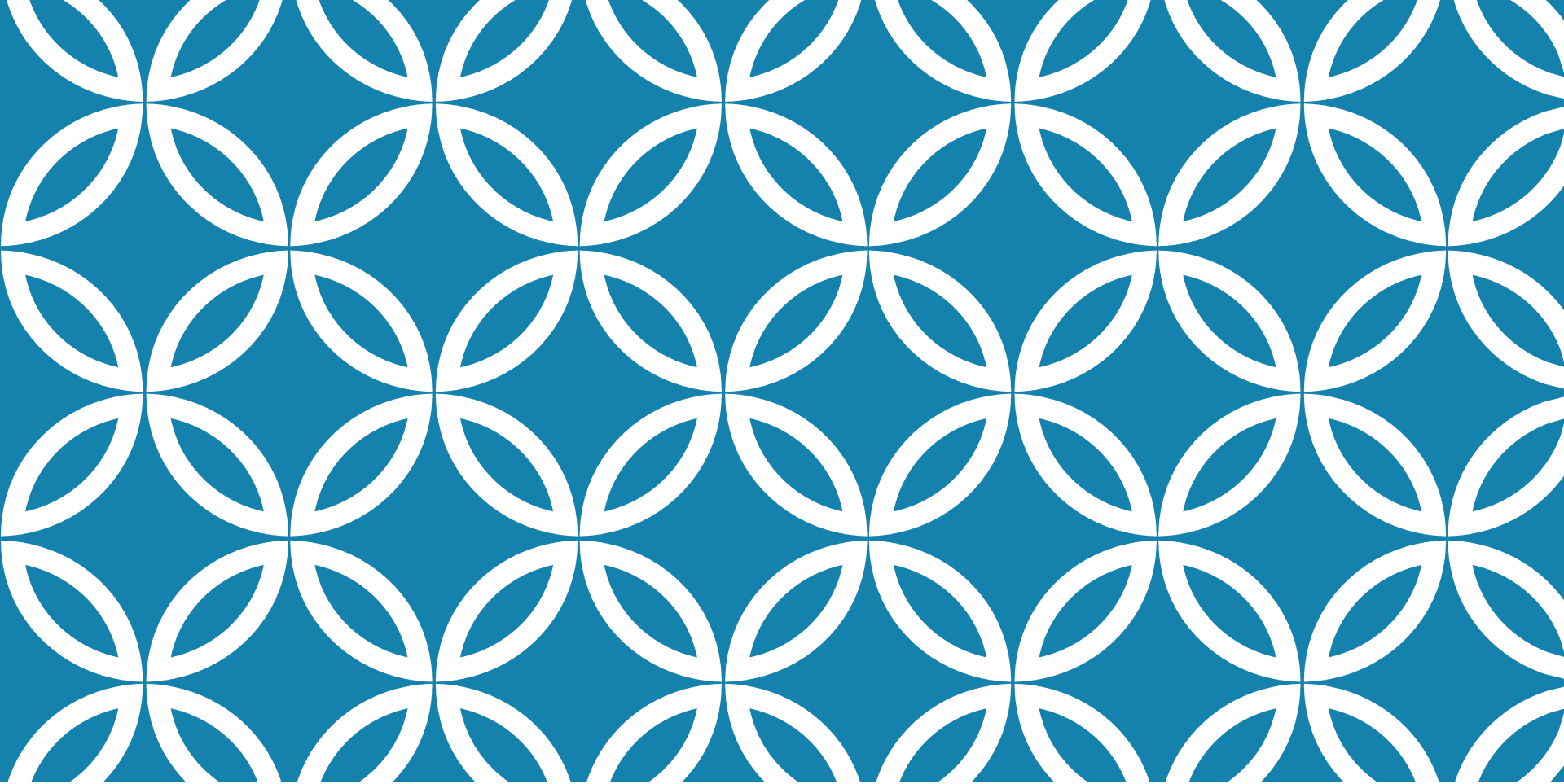


DATA STRUCTURES AND ALGORITHMS

Queues

Hans Christian K.

QUEUES

A **Queue** is a special kind of list, where items are inserted at one end (**the Rear/Tail**) And deleted at the other end (**the Front/Head**).

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

MAKENULL:

ENQUEUE(*x*,Q): Inserts element x at the end of Queue Q.

DEQUEUE(Q): Deletes the first element of Q.

FRONT(Q): Returns the first element on Queue 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

QUEUES IMPLEMENTATIONS

Static

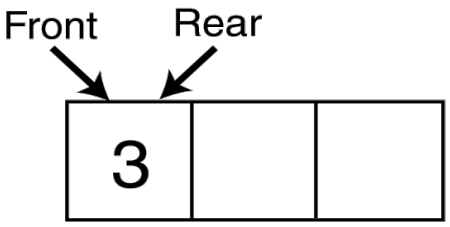
- Queue is implemented by an array, and size of queue remains fix

Dynamic

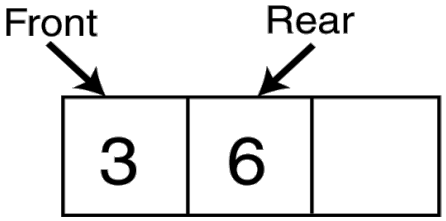
- A **queue** can be **implemented** as a **linked list,** and *expand* or *shrink* with each *enqueue* or *dequeue* operation.

Static Queue Type 1 : Static Front

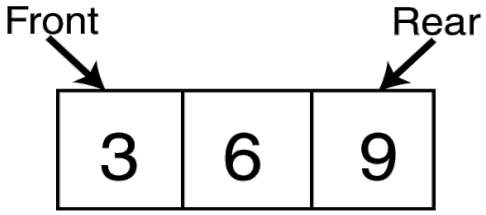
Enqueue(3);



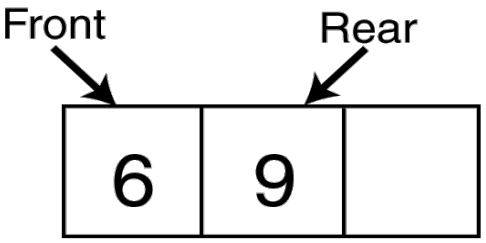
Enqueue(6);



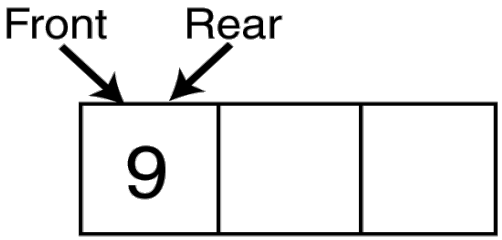
Enqueue(9);



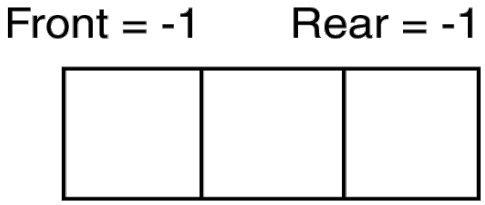
Dequeue();



Dequeue();



Dequeue();



Static Queue Type 2 : BASIC QUEUE | ENQUEUE

			<-A (H=0,T=0)
A (H,T)			<-B
A (H)	B (T)		<-C
A (H)	B	C (T)	<-D (FULL)
	B (H)	C (T)	<-D (FULL)

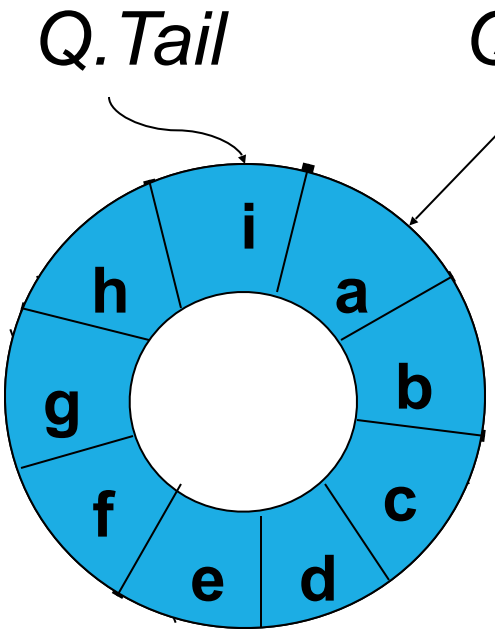
BASIC QUEUE | DEQUEUE

	A (H)	B (T)	Dequeue
		B (H, T)	Dequeue
			H=0,T=0

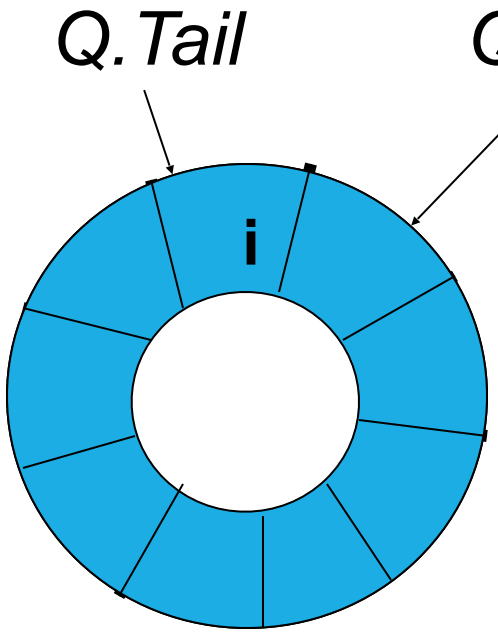
Static Queue Type 2 : SHIFTING QUEUE | ENQUEUE

			<-A (H=0,T=0)
A (H,T)			<-B
A (H)	B (T)		<-C
A (H)	B	C (T)	<-D (FULL)
A (H)	B	C (T)	DEQUEUE DEQUEUE
		C (H,T)	<-D
C (H)	D (T)		

CIRCULAR QUEUE



A Completely Filled Queue



A Queue with Only 1 Element

Static Queue Type 3 : CIRCULAR QUEUE | ENQUEUE

A (H)	B (T)		<-C
A (H)	B	C (T)	DEQUEUE
	B (H)	C (T)	DEQUEUE
		C (H,T)	<-D
D (T)		C (H)	<-E
D	E (T)	C (H)	<-F (Full)
D	E (T)	C (H)	

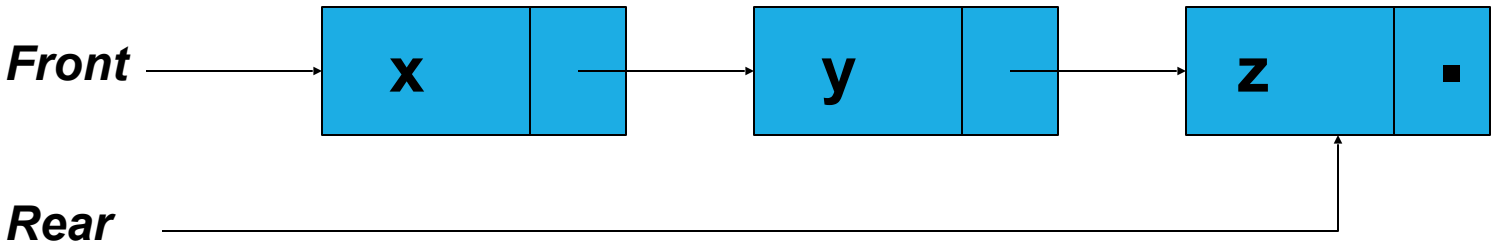
CIRCULAR QUEUE | DEQUEUE

	B (H)	C (T)	DEQUEUE
		C (H,T)	<-D,E
D	E (T)	C (H)	DEQUEUE
D (H)	E (T)		DEQUEUE
	E (H,T)		DEQUEUE
			H=0,T=0

DYNAMIC IMPLEMENTATION OF QUEUES

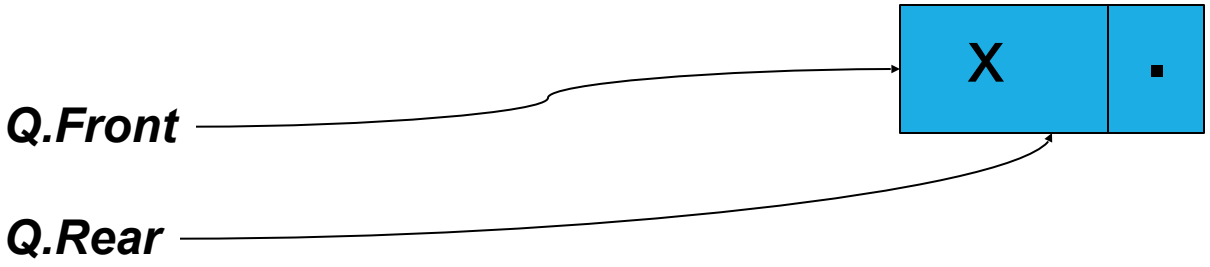
Dynamic implementation is done using pointers.

- HEAD / FRONT: A pointer to the first element of the queue.
- TAIL / REAR: A pointer to the last element of the queue.

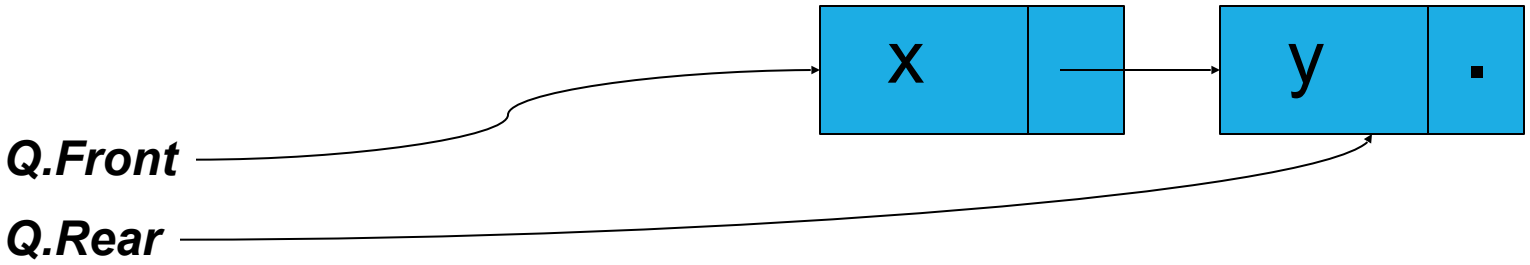


DYNAMIC IMPLEMENATATION

Enqueue (X)

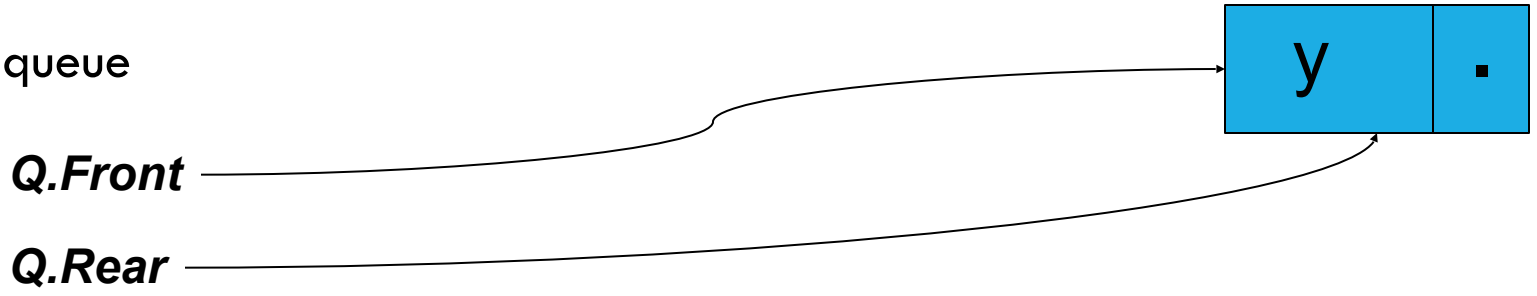


Enqueue (Y)

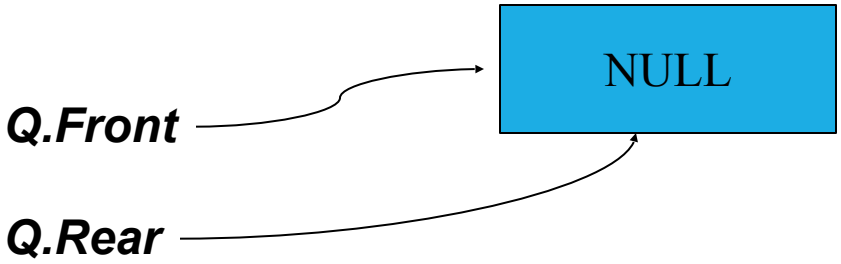


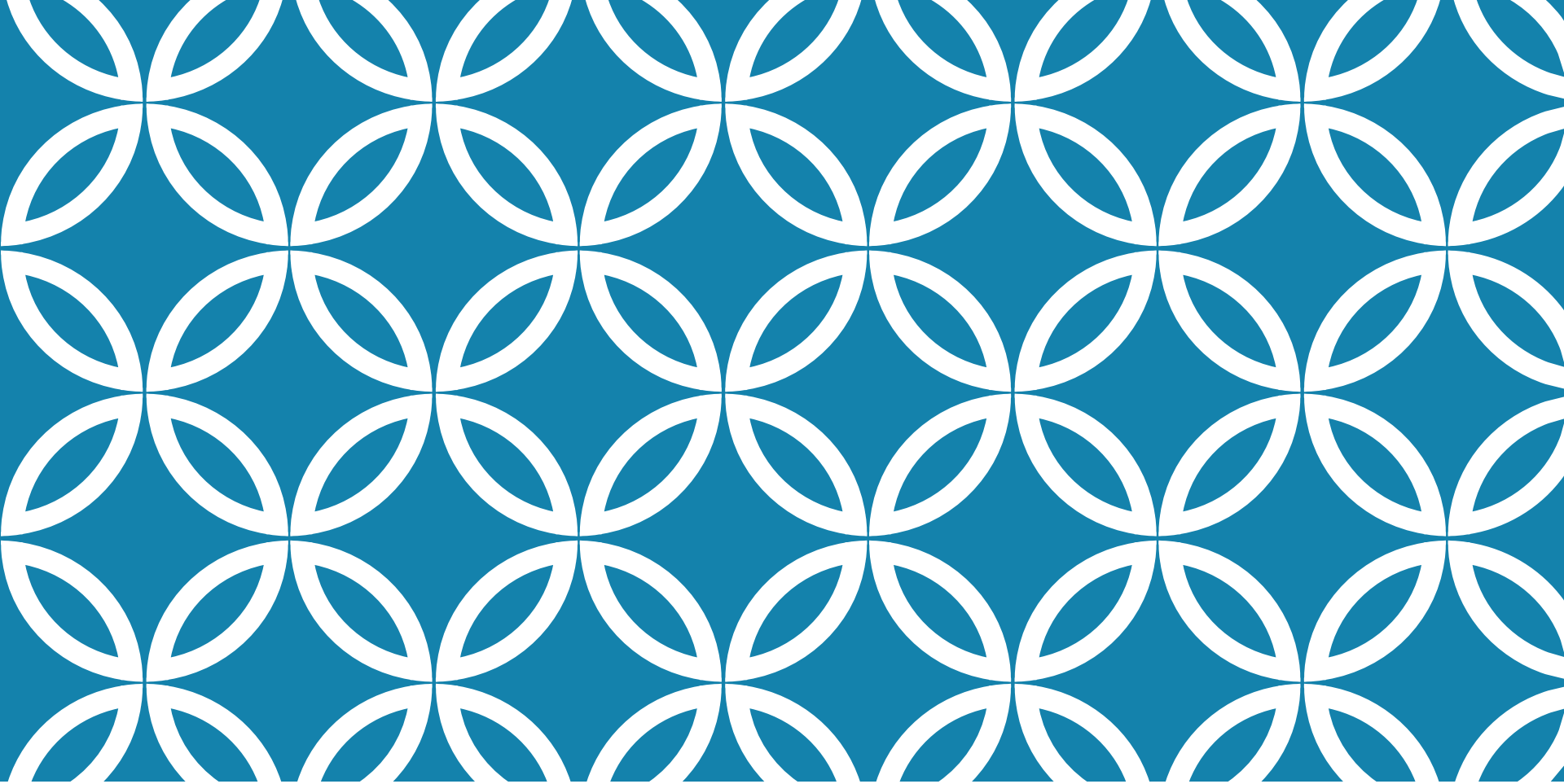
DYNAMIC IMPLEMENTATION

Dequeue



MakeNULL





PRIORITY QUEUES

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

DEFINITION

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, and a
- Descending Priority 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

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

BASIC QUEUE | ENQUEUE

			<-A (H=0,T=0)
A (H,T)			<-B
A (H)	B (T)		<-C
A (H)	B	C (T)	<-D (FULL)
	B (H)	C (T)	<-D (FULL)

BASIC QUEUE | DEQUEUE

	A (H)	B (T)	Dequeue
		B (H, T)	Dequeue
			H=0,T=0

BASIC QUEUE

```
Type TQueue : <Array[1..MAX] of Char
                    Head, Tail : Integer>
Function isEmpty(I Q : TQueue) -> Boolean {
    return (Q.Head == 0 AND Q.Tail == 0)
}

Function isFull(I Q : TQueue) -> Boolean {
    return (Q.Tail == MAX)
}
```

BASIC QUEUE

```
Procedure Enqueue(I/O Q: TQueue, I data: Char) {

}

Function Dequeue(I/O Q: TQueue) -> Char {

}
```

BASIC QUEUE

```
Procedure Enqueue(I/O Q: TQueue, I data: Char) {
    if (isFull(Q)) {
            write (“Full”)
     } else {
            if (isEmpty(Q)) {
                Q.Queue[1] <- data
                Q.Head++  // atau Q.Head = 1
                Q.Tail++    // atau Q.Tail = 1
            } else {
                Q.Tail++
                Q.Queue[Q.Tail] <- data
            }
     }
}
```

BASIC QUEUE

```
Function Dequeue(I/O Q: TQueue) -> Char {
    if (isEmpty(Q)) {
        return “” (default empty value) // print “Empty”
    } else {
        data <- Q.Queue[Q.Head]
        if (Q.Head == Q.Tail) {
            Q.Head <- 0
            Q.Tail <- 0
        } else {
            Q.Head++
        }
        return data
    }
}
```

SHIFTING QUEUE | ENQUEUE

			<-A (H=0,T=0)
A (H,T)			<-B
A (H)	B (T)		<-C
A (H)	B	C (T)	<-D (FULL)
A (H)	B	C (T)	DEQUEUE DEQUEUE
		C (H,T)	<-D
C (H)	D (T)		

SHIFTING QUEUE

```
Procedure Enqueue(I/O Q: TQueue, I data: Char) {

}

Function Dequeue(I/O Q: TQueue) -> Char {
    // Same with Basic Queue
}
```

SHIFTING QUEUE

```
Procedure Enqueue(I/O Q: TQueue, I data: Char) {
    if (isFull(Q)) {
        if (Q.Head == 1) {
            write("Full")
        } else {
            temp <- Q.Head. //3
            i <- 1
            while(temp <= MAX) {
                Q.Queue[i] <- Q.Queue[temp]
                i++; temp++
            }
            Q.Head <- 1;  Q.Tail <- i
        }
    } else if lanjut —>
    } else if (isEmpty(Q) {
        Q.Head++;
        Q.Tail++
    } else {
        Q.Tail++
    }
    Q.Queue[Q.Tail] <- data
}
```

Static Queue Type 3 : CIRCULAR QUEUE | ENQUEUE

A (H)	B (T)		<-C
A (H)	B	C (T)	DEQUEUE
	B (H)	C (T)	DEQUEUE
		C (H,T)	<-D
D (T)		C (H)	<-E
D	E (T)	C (H)	<-F (Full)
D	E (T)	C (H)	

CIRCULAR QUEUE | DEQUEUE

	B (H)	C (T)	DEQUEUE
		C (H,T)	<-D,E
D	E (T)	C (H)	DEQUEUE
D (H)	E (T)		DEQUEUE
	E (H,T)		DEQUEUE
			H=0,T=0

CIRCULAR QUEUE

```
Type TQueue : <Array[1..MAX] of Char
                    Head, Tail : Integer>
Function isEmpty(I Q : TQueue) -> Boolean {
    return (Q.Head == 0 AND Q.Tail == 0)
}

Function isFull(I Q : TQueue) -> Boolean {
    return (Q.Head == 1 AND Q.Tail == MAX) OR
           (Q.Tail+1 == Q.Head)
}
```

CIRCULAR QUEUE

```
Procedure Enqueue(I/O Q: TQueue, I data: Char) {

}

Function Dequeue(I/O Q: TQueue) -> Char {

}
```

DEQue (Double-Ended Queue)