DATA STRUCTURES

Queues

Lecture No. 9,10

Queues

- A **Queue** is a special kind of list, where items are inserted at one end (**the rear**) And deleted at the other end (**the front**).
- Accessing the elements of queues follows a First In First Out (FIFO) order.
- Example
  - Like customers standing in a check-out line in a store, the first customer in is the first customer served.

Common Operations on Queues

- **FRONT(*Q*):** Returns the first element on Queue Q.
- **ENQUEUE(*x*,*Q*):** Inserts element x at the end of Queue Q.
- **DEQUEUE(*Q*):** Deletes the first element of *Q.*
- **ISEMPTY(*Q*):** Returns true if and only if Q is an empty queue.
- **ISFULL(Q):** Returns true if and only if Q is full.

Enqueue and Dequeue

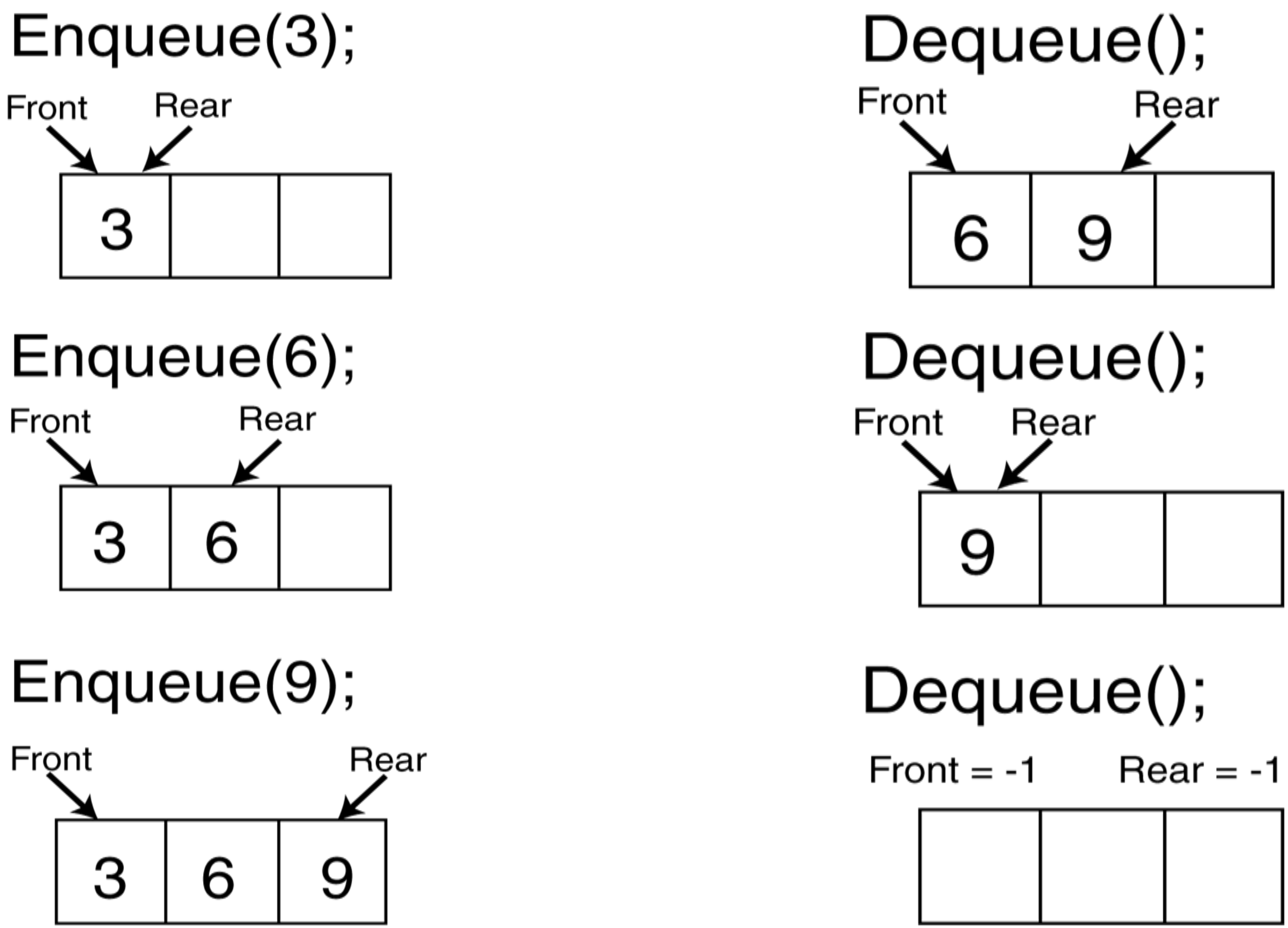
- Primary queue operations: Enqueue and Dequeue

- **Enqueue** – insert an element at the rear of the queue.
- **Dequeue** – remove an element from the front of the queue.

Queues Implementations

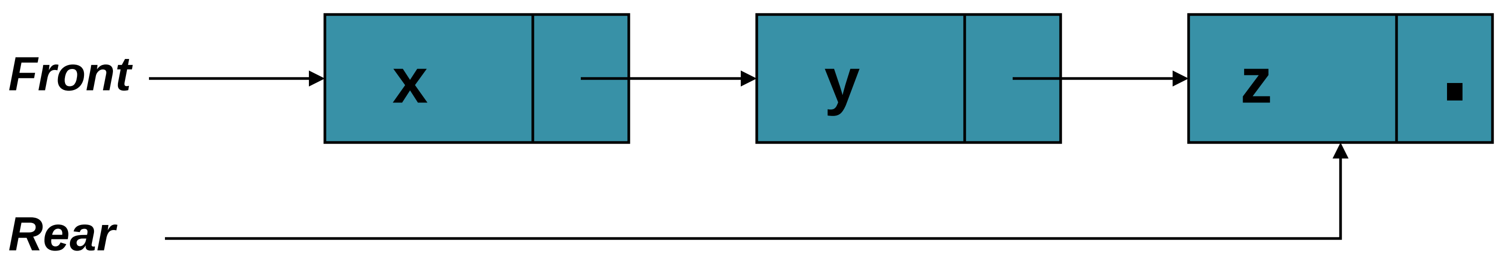
- Static
  - Queue is implemented by an array, and size of queue remains fix
- Dynamic
  - A **queue** can be **implemented** as a **linked list**, and *expand* or *shrink* with each *enqueue* or *dequeue* operation.

Static Implementation of Queues



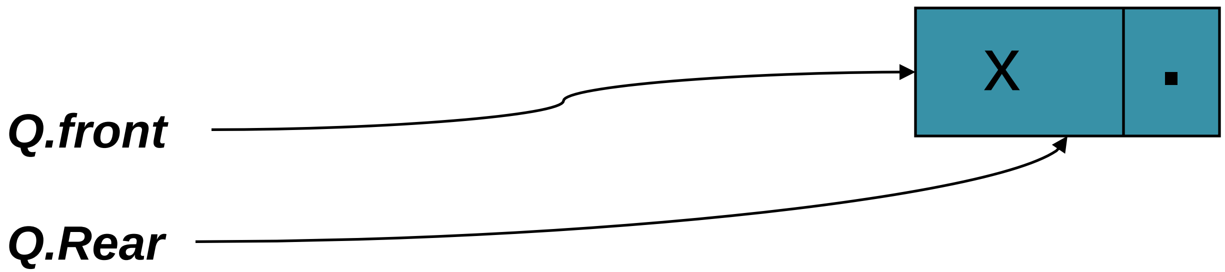
Dynamic Implementation of Queues

- Dynamic implementation is done using pointers.
  - FRONT: A pointer to the first element of the queue.
  - REAR: A pointer to the last element of the queue.

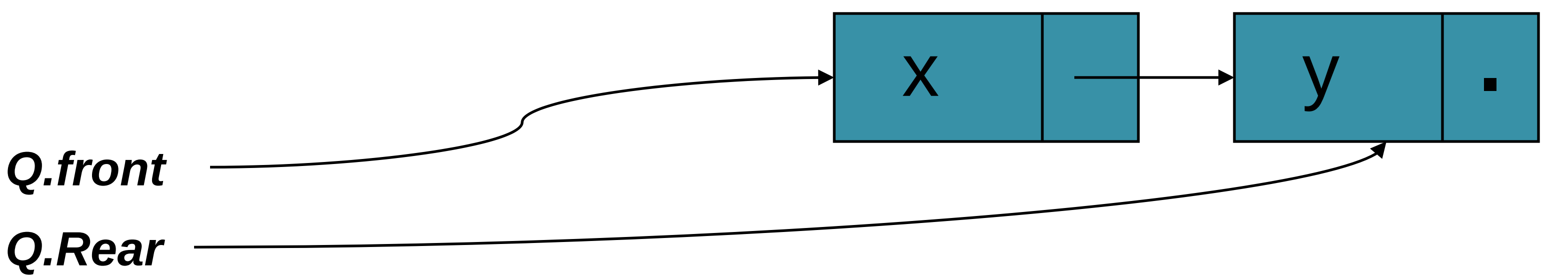


Dynamic Implementation

- Enqueue (X)

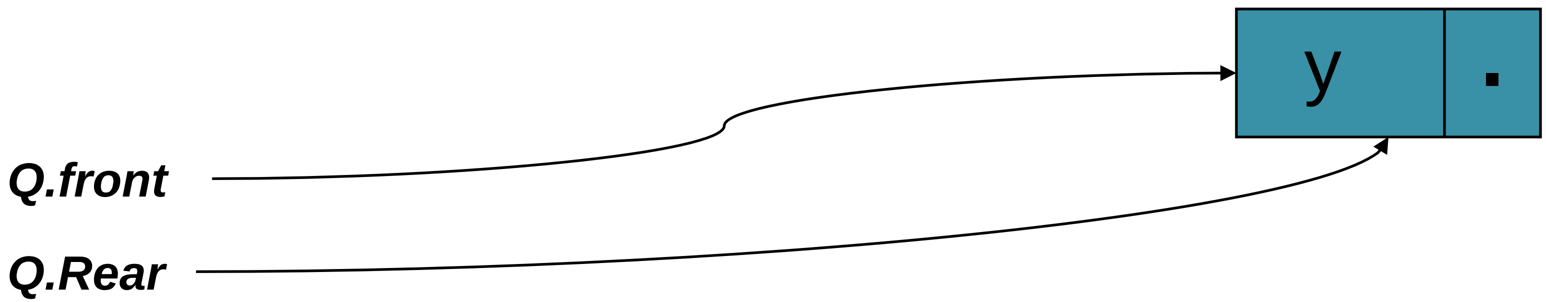


- Enqueue (Y)

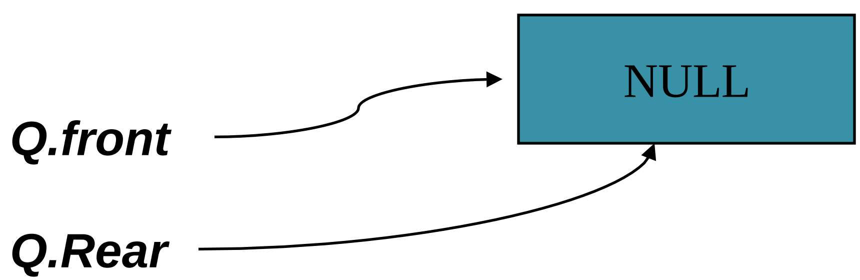


Dynamic Implementation

- Dequeue



- MakeNULL



Dynamic implementation of Queue

```
class DynQueue{
   private:
   struct queueNode
     {
          int num;
           queueNode *next;
     };
     queueNode *front;
     queueNode *rear;
public:
     DynQueue();
     ~DynQueue();
      void enqueue();
      void dequeue();
      bool isEmpty();
      void displayQueue();
      void makeNull();
};
```

Constructor

```
DynQueue::DynQueue()
{
   front = NULL;
   rear = NULL;
}
```

Enqueue( ) Function

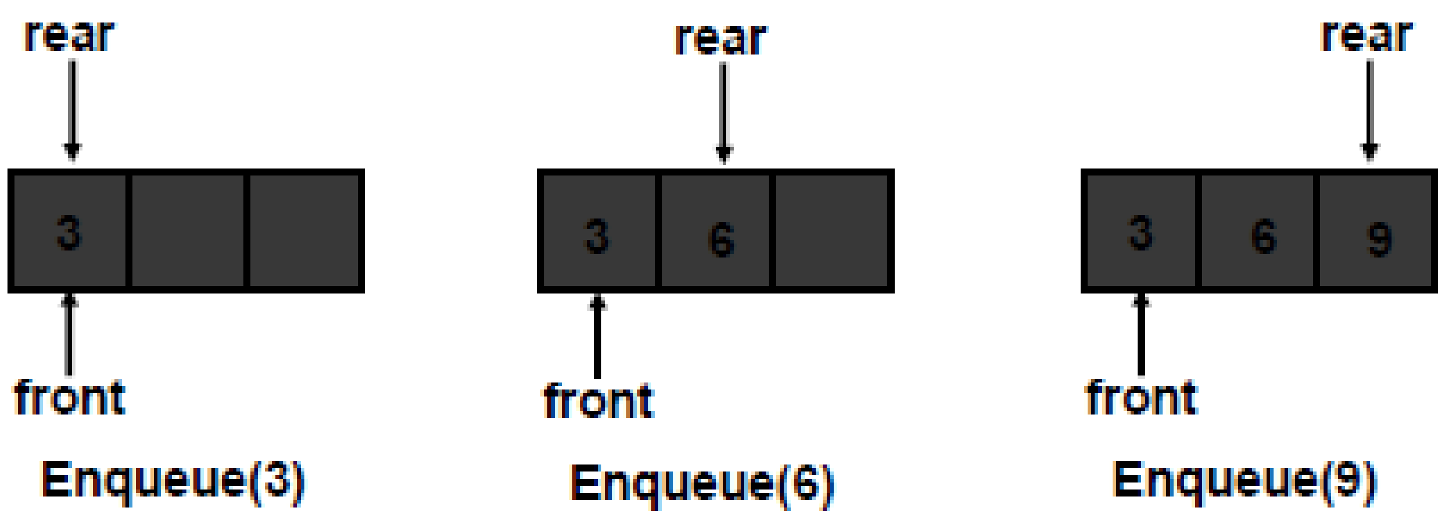
```
void DynQueue::enqueue()
{
    queueNode *ptr = new queueNode;
    cout<<"Enter Data";
    cin>>ptr->num;
    ptr->next= NULL;
    if (front == NULL)
    {       front = ptr;
            rear = front;       }
    else{
            rear->next=ptr;
            rear = ptr;         }
}
```

Dequeue( ) Function

```
void DynQueue::dequeue()
{
    queueNode *temp;
    temp = front;
    if(isEmpty())
            cout<<"Queue is Empty";
   else
   {
            cout<<"data deleted="<<temp->num;
            front = front->next;
            delete temp;
   }
}
```

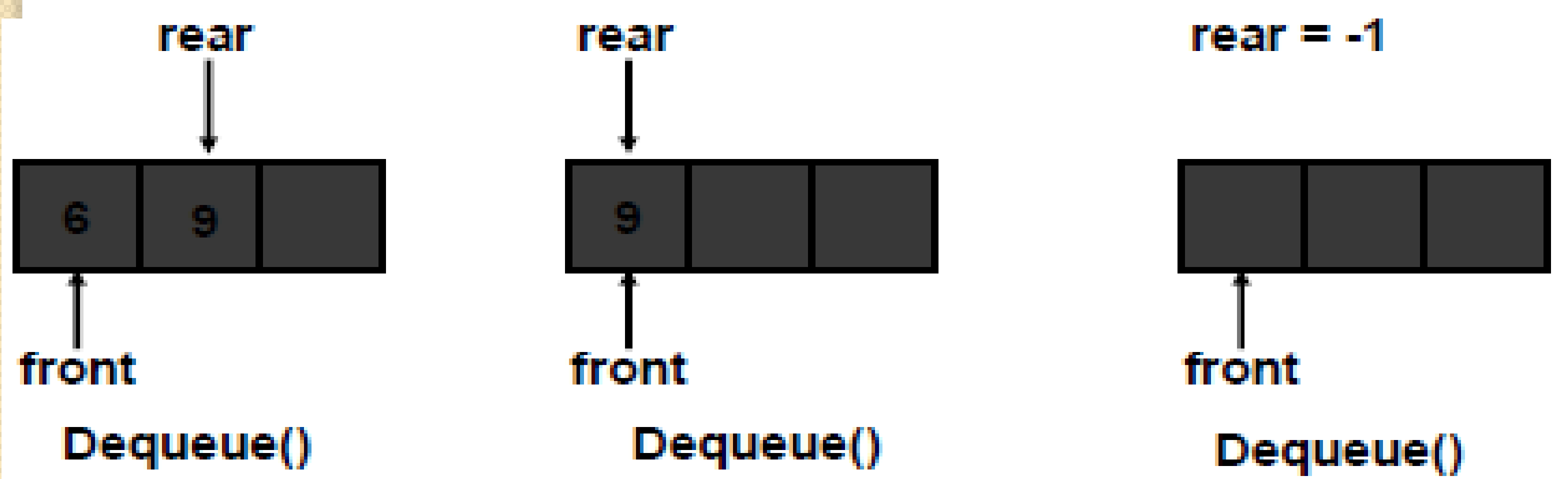
Static Implementation of Queue

- Static implementation is done using arrays
- In this implementation, we should know the exact number of elements to be stored in the queue.
- When enqueuing, the front index is always fixed and the rear index moves forward in the array.



Static Implementation of Queue

- When dequeuing, the front index is fixed, and the element at the front of the queue is removed. Move all the elements after it by one position. (Inefficient!!!)



Static Implementation of Queue

- **A better way**
  - When an item is enqueued, the rear index moves forward.
  - When an item is dequeued, the front index also moves forward by one element
- **Example:**

X = occupied, and O = empty

- (front) XXXXOOOOOO (rear)
- OXXXXOOOO (after 1 dequeue, and 1 enqueue)
- OOXXXXXOO (after another dequeue, and 2 enqueues)
- OOOOXXXXX (after 2 more dequeues, and 2 enqueues)

- **The problem here is that the rear index cannot move beyond the last element in the array.**

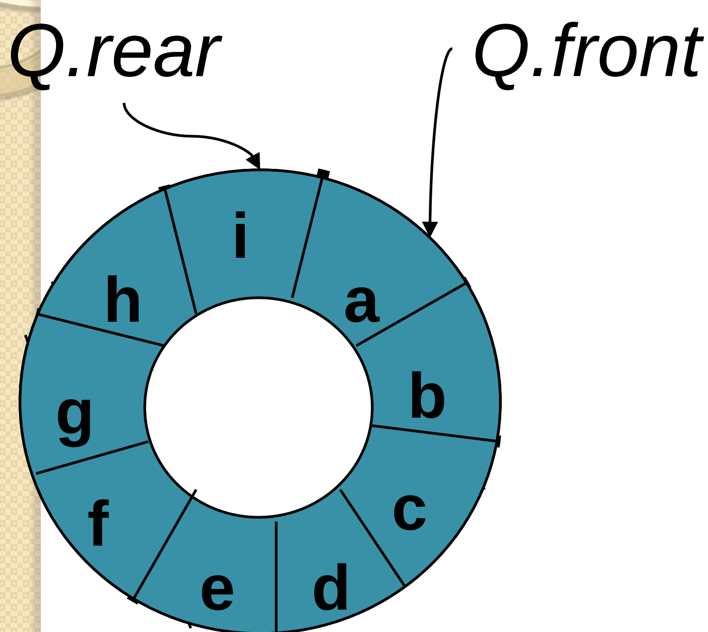
Static Implementation of Queue

- To overcome the above limitation, we can use **circular array implementation of queues.**
- In this implementation, first position follows the last.

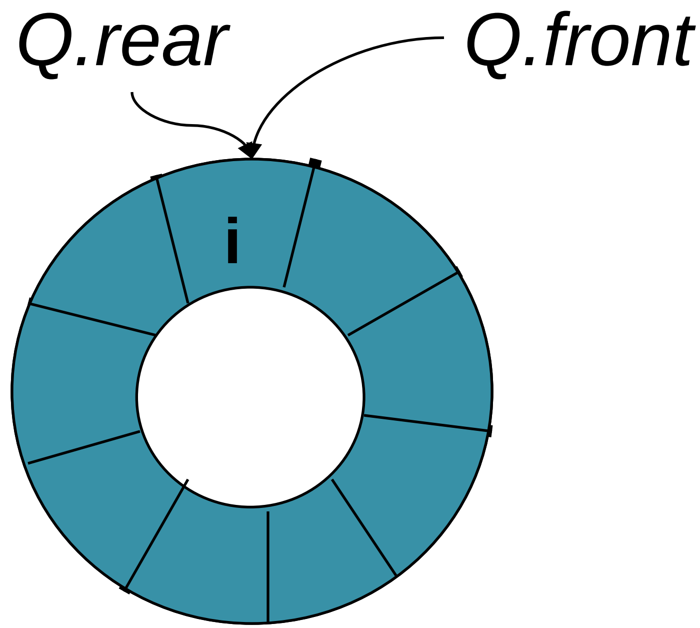
- **When an element moves past the end of a circular array, it wraps around to the beginning, e.g**
  - OOOOO7963 ->4OOOO7963 (after Enqueue(4))
  - After Enqueue(4), the rear index moves from 3 to 4.

- **How to detect an empty or full queue, using a circular array algorithm?**
  - Use a counter of the number of elements in the queue.

Circular Queue



A Completely
Filled Queue



A Queue with
Only 1 Element

Circular Queue Implementation

```
class CirQueue
{
   private:
         int queue[5];
         int rear;
         int front;
         int maxSize;
         int counter;

public:
         CirQueue();
         void enqueue();
         void dequeue();
         bool isEmpty();
         bool isFull();
         void display();
};
```

Constructor

```
CirQueue::CirQueu
   e()
{
   front = 0;
    rear = -1;
    maxSize = 5;
    counter =0;
}
```

Enqueue( ) Function

```
void CirQueue::enqueue()
{
    if ( isFull())
    cout<<"queue is full";
   else
   {
      rear = (rear + 1) % maxSize;
      cout<<"Enter Data=";
      cin>> queue[rear];
      counter ++;
   }
}
```

Dequeue( ) Function

```
void CirQueue::dequeue()
{
    if ( isEmpty())
    cout<<"Queue is empty";
   else
   {
    cout<< "Element deleted="<<queue[front];
      front = (front +1)% maxSize;
      counter --;
   }
}
```

Display( ) Function

```
void CirQueue::display()
{
    if(isEmpty())
    cout<<"Queue is empty";
   else
   {
           for (int i=0; i<counter; i++)
           cout<< queue[(front+ i)% maxSize]<<endl;
   }

}
```

isEmpty( ) and isFull( )

```
bool CirQueue::isEmpty()
{
    if (counter == 0)
    return true;
   else
    return false;
}
bool CirQueue::isFull()
{
    if (counter < maxSize)
    return false;
   else
    return true;
}
```

PRIORITY QUEUES

A priority queue is a data structure in which prioritized insertion and deletion operations on elements can be performed according to their priority values.

There are two types of priority queues:

- Ascending Priority queue
- Descending Priority queue

Introduction

- Stack and Queue are data structures whose elements are ordered based on a sequence in which they have been inserted.
- E.g. pop() function removes the item pushed last in the stack.
- Intrinsic order among the elements themselves (e.g. numeric or alphabetic order etc.) is ignored in a stack or a queue.

Types of Priority Queue

- **Ascending Priority queue**: a collection of items into which items can be inserted ***randomly*** but only the ***smallest*** item can be removed.
- If "**A-Priority-Q**" is an ascending priority queue then
  - Enqueue() will insert item 'x' into **A-Priority-Q**,
  - minDequeue() will remove the minimum item from **A-Priority-Q** and return its value.

Types of Priority Queue

- **Descending Priority queue**: a collection of items into which items can be inserted ***randomly*** but only the ***largest*** item can be removed
- If "**D-Priority-Q**" is a descending priority queue then
  - Enqueue() will insert item x into **D-Priority-Q**,
  - maxDequeue( ) will remove the maximum item from **D-Priority-Q** and return its value

Generally

- In both the above types, if elements with equal priority are present, the FIFO technique is applied.

- Both types of priority queues are similar in a way that both of them remove and return the element with the highest **"Priority"** when the function remove() is called.
  - For an ascending priority queue item with smallest value has maximum "priority"
  - For a descending priority queue item with highest value has maximum "priority"

- This implies that we must have criteria for a priority queue to determine the Priority of its constituent elements.
- the elements of a priority queue can be numbers, characters or any complex structures such as phone book entries, events in a simulation.

Priority Queue Issues

- In what manner should the items be inserted in a priority queue
  - Ordered (so that retrieval is simple, but insertion will become complex)
  - Arbitrary (insertion is simple but retrieval will require elaborate search mechanism)
- Retrieval
  - In case of un-ordered priority queue, what if minimum number is to be removed from an ascending queue of n elements (n number of comparisons)
- In what manner should the queue be maintained when an item is removed from it
  - Emptied location is kept blank (how to recognize a blank location ??)
  - Remaining items are shifted