```
            //insert the data
            intArray [i+1] = data;
            itemCount ++;
        }
    }
}

static int removeData () { //remove from the front
    if (isEmpty ()) {
        System.out.println ("Queue is Empty ");
    }
    int HighestPriorityElement = intArray [0];
    for (int i=1; i< itemCount ; i++ ) {
        intArray [i-1] = intArray [i];
    }
itemCount --;
return HighestPriorityElement;
}
```

Highest Priority is in the end of the Array

1. Ascending order priority Queue.

```
Static void insert (int data) {
  int i=0;
  if (! isFull()){
     // if queue is empty, insert the data
     if (itemCount == 0){
          //intArray [itemCount ++ ] = data;
          intArray [0]= data;
          itemCount ++;
     }
     else {
          for (i = itemCount-1 ; i>=0 ; i-- ){
             if ( data > intArray [i]){
                intArray [i+1] = intArray [i];
             }
             else {
                break;
             }
          }
          //insert the data
             Array [i+1] = data;
          itemCount ++;
     }
  }
}
```

```
static int removeData () {
    if (isEmpty()){
        System.out.println("Queue is Empty");
    }
    int lowestPriorityElement = intArray[itemCount-1];
    itemCount--;
    return lowestPriorityElement; // remove the
} highest value / big value which is having the
    highest priority
    // HighestPriorityElement means highest value
    which is the highest priority element
    // removing the last index of the array
    which is having the highest priority
```

2. Descending order Priority Queue.

```
static void insert (int data){
int i=0;
    if (! isFull()){
        // if queue is empty, insert the data
        if (itemCount==0){
            //intArray [itemCount++] = data;
            intArray [0] = data;
            itemCount++;
        }
        else {
            for (i= itemCount-1; i>=0; i--){
                if (data < intArray [i]){
                    intArray [i+1] = intArray [i];
                }
                else {
                    break;
                }
            }
```

```
            // insert the Data
            intArray [i+1] = data;
            itemCount ++;
        }
    }
}

static int removeData() {
    if (isEmpty()) {
        System.out.println("queue is Empty");
    }
    int HighestPriorityElement = intArray[itemCount-1];
    itemCount--;
    return HighestPriorityElement;
}
```

Insert the Data.

Case 1: Highest priority in the front

Ascending order data < Current element in the Array
Descending order data > current element in the Array

Case 2: Highest priority in the end

Ascending order data > current element in the Array
Descending order data < current element in the Array

Remove the Data

Case 1 : Highest priority in the front

```
lowestElement = intArray [0];
  //Shifting the next element before index
for (int i=1 ; i<itemCount ; i++) {
    intArray [i-1] = intArray [i];
}
```

Case 2 : Highest priority in the end

```
HighestElement = intArray [itemCount-1];
```