B. Tech. I CSE (Sem-2)
Data Structures
CS102

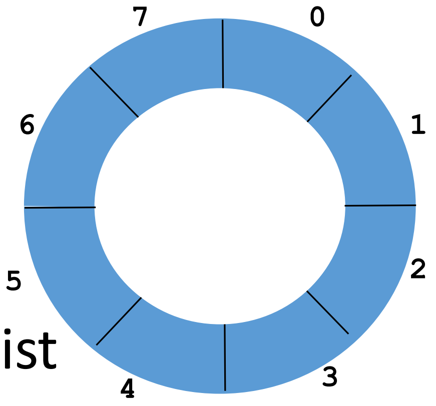
Queue Variants

Circular Queue (CQueue),
Doubly Ended Queue (DeQueue),
Priority Queue

*Some slides are kept with logical or syntax error. So, if you are absent in theory class, please also refer the slides provided at the end of the presentation.

Dr. Rupa G. Mehta
Dr. Dipti P. Rana
Department of Computer Science and Engineering
SVNIT, Surat

CQueue

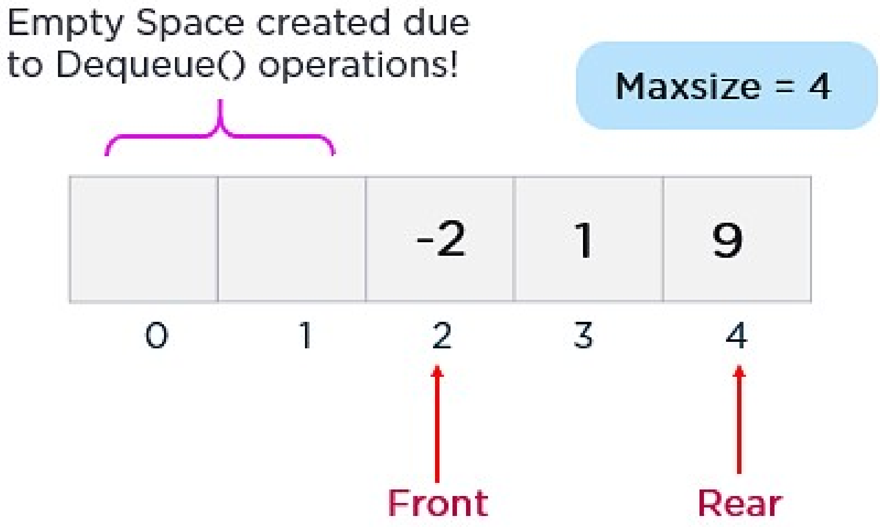


- Last element points to the first element making a circular list
- Rear end is connected to the front end forming a circular loop
- Advantage
  - Insertion and deletion operations are independent of one another
    - This prevents an interrupt handler from performing an insertion operation at the same time when the main function is performing a deletion operation

CQueue Example

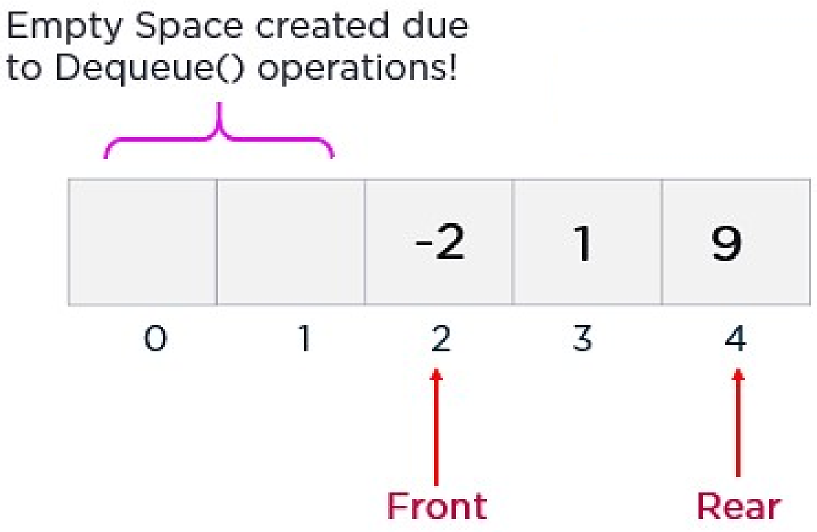
- Traffic light functioning
  - The colors in the traffic light follow a circular pattern
- In page replacement algorithms
  - A circular list of pages is maintained and when a page needs to be replaced, the page in the front of the queue will be chosen

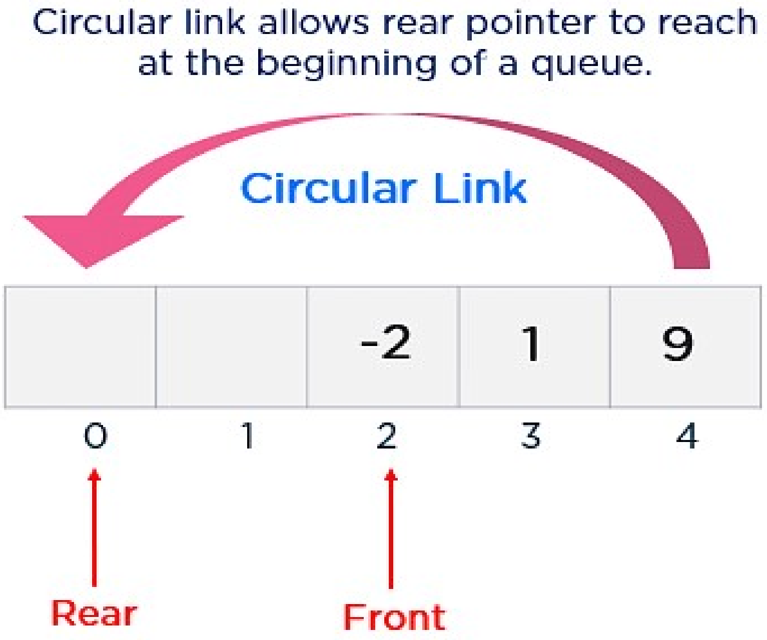
Circular Queue



Circular Queue

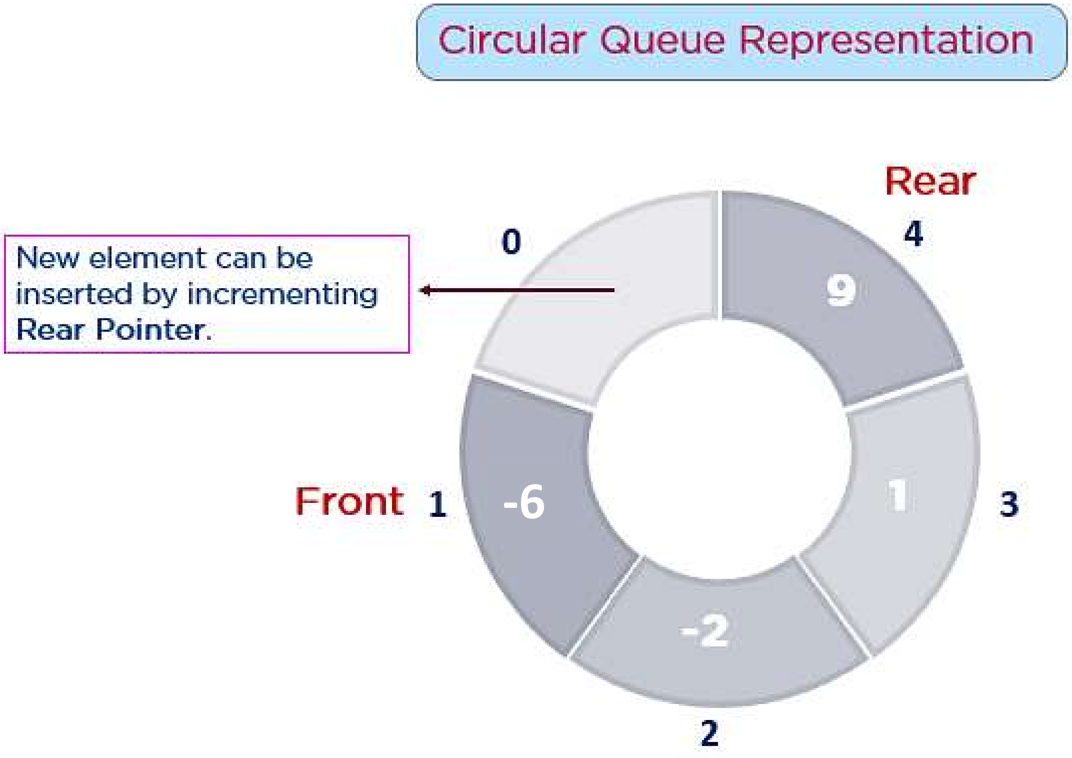
Insert 3


Empty Space created due to Dequeue() operations!


Circular link allows rear pointer to reach at the beginning of a queue.

To Insert 3, first make Rear = 0

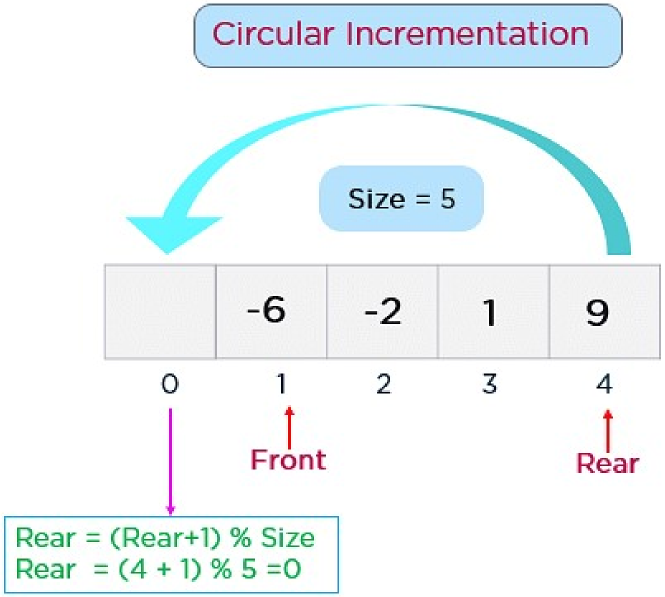
Also named as Ring Buffer


Circular Queue Representation
New element can be inserted by incrementing Rear Pointer.
Rear
0
4
9
Front
1
-6
1
3
-2
2

Circular Queue: Operations

- Front - Used to get the starting element of the Circular Queue.
- Rear - Used to get the end element of the Circular Queue.
- enQueue(value) - Used to insert a new value in the Circular Queue. This operation takes place from the end of the Queue.
- deQueue() - Used to delete a value from the Circular Queue. This operation takes place from the front of the Queue.

Circular Queue: Increment of Rear



Rear = (Rear+1) % Size
Rear = (4 + 1) % 5 =0

- MaxSize of your queue is 5
- Rear pointer = 4
- Enqueue :
- Rear + 1 = 4 + 1 = 5 (Overflow Error)
- Rear = (Rear + 1)% MaxSize = 0 (Reached loc. 0 / Beginning of queue)

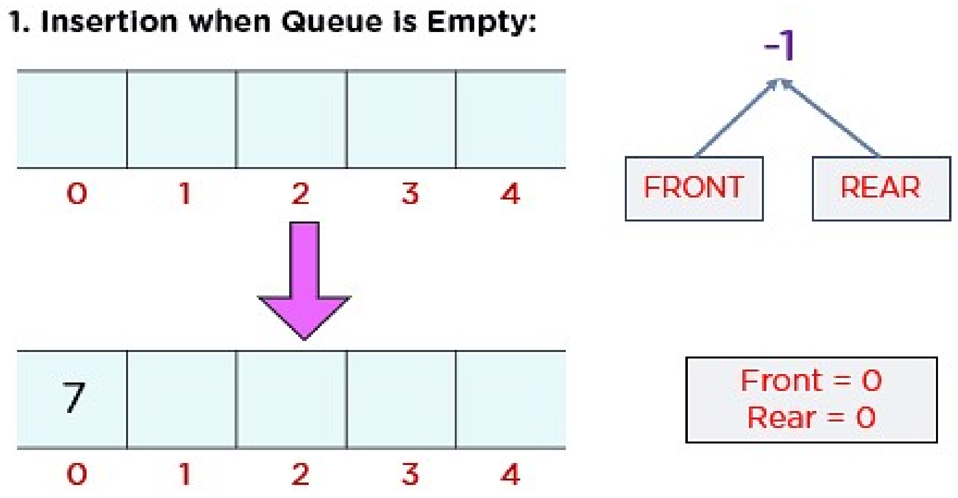
Circular Queue : Enqueue(x) Operation

```
Enqueue(Element)
//If the queue is full, there will be an Overflow error
1: __________________________
2:      print “Overflow”, Exit
//Check if the queue is empty
3: _________________________
        Let Front=0, Rear= 0
4: Else
   _________________________
5: _________________________
6: Exit
```

Circular Queue : Enqueue(x) Operation

```
Enqueue(Element)
//If the queue is full, there will be an Overflow error
1: If  (Rear + 1) % Maxsize = Front
2:      print “Overflow”, Exit
//Check if the queue is empty
3: If(Rear =-1) and (Front = -1) then
        Let Front=0, Rear= 0
4: Else
        Rear = (Rear + 1) % Maxsize
5: Queue[Rear] = Element
6: Exit
```

Circular Queue: Insertion

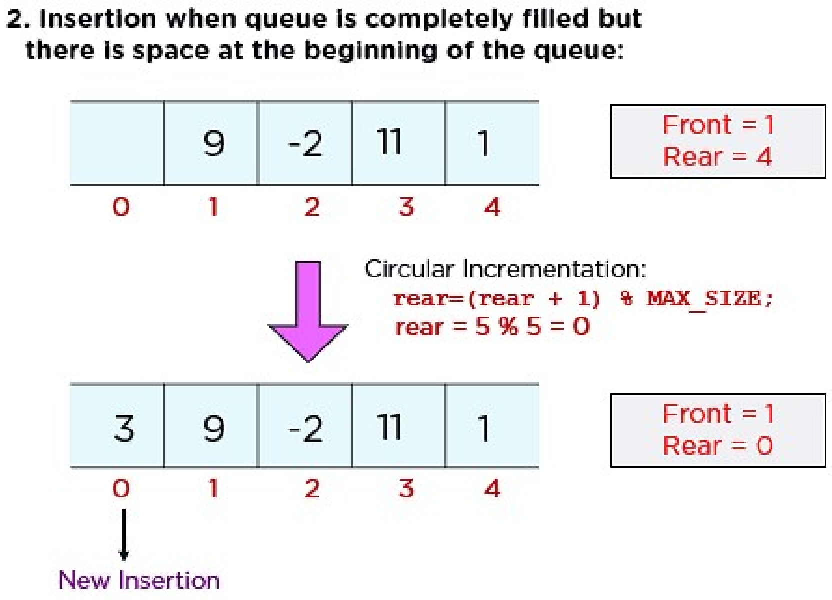


1. Insertion when Queue is Empty:

0	1	2	3	4

0	1	2	3	4
7				

Front = 0
Rear = 0



2. Insertion when queue is completely filled but there is space at the beginning of the queue:

0	1	2	3	4
	9	-2	11	1

Front = 1
Rear = 4

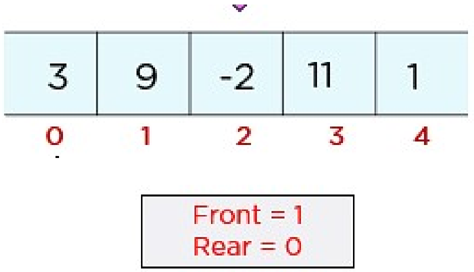
Circular Incrementation:

```
rear=(rear + 1) % MAX_SIZE;
rear = 5 % 5 = 0
```

0	1	2	3	4
3	9	-2	11	1

Front = 1
Rear = 0

3. Insertion in Full Queue



0	1	2	3	4
3	9	-2	11	1

Front = 1
Rear = 0

Insertion with Front = 1 and Rear = 0
(Rear+1) % Max size = Front
so Overflow

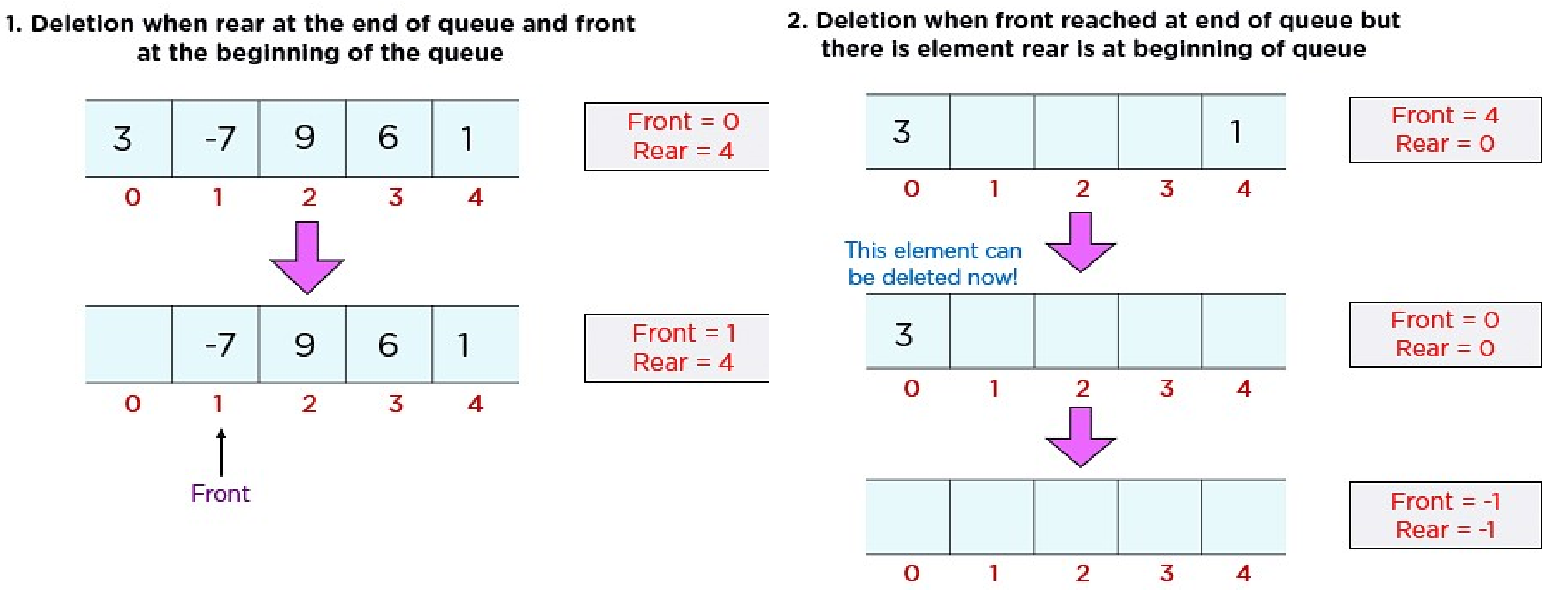
Circular Queue: Deletion Operation

```
Dequeue()
// the queue is empty
1: if (Front = -1 & Rear = -1)
2:      ______________________
3: Let Element = Queue[Front]
// Last element of the queue
4. IF Front = Rear
        ______________________
//Front is  reaching Maxsize, set Front = 0  Otherwise Front = Front + 1
5. else
        ______________________
Step 7: return (Element)
```

Circular Queue: Deletion Operation

```
Dequeue()
// the queue is empty
1: if (Front = -1 & Rear = -1)
2:      print “Queue is Empty”
3: Let Element = Queue[Front]
// Last element of the queue
4. IF Front = Rear
        Let Front = -1, Rear = -1
//Front is  reaching Maxsize, set Front = 0  Otherwise Front = Front + 1
5. else
           Front = (Front+1) % Maxsize
Step 7: return (Element)
```

Circular Queue: Deletion Simulation



1. Deletion when rear at the end of queue and front at the beginning of the queue

3	-7	9	6	1
0	1	2	3	4

Front = 0
Rear = 4

	-7	9	6	1
0	1	2	3	4

Front = 1
Rear = 4

2. Deletion when front reached at end of queue but there is element rear is at beginning of queue

3				1
0	1	2	3	4

Front = 4
Rear = 0

3				
0	1	2	3	4

Front = 0
Rear = 0

0	1	2	3	4

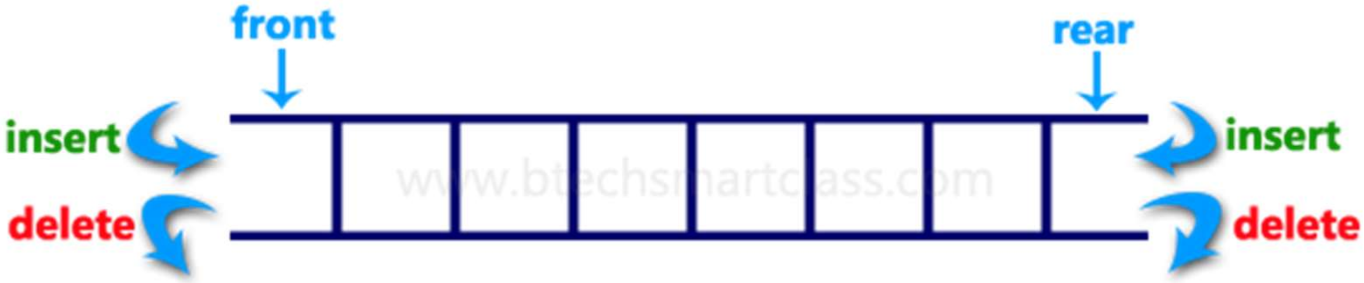
Front = -1
Rear = -1

Circular Queue

- Implementation as Circular Array
  - Space per element excellent
  - Operations very simple / fast

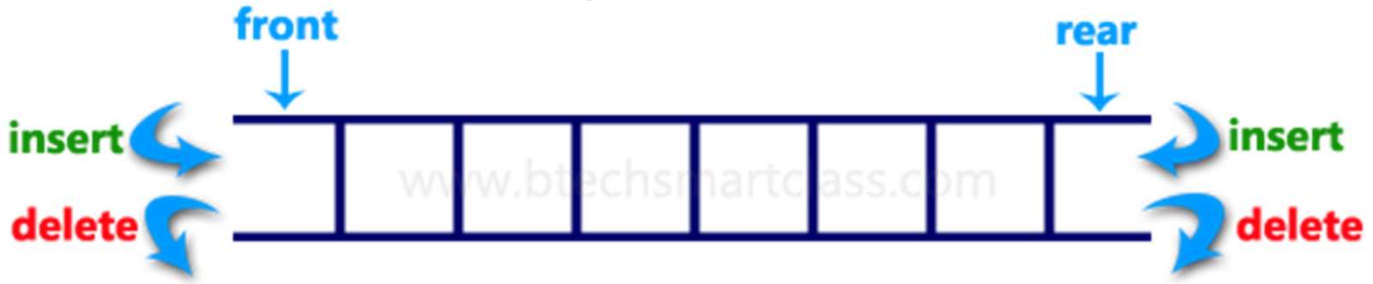
Doubly Ended Queue : DEQUEUE

- A type of queue where the insertions and deletions happen at the front or the rear end of the queue
- Can be implemented either using a circular array or a circular doubly linked list
- Two pointers are maintained, Front and Rear which point to either end of the deque.



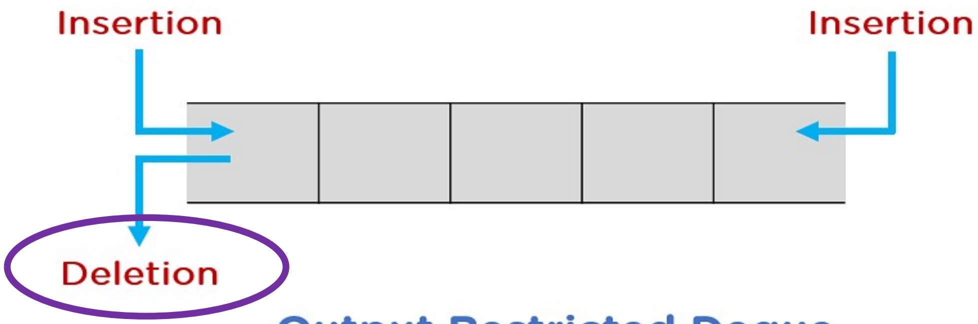
Doubly Ended Queue : DEQUEUE

- A type of queue where the insertions and deletions happen at the front or the rear end of the queue

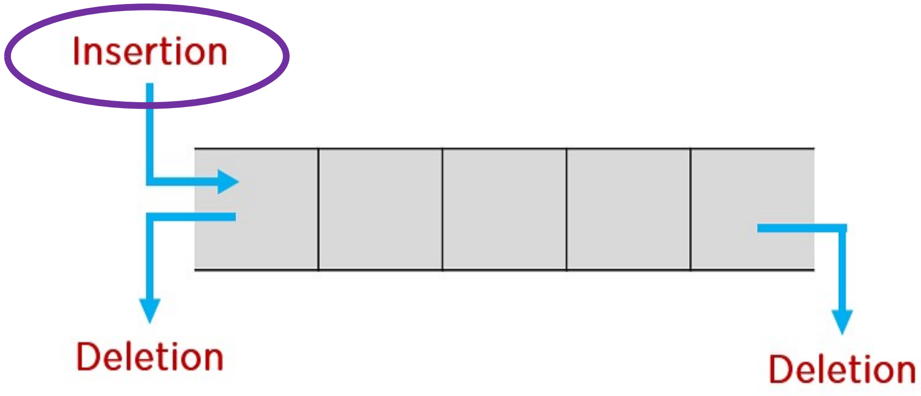


- The various operations that can be performed are:
  1. Insert an element at the front end
  2. Insert an element at the rear end
  3. Delete an element at the front end
  4. Delete an element at the rear end

Doubly Ended Queue : DeQUEUE Variants



Output Restricted Deque



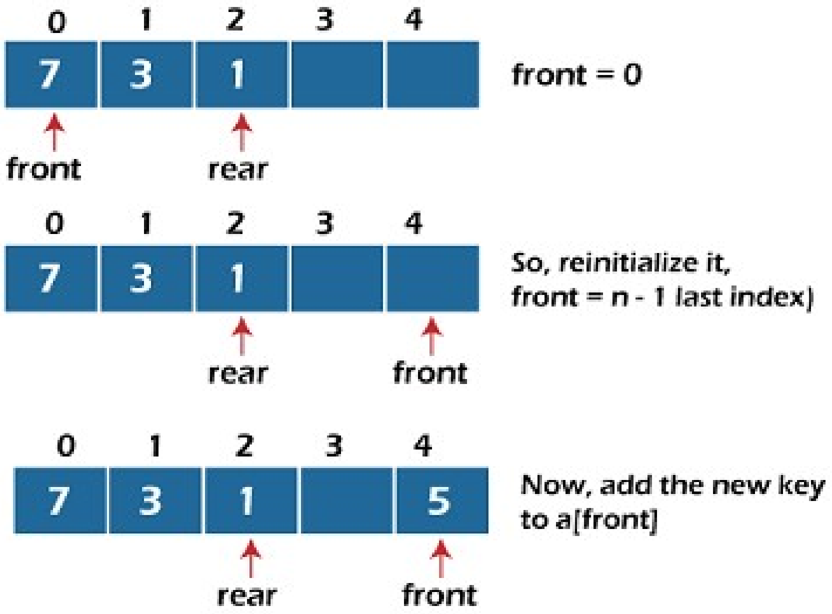
Input Restricted Deque

Basic DeQueue Operations

- **insertFront:** Insert or add an item at the Front of the dequeue
- **insertRear:** Insert or add an item at the Rear of the dequeue
- **deleteFront:** Delete or remove the item from the Front of the dequeue
- **deleteRear:** Delete or remove the item from the Rear of the dequeue
- **getFront:** Retrieves the Front item in the dequeue
- **getRear:** Retrieves the last(Rear) item in the queueue
- **isEmpty:** Checks if the dequeue is empty
- **isFull:** Checks if the dequeue is full

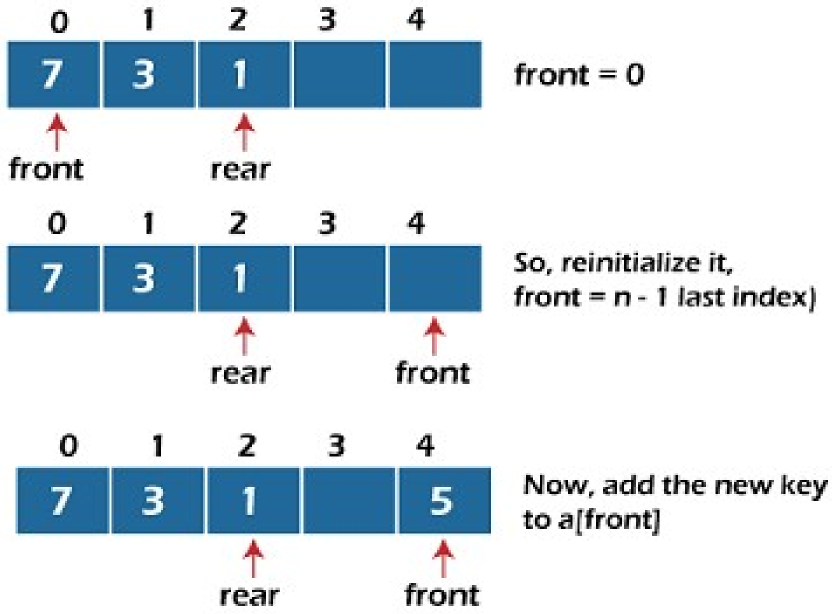
DeQueue: Insertion at the Front (Left)

```
InsertFront (Value)
//If the queue is full, there will be an Overflow error
1: ________________________
2:      print "Overflow", Exit
// Check if the queue is empty, first entry
3: ________________________
        Let Front=0, Rear= 0
4: else //If not the first entry
 ________________________
               _________________
         else
               Front = Front-1
5: ________________________
6: Exit
```



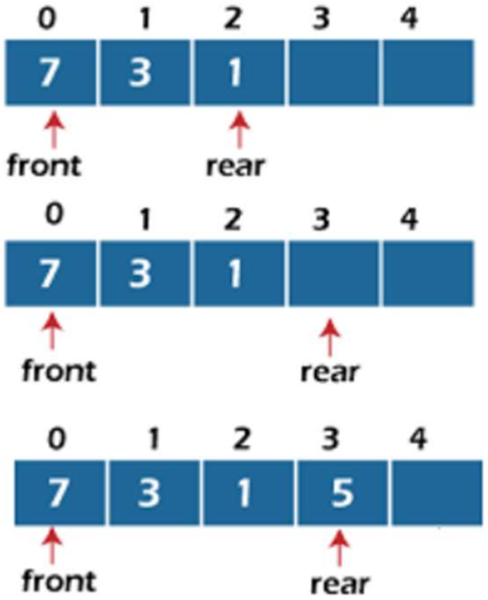
DeQueue: Insertion at the Front (Left)

```
InsertFront (Value)
//If the queue is full, there will be an Overflow error
1: If  (Rear + 1) % Maxsize = Front
2:      print “Overflow”, Exit
// Check if the queue is empty, first entry
3: If(Rear =-1) and (Front = -1) then
        Let Front=0, Rear= 0
4: else //If not the first entry
         If Front = 0 then
               Front = Max-1
         else
               Front = Front-1
5:  Q[Front] = Value
6: Exit
```



DeQueue: Insertion at the Rear End

```
InsertRear ()
//If the queue is full, there will be an Overflow error
1: ____________________________
2:      print "Overflow", Exit
// Check if the queue is empty, first entry
3: ____________________________
        Let Front=0, Rear= 0
4: else //If not the first entry
          ____________________________
5: ______________________
6: Exit
```

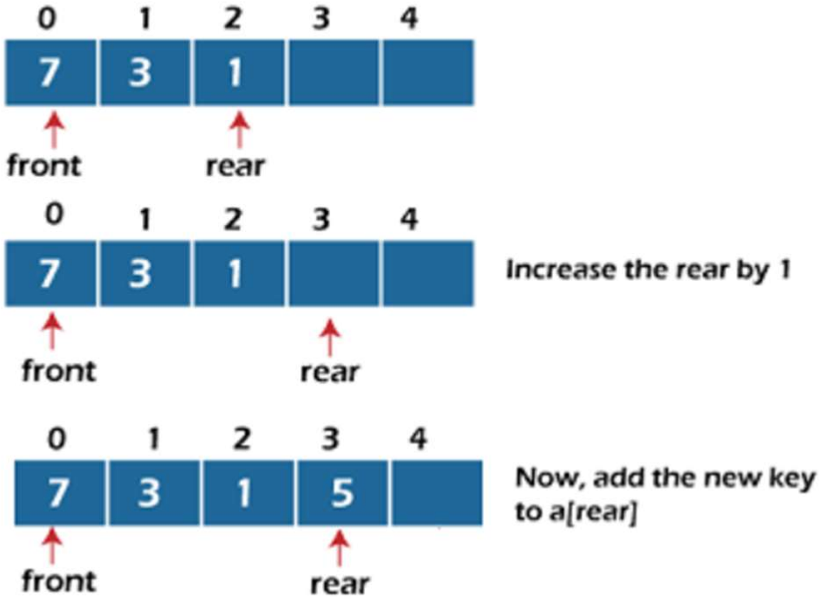


Increase the rear by 1

Now, add the new key to a[rear]

DeQueue: Insertion at the Rear End

```
InsertRear ()
//If the queue is full, there will be an Overflow error
1: If  (Rear + 1) % Maxsize = Front
2:      print “Overflow”, Exit
// Check if the queue is empty, first entry
3: If(Rear =-1) and (Front = -1) then
        Let Front=0, Rear= 0
4: else //If not the first entry
        Rear = (Rear + 1) % Maxsize
5:  Q[Rear] = Value
6: Exit
```

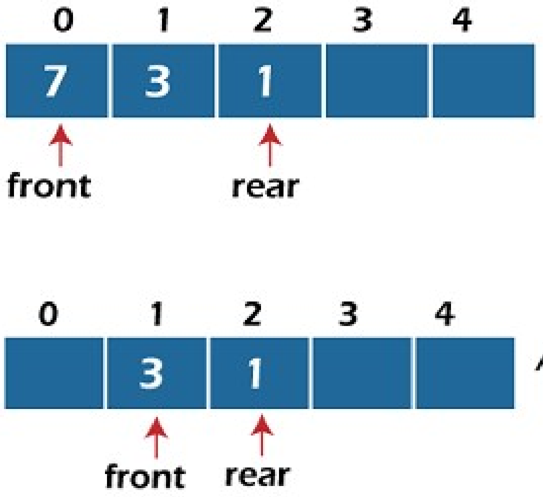


Increase the rear by 1

Now, add the new key to a[rear]

DeQueue: Deletion at the Front

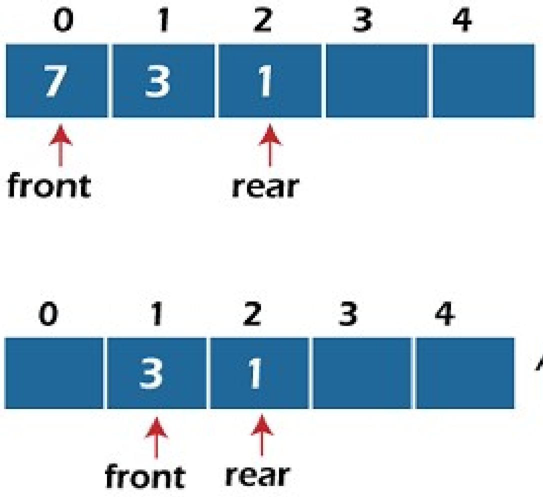
```
deleteFront ()
// the queue is empty
1: if (Front = -1 & Rear = -1)
2:      ____________________
3: ____________________
// Last element of the queue
4. ____________________
        Let Front = -1, Rear = -1
5. Else
        ____________________
6: return(Element)
```



After deleting the element 7 front end

DeQueue: Deletion at the Front

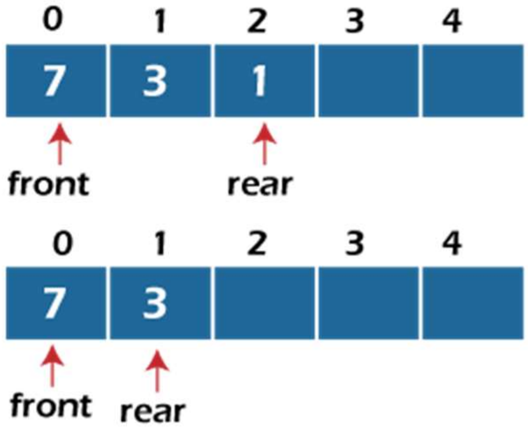
```
deleteFront ()
// the queue is empty
1: if (Front = -1 & Rear = -1)
2:      print “Queue is Empty”
3: Let Element = Queue[Front]
// Last element of the queue
4. IF Front = Rear
         Let Front = -1, Rear = -1
5. Else
        Front = (Front+1)%Maxsize
6: return(Element)
```



After deleting the element 7 front end

DeQueue: Deletion at the Rear End

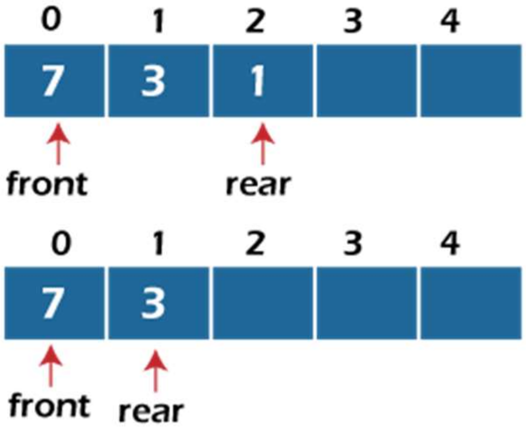
```
deleteRear ()
// the queue is empty
1: ______________________
2:     print “Queue is Empty”
3: ______________________
// Last element of the queue
4. If Front = Rear
        Let Front = -1, Rear = -1
5. Else
6: ______________________
       then set rear = Maxsize - 1
7. Else
      ______________________
8: return(Element)
```



After deleting element 1 from rear end

DeQueue: Deletion at the Rear End

```
deleteRear ()
// the queue is empty
1: if (Front = -1 & Rear = -1)
2:      print "Queue is Empty"
3: Let Element = Queue[rear]
// Last element of the queue
4. If Front = Rear
         Let Front = -1, Rear = -1
5. Else
6: If rear = 0 (rear is at front)
        then set rear = Maxsize - 1
7. Else
        Rear  = Rear-1
8: return(Element)
```



After deleting element 1 from rear end

DeQueue: Applications

- Palindrome checking
  - Deques can be used to check if a word or phrase is a palindrome
  - By inserting each character of the word or phrase into a deque, it is possible to check if the word or phrase is a palindrome by comparing the first and last characters, the second and second-to-last characters, and so on.
- Task scheduler
  - Deques can be used to implement a task scheduler that keeps track of tasks to be executed
  - Tasks can be added to the back of the deque, and the scheduler can remove tasks from the front of the deque and execute them.

DeQueue: Applications

- Multi-level undo/redo functionality
  - Deques can be used to implement undo and redo functionality in applications
  - Each time a user performs an action, the current state of the application is pushed onto the deque
  - When the user **undoes** an action
    - The **front of the deque is popped**, and the **previous state is restored**
  - When the user **redoes** an action
    - The **next state is popped** from the **deque**
- In computer
  - Used in many algorithms like LRU Cache, Round Robin Scheduling

Priority Queue

- A collection of items and their “priorities”
- A data structure arranges the elements in either ascending or descending order
- Allows quick access/removal to the “top priority” thing
- Usually a **smaller priority value** means the item is “more important”
- Rules of processing elements of a priority queue are:
  - An element with **higher priority** is processed **before an element with lower priority**
  - Two elements with **same priority** are processed on a **first come first served** (FCFS) basis
  - The priority queue moves the highest priority elements at the beginning of the priority queue and the lowest priority elements at the back of the priority queue
- Supports only those elements that are comparable
  - The “priority” value can be any number so long as it is comparable

Priority Queue

- Operations
- Insert (item, priority)
  - Add a new item to the PQ with indicated priority
- RemoveMin/Extract
  - Remove and return the “top priority” item from the queue
  - Usually the item with the smallest priority value (i.e. highest priority item)

MinPriority Queue

- RemoveMax/Extract
  - Remove and return the “top priority” item from the queue
  - If the item with the highest priority value (i.e. highest priority item)

MaxPriority Queue

- IsEmpty
  - Indicate whether or not there are items still on the queue

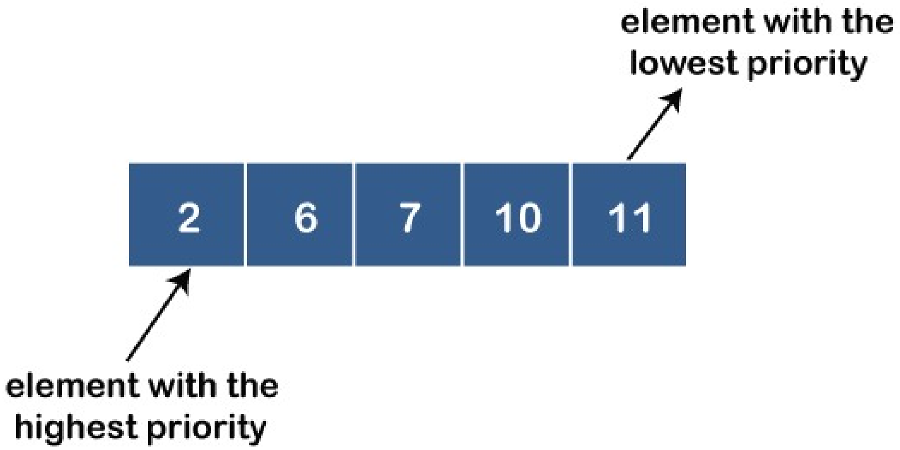
Priority Queue

- More operations based on Applications
  - increaseKey (element, amount)
  - decreaseKey (element, amount)
  - newQueue = union (oldQueue1, oldQueue2)

Types of priority queue

Ascending Order Priority Queue

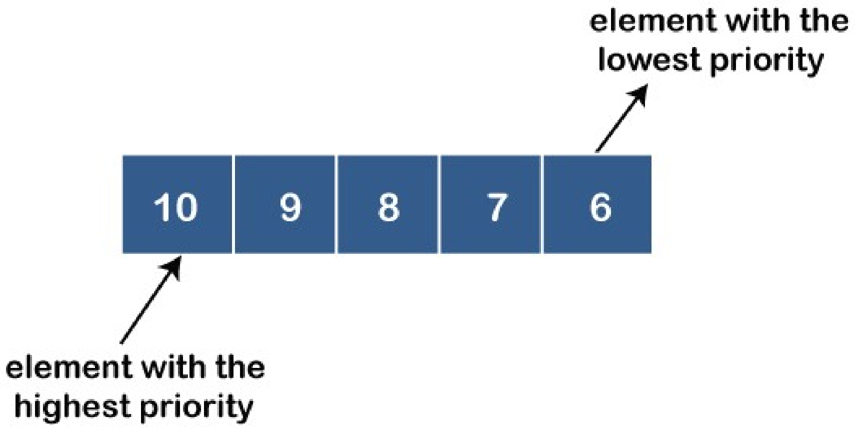
- In ascending order priority queue, a **lower priority number** is given as a **higher priority in a priority**


element with the lowest priority
2
6
7
10
11
element with the highest priority

Types of priority queue

Descending Order Priority Queue

- In descending order priority queue, a **higher priority number** is given as a **higher priority in a priority**



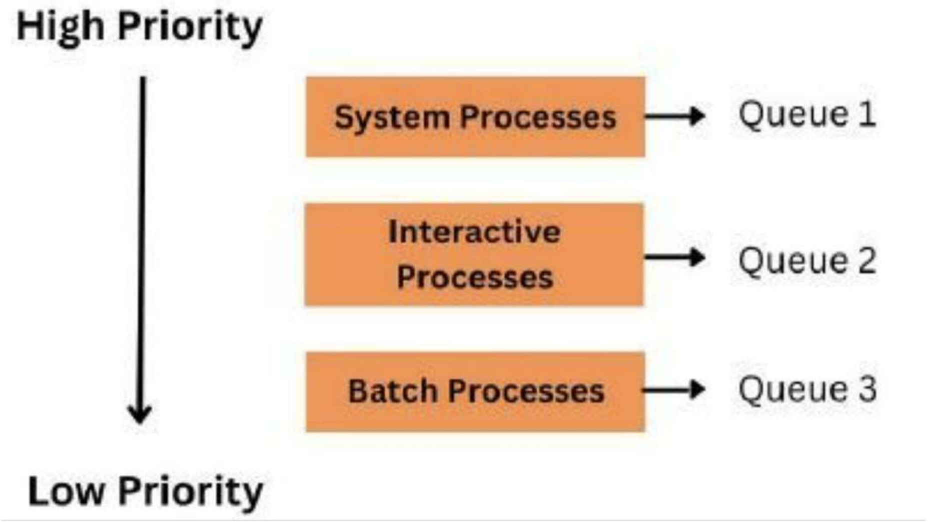
element with the lowest priority

10	9	8	7	6

element with the highest priority

Multilevel Queue

- Operating systems use a particular kind of scheduling algorithm called multilevel queue scheduling to control how resources are distributed across distinct tasks
- It is an adaptation of the conventional queue-based scheduling method, in which **processes are grouped according to their priority, process type, or other factors**

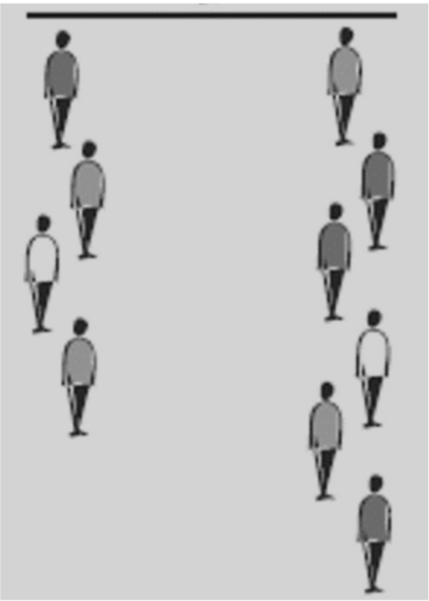


Array Representation of Priority Queues

- Implement a separate queue for each priority number is maintained
- Each of these queues will be implemented using circular arrays or circular queues
- Every individual queue will have its own FRONT and REAR pointers
- Can use a two-dimensional array for this purpose where each queue will be allocated same amount of space
- Given the front and rear values of each queue, a two dimensional matrix can be formed

Priority Queue: Applications

- Hospital Management
  - Outdoor Patient, Indoor Patient, Emergency Patient
  - Severeness of Injury
- Airline check-in for {Business class, Economy class}
  - FIFO within each class



Priority Queue: Applications

Stock Trading

- Investors place orders consisting of three items (**action, price, count**) where
  - Action is either **buy or sell,**
  - Price is the worst price you are willing to pay for the purchase or get from your sale
  - Count is the number of shares
- All the buy orders (bids) have prices lower than all the sell orders
- **Homework due**
  - Priority?

Priority Queue: Applications

- Simulations
  - Can simulate the bank to get some idea of how long customers must wait
  - Bank-customer arrival times and transaction times, to decide how many tellers are needed
    - Assume we have a way to generate random inter-arrival times
    - Assume we have a way to generate transaction times

Priority Queue: Applications

- Scheduling jobs to run on a desk/computer
  - Default priority = arrival time
  - Priority can be changed by operator
- Scheduling events to be processed by an event handler
  - Interrupt handling
    - When programming a real-time system that can be interrupted
    - It is necessary to process the interrupts immediately before proceeding with the current job
  - Load balancing and Job scheduling in Process management
  - Default priority = time of occurrence
  - Higher priority will be with CPU task rather than the user task
- Transfer data asynchronously e.g., pipes, file IO, sockets.
- To find shortest path in graph
  - Dijkstra's Algorithm and Prim's Algorithm
- Heap sort

Double-Ended Priority Queues

- Primary operations
  - Insert
  - Remove Max
  - Remove Min

  - Single-ended priority queue supports just one of the above remove operations