Introduction to Queue ADT

Dr. Tahir Maqsood

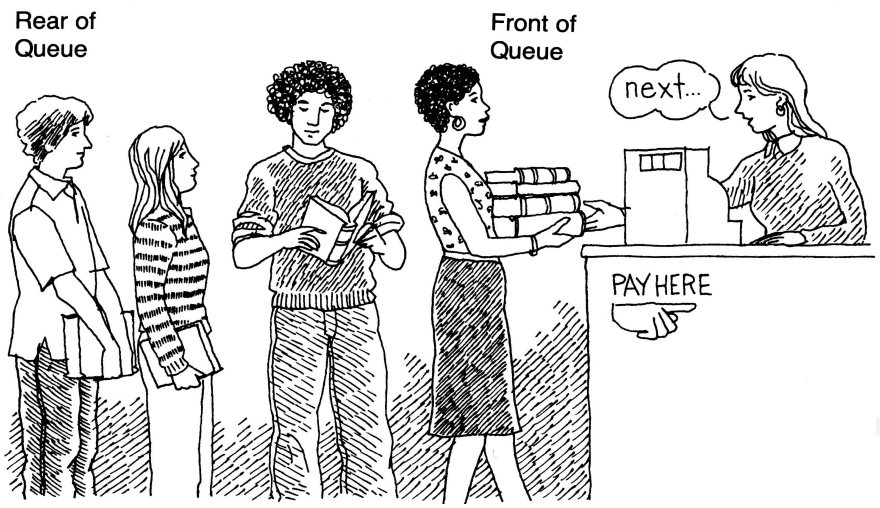
Department of Computer Science
COMSATS University Islamabad
Lahore Campus

Outline

- Queue Abstract Data Type (ADT)
- Queue ADT with Array Implementation
- Operations
  - **Insert** a new element in the list
    - insert at the **beginning** of the list
    - insert at the **end** of the list
    - insert **anywhere** in the list
  - **Delete** an element from the list
    - delete from the **beginning** of the list
    - delete from the **end** of the list
    - delete **any** element in the list
  - **Display** print the elements of list

What is Queue?

- It is an ordered group of homogeneous items of elements.
- Queues have two ends:
  - Elements are added at one end.
  - Elements are removed from the other end.
- The element added first is also removed first (**FIFO**: First In, First Out).

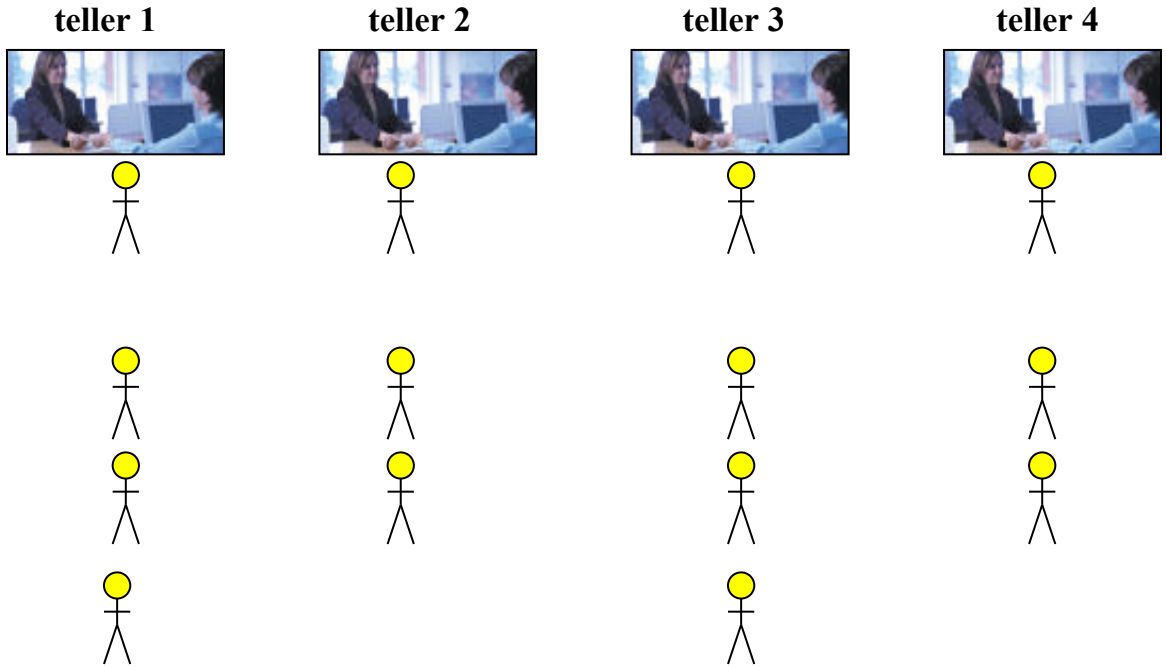


- **Upshot**
- A queue is a linear data structure into which items can only be inserted at rear and removed from front.
- **Difference between Stack and Queue**

Stack	Queue
Stack is a **LIFO** (Last In First Out) structure	queue is a **FIFO** (First In First Out) structure
Stack has one end(Top)	Queues have two ends(Front and Rear)

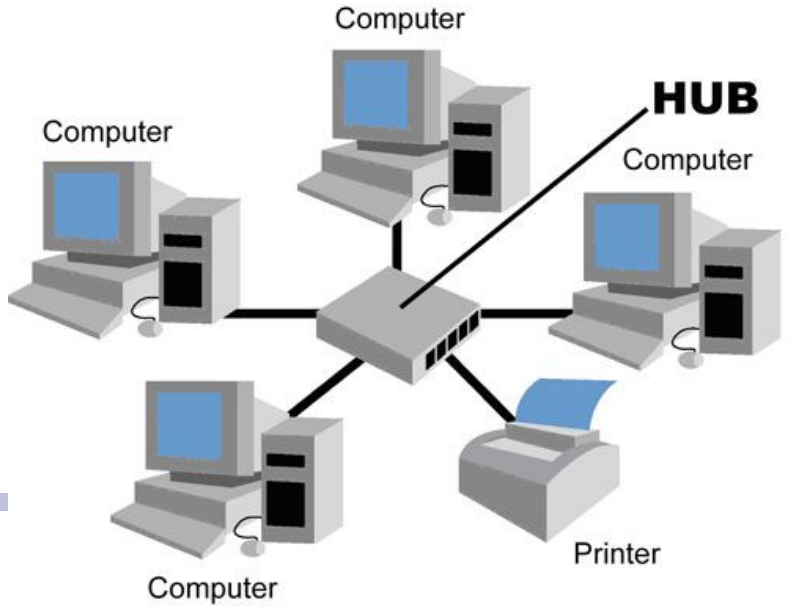
Examples

For example, we queue up while depositing a utility bill or purchasing a ticket. The objective of that queue is to serve persons in their arrival order; the first coming person is served first. The person, who comes first, stands at the start followed by the person coming after him and so on. At the serving side, the person who has joined the queue first is served first.



Example

- Example 2: Suppose there are four computers and one printer is connected. Computer B send print at time 9:30 am which takes 15 minutes for printing, Computer D send print command at time 9:35 which takes 7 minutes, computer A send print command at 9: 36 which takes 2 minutes and computer C send print command at 9:37 which takes 30 seconds. How printers organized these jobs in its buffer**?**
- **In which order job will prints?**



Queue Specification

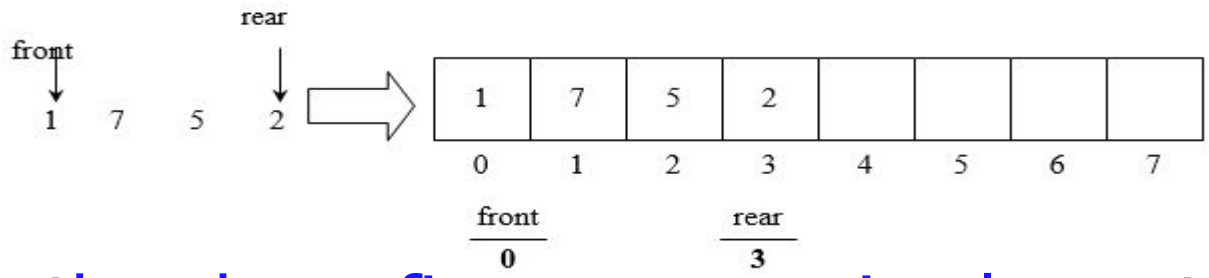
- Definitions: (provided by the user)
  - *MAX_ITEMS*: Max number of items that might be on the queue
  - *ItemType*: Data type of the items on the queue

- Operations
  - MakeEmpty
  - Boolean IsEmpty
  - Boolean IsFull
  - Enqueue (ItemType newItem)
  - Dequeue (ItemType& item)

Enqueue

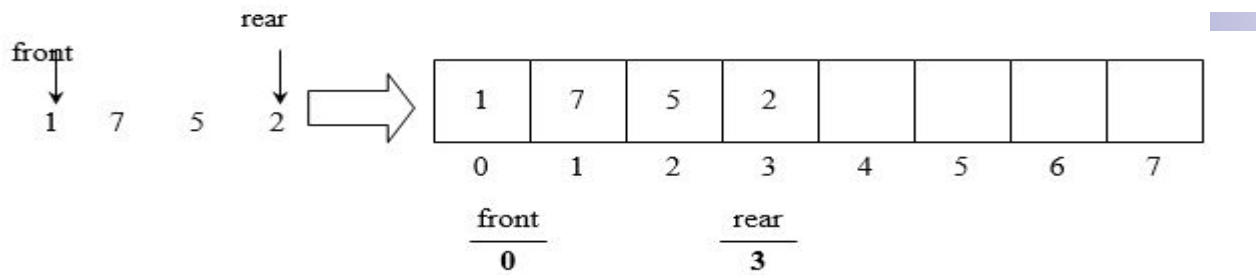
- Signature: Enqueue (ItemType newItem)
- Function: Adds newItem to the rear of the queue.
- Preconditions: Queue has been initialized and is not full.
- Postconditions: newItem is at rear of queue.

Working of enqueue()



- In the above figure, queue implementation using array is shown.
- As the array size is 8, therefore, the index of the array will be from 0 to 7.
- The number of elements inside array are *1*, *7*, *5* and *2*, placed at start of the array.
- The *front* and *rear* in this implementation are not pointers but just indexes of arrays.
- *front* contains the starting index i.e. *0* while *rear* comprises *3*.

Working of enqueue()



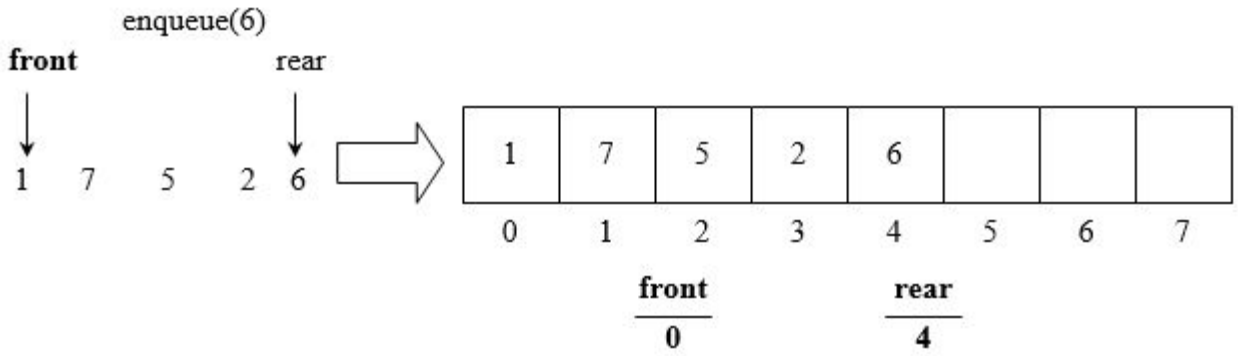


Fig 2. Insertion of one element 6

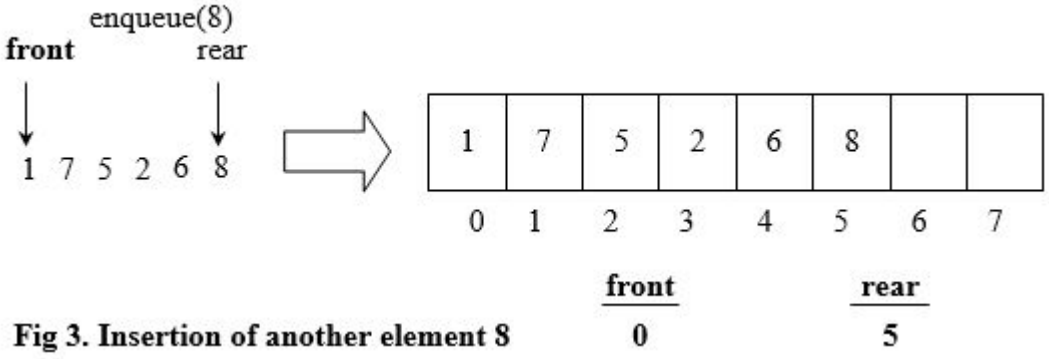


Fig 3. Insertion of another element 8

Working of enqueue()

When an element is removed from the queue. It is removed from the *front* index.

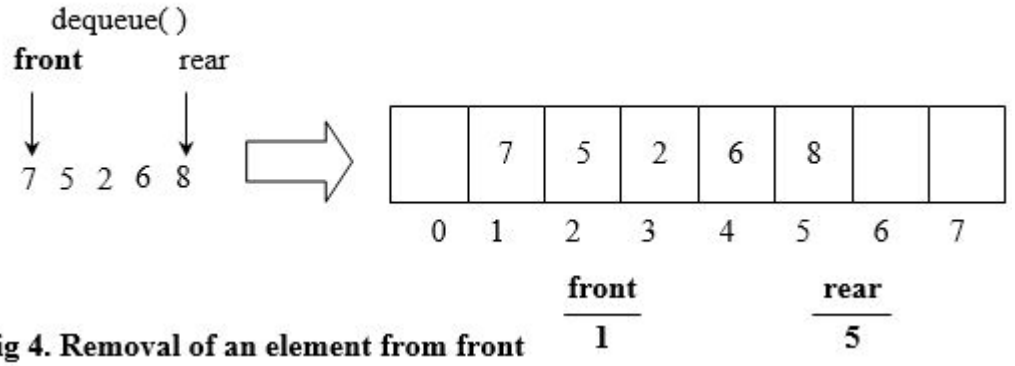


Fig 4. Removal of an element from front

After another call of *dequeue()* function:

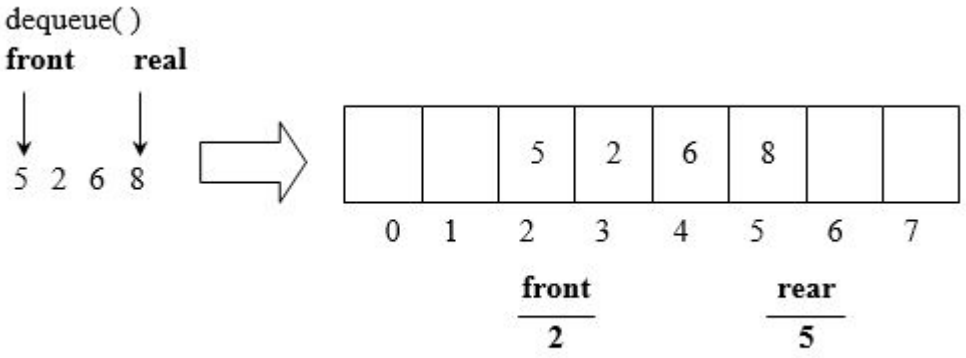


Fig 5. Removal of another element from front

Working of enqueue()

- With the removal of element from the queue, we are not shifting the array elements. The shifting of elements might be an expensive exercise to perform and the cost is increased with the increase in number of elements in the array. Therefore, we will leave them as it is

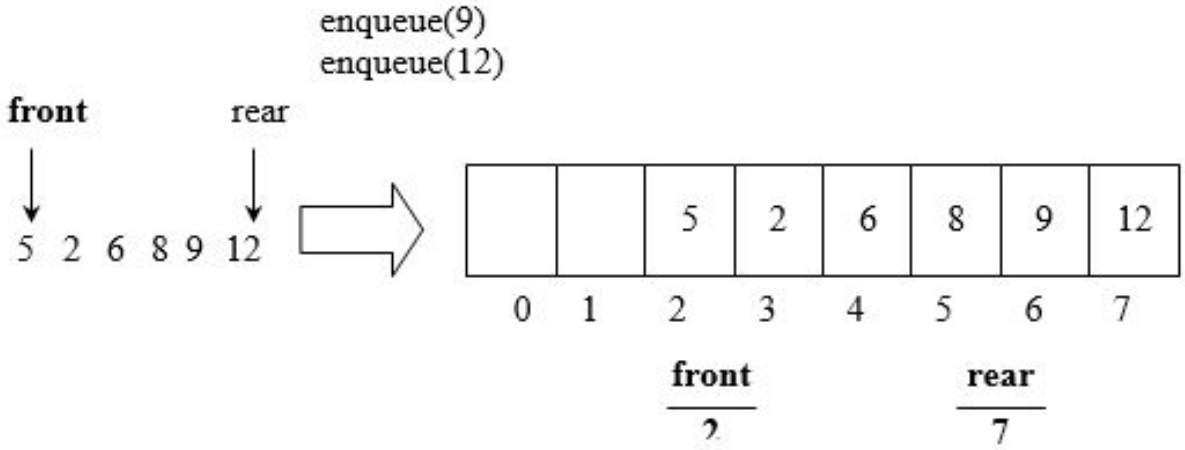


Fig 6. Insertion of elements in the queue

Insertion Operation

- After insertion of two elements in the queue, the array that was used to implement it, has reached its limit as the last location of the array is in use now.
- isFull() returns true if the number of elements (*noElements*) in the array is equal to the *size* of the array. Otherwise, it returns false.
- It is the responsibility of the caller of the queue structure to call isFull() function to confirm that there is some space left in the queue to *enqueue()* more elements.

```
int  isFull()
{
    return  noElements  ==  size;
}
```

enqueue()

```
void enqueue()
{
    int val;
    if (rear == n - 1)
        cout<<"Queue Overflow"<<endl;
    else
    {
        if (front == - 1)
        front = 0;
        cout<<"Insert the element in queue : "<<endl;
        cin>>val;
        rear++;
        queue[rear] = val;
    }
}
```

```
void enqueue(int item)
{
    if(rear==size-1)
    {//as you cannot insert in the front
        cout<<"Queue is Full";
    }
    else if(rear==-1)
    {
        rear=front=0;
        queue[rear]=item;
    }
    else
    {
        rear++;
        queue[rear]=item;
    }
}
```

dequeue( )

```
void Dequeue()
{
    if (front == - 1 || front > rear) {
        cout<<"Queue Underflow ";
    return ;
    } else {
        cout<<"Element deleted from queue is : "<< queue[front] <<endl;
        front++;;
    }
}
```

```
void dequeue()
{
    if(rear==-1)
    {
        cout<<"Queue is Empty";
    }
    else if(front==rear)
    {
        rear=front=-1;
    }
    else
    {
        front++;
    }
}
```

display( )

```
void Display()
{
    if (front == - 1)
    cout<<"Queue is empty"<<endl;
    else
    {
        cout<<"Queue elements are : ";
        for (int i = front; i <= rear; i++)
            cout<<queue[i]<<" ";
        cout<<endl;
    }
}
```

enqueue( )

```
void enqueue(int val)
{
    if (rear == NULL)
    {
        rear = new node;
        rear->next = NULL;
        rear->data = val;
        front = rear;
    }
    else
    {
        temp= new node;
        rear->next = temp;
        temp->data = val;
        temp->next = NULL;
        rear = temp;
    }
}
```

```
void enqueue(int item)
{
    node *temp=new node;
    temp->item=item;
    if(front==NULL)
    {
        front=rear=temp;
        temp->next=NULL;
    }
    else
    {
        rear->next=temp;
        rear=temp;
        temp->next=NULL;
    }
}
```

dequeue( )

```
void dequeue()
{
    temp = front;
    if (front == NULL)
    {
        cout<<"Underflow"<<endl;
        return;
    }
    else if (temp->next != NULL)
    {
        temp = temp->next;
        cout<<"Element deleted from queue is : "<<front->data<<endl;
        delete(front);
        front = temp;
    }
    else
    {
        cout<<"Element deleted from queue is : "<<front->data<<endl;
        delete(front);
        front = NULL;
        rear = NULL;
    }
}
```

```
void dequeue()
{
    if(front==NULL)
    {
        cout<<"Queue is Empty";
    }
    else if(front==rear)
    {
        front=rear=NULL;
    }
    else
    {
        front=front->next;
    }
}
```

display( )

```
void Display()
{
    temp = front;
    if ((front == NULL) && (rear == NULL))
    {
        cout<<"Queue is empty"<<endl;
        return;
    }
    cout<<"Queue elements are: ";
    while (temp != NULL)
    {
        cout<<temp->data<<" ";
        temp = temp->next;
    }
    cout<<endl;
}
```

```
void display()
{
    if(rear==NULL)
    {
        cout<<"Queue is Empty";
    }
    else
    {
        node *temp=front;
        while(temp!=NULL)
        {
            cout<<temp->item<<" ";
            temp=temp->next;
        }
    }
}
```

Array Implementation of Queues

- To Enqueue an element X, we increment Size and Rear, then set Queue[Rear]=X.
- To Dequeue an element, we set the return value to Queue[Front], decrement Size, and then increment Front.
- Whenever Front or Rear gets to the end of the array, it is wrapped around to the beginning. This is known as a circular array implementation.

Circular Queues

- The simple array implementation of queue has a problem
  - It is wasting space
  - Because front is moving towards back and leaving empty space behind.
  - Elements are stored towards end of array and front cells may be empty due to dequeue

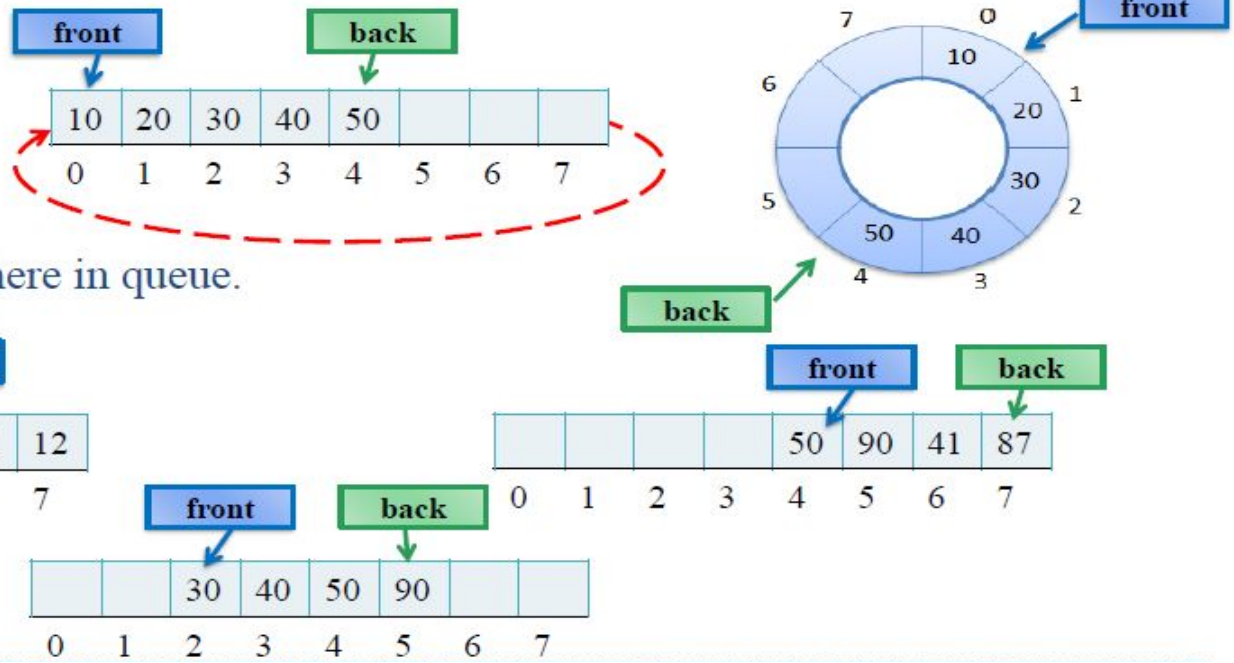
To resolve this problem array can be used as a circular array.

- When we reached at end of array, start from beginning

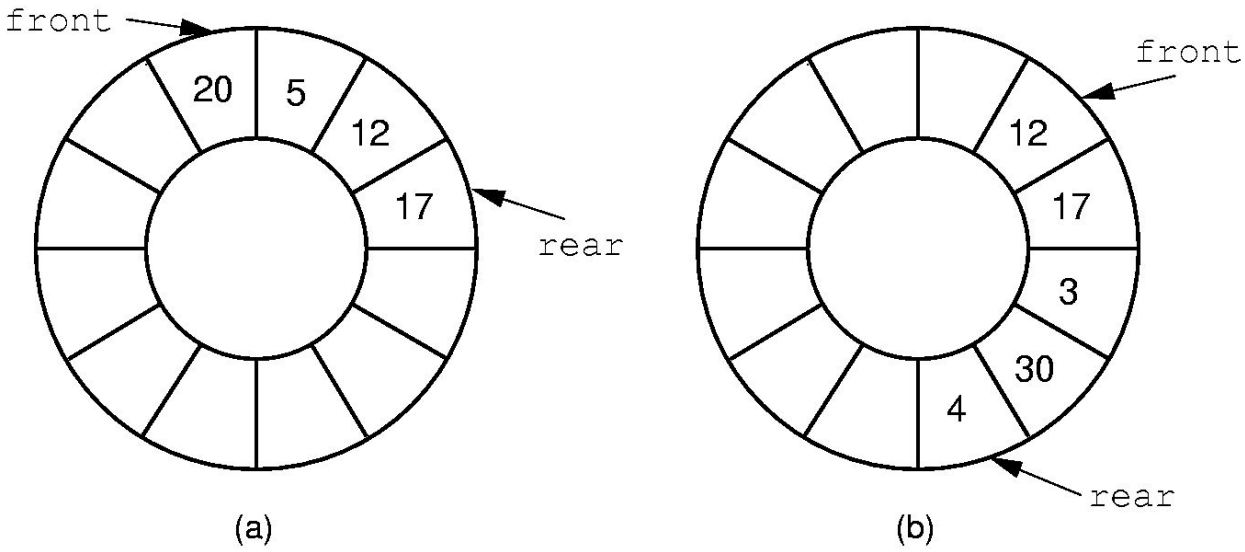
- Whenever Front or Rear gets to the end of the array, it is wrapped around to the beginning. This is known as a circular array implementation.

Circular Queue

Circular queue is a simple queue, But logically it says that queue[0] comes after queue[N-1]

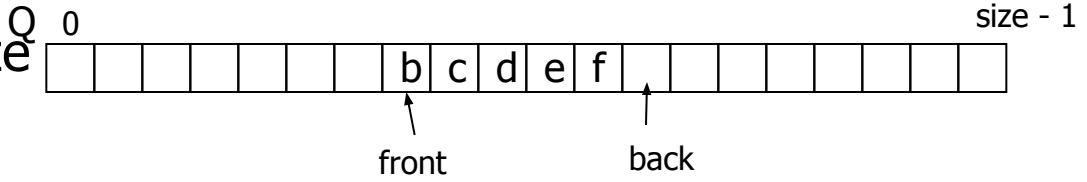


Queue Implementation (2)



Circular Array Q Data Structure

- enqueue(Object x) {
  - Q[back] = x ;
  - back = (back + 1) % size
  }



How test for empty list?

How to find K-th element in the queue?

```
dequeue() {
    x = Q[front] ;
    front = (front + 1) % size;
    return x ; }
```

Algorithms

Algorithm:CIRCULAR_ENQUEUE(Queue, E)

Input: a queue, an element E

Output: updated queue with E inserted

Steps:

```
1.  If(Queue is Full)
2.    Print "Queue overflow"
3.  Else
4.    Back=(Back+1)%N
5.    Queue[Back]=E
6.    IF(Front==-1)
7.              Front=0
8.    End If
9.  End If
```

Algorithm: CIRCULAR_DEQUEUE(Queue)

Input: a queue

Output: updated queue, with front element removed

Steps:

```
1.  Let E = null
2.  IF(Queue is Empty)
3.    Print "Queue underflow"
4.  Else
5.    E=Queue[Front]
6.    Front=(Front+1)%N
7.      IF(Front==Back)
8.          Front=Back=-1
9.      End If
10. End If
11. return E
```

Helper Functions

Helper Methods	Queue	Circular Queue
isEmpty()	front, back = -1	same
isFull()	back+1= N	(back+1)%N= front
Size()	back-front + 1	(N-front+back)%N + 1 If front<back back-front + 1 Else (N-front) +(back+1)

Example Applications

- When jobs are submitted to a printer, they are arranged in order of arrival.
- Every real-life line is a queue. For instance, lines at ticket counters are queues, because service is first-come first-served.
- A whole branch of mathematics, known as *queueing theory*, deals with computing, probabilistically, how long users expect to wait on a line, how long the line gets, and other such questions.

Priority Queue

- Imagine a ticket window at Daewoo Terminal, the person who arrives first will get the ticket before the person who arrives later.
- Now imagine process queue for CPU, ideally it should execute the process whose request was arrived first, but some time situation arise when a component/process raise request that is of highest priority and needs to be executed first regardless of the order of arriving in process queue, like computer shut down
- That is the case where we need a queue which can handle priority.

Priority Queue

- A priority queue is a collection of zero or more items, associated with each item is a priority.
- In a normal queue the enqueue operation add an item at the back of the queue, and the dequeue operation removes an item at the front of the queue.
- In priority queue, enqueue and dequeue operations consider the priorities of items to insert and remove them.
- Priority queue does not follow "**first in first out**" order in general.
- The highest priority can be either the most minimum value or most maximum
- we will assume the highest priority is the minimum.

Priority Queue as ADT

Operations:

- **Enqueue(E**)
- **Dequeue**(): remove the item with the highest priorit
- **Find**() return the item with the highest priority

Examples

- Process scheduling
  - Few processes have more priority
- Job scheduling
  - N Jobs with limited resources to complete Patients treatment in emergency

Priority Queue as ADT

First Approach:

- enqueue operation add item at the back of the queue → O(1)
- dequeue operation removes item with highest priority → O(N)
  - Find highest priority element, remove it by shifting elements to left

Second Approach:

- enqueue operation insert items according to their priorities. → O(N)
  - A higher priority item is always enqueued before a lower priority element, so item with highest priority will always be at start.
- dequeue operation removes an item from front end → O(1)

Key Note:

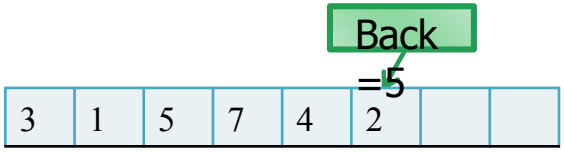
- If two or more elements have same priority, then they follow FIFO. Element that came first, will be removed first.

First Approach

Enqueue(3)Enqueue(1)

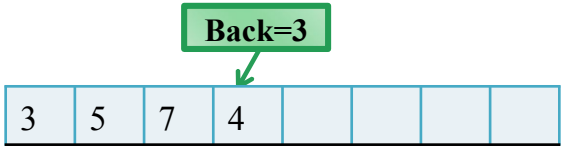
Enqueue(5) Enqueue(7)

Enqueue(4) Enqueue(2)



Dequeue()

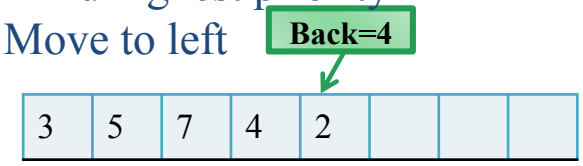
Find highest priority 2

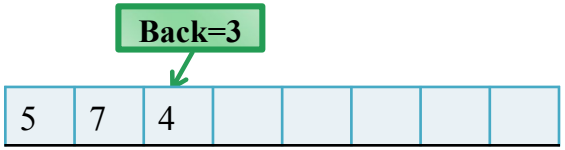


Dequeue()

Find highest priority 1

- Move to left





Algorithms

Algorithm: **ENQUEUE** (PQ, E)

Input: priority queue, an element E

Output: updated queue, with E inserted

Steps:

```
1.  If (PQ isFull)
2.    Print "Queue overflow"
3.  Else
4.      Back=Back+1
5.      PQ[Back]=E
6.  End If
```

Algorithm: **DEQUEUE** (PQ)

Input: priority queue,

Output: updated queue, with highest priority element removed

Steps:

```
1.  Let E = null
2.  If (PQ is Empty)
3.    Print "Queue underflow"
4.  Else
5.    index=FIND_INDEX_OF_MIN(PQ) // function to find index of min
6.  E=PQ[index]
7.  For i=index;i<back;i++ //shift back
8.      PQ[i]=PQ[i+1]
9.    End for
10.  back--
11. End if
12. return E
```

Algorithms

Algorithm: FIND_INDEX_OF_MIN()

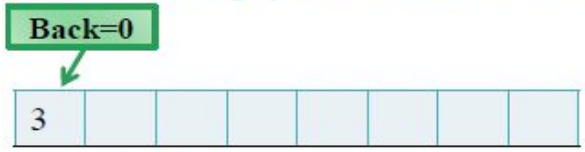
1. Let m=0
2. **For** i=1; i< =Back; i++
3. If PQ[i]'s priority <PQ[m]'s priority // it depends what is stored in array, priority as a number itself, or an object which contains data and priority as attributes
4. m=i
5. End if
6. Return m

	Priority Queue simulation
1	Use single array of integers which only store priority
2	Use one array for data, other for priority
3	Use an object's array, where object contains both its data and priority as well

Second Approach

Enqueue(3)

Queue is empty, so insert at front

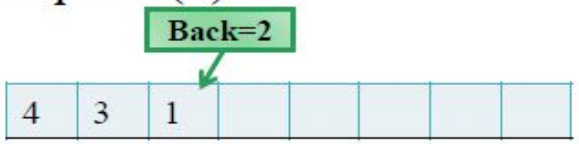


Enqueue(4)

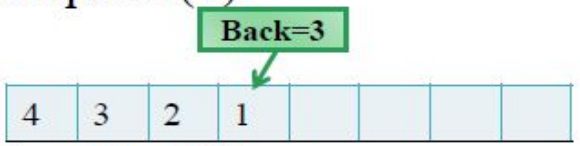
Queue is not Empty, So find appropriate location to insert according to priority

shift elements to right and insert, if needed

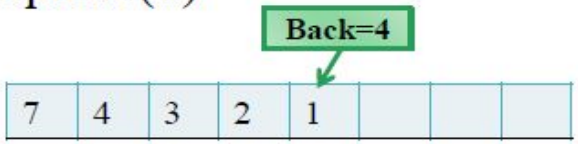
Enqueue(1)→



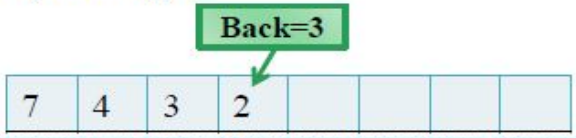
Enqueue(2)



Enqueue(7)



Dequeue()



Algorithm

Algorithm: **ENQUEUE** (PQ, E)

Input: priority queue, an element E

Output: updated queue, with E inserted

Steps:

```
1.  If(PQ is Full)
2.      Print "Queue overflow"
3.  Else
4.    If(Back==-1)
5.       PQ[++Back]=E
6.    Else
7.       Back=Back+1
8.       SORTED_INSERT()
9.
10.   End If
11. End If
```

SORTED_INSERT()

```
1.  Set i=0
2.  //find correct location
3.  While (E's priority > PQ[i]'s priority and i <Back)
4.              i=i+1
5.   End While
6.  // shift to right
7.  For j = Back ; j > i; j--
8.     PQ[j]=PQ[j-1]
9.  End For
10. PQ[i]=E
```

Algorithm

Algorithm:DEQUEUE()

Input: priority queue,

Output: updated queue, with highest priority element removed

Steps:

1. Let E = null
2. If(PQ is Empty)
   Print "Queue underflow"
3. Else
4. E=PQ[Back]
5. Back= Back-1
6. End if
7. Return E