Queue ADT

Data: <value, arrival time>

- Data of same data type
- FIFO
- Access from both sides

Operations:

createNew(capacity, dataType)

- Head and tail
- Head and tail at same position initially

→ create an empty queue

isEmpty() → yes/no

isFull() → yes/no

add(val) / enqueue

→ add element with arrival time larger than any existing element in queue

delete() / dequeue

→ remove element with the oldest arrival time

Queue implementation using array

Data: Array A, integer count

```
createNew(capacity, dataType){
    A.createNew(capacity, dataType)
    count = 0
}

isEmpty(){
    if(count == 0){
        return TRUE
    }
    return FALSE
}

isFull(){
    if(count == A.len){
        return TRUE
    }
    return FALSE
}
```

Queue implementation using array

Data: Array A, integer count

```
createNew(capacity, dataType){                    O(1)
    A.createNew(capacity, dataType)
    count = 0
}

isEmpty(){                                        O(1)
    if(count == 0){
        return TRUE
    }
    return FALSE
}
```

```
isFull(){                                         O(1)
    if(count == A.len){
        return TRUE
    }
    return FALSE
}
```

```
add ( val ) {
    if ( isFull ()) {
        return ERROR
    }
    A. StoreValue ( val, count)
    count = count + 1
}
```

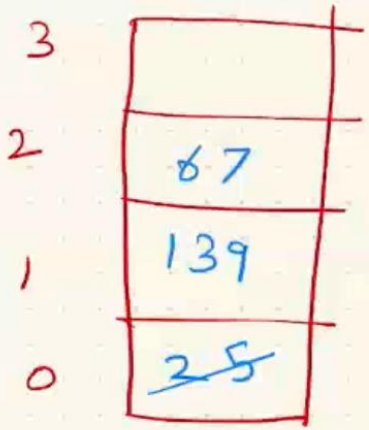
$\Theta(1)$

```
delete () {
    if( isEmpty () ) {
        return ERROR
    }
    answer = A[ 0 ]
```

```
add ( val ) {
    if ( isFull ()) {
        return ERROR
    }
    A.StoreValue ( val, count)
    count = count + 1
}
```

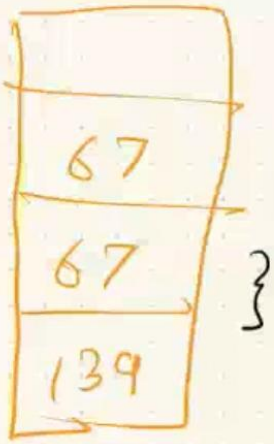
$\Theta(1)$

```
delete () {                                   Θ(n)
    if( isEmpty () ) {
        return ERROR
    }
    answer = A[0]
    for (index=0, index<count-1 , index++ ){
        A[index] = A[index+1]
    }
    count --
    return answer
}
```



answer = 25

count = 3



$(n-1) \times 7$

$\Theta(n) \times \Theta(1)$

Queue implementation with two counters

Data: Array A, integer head, integer tail

```
createNew(capacity){
    A.createNew(capacity)
    head=0    tail=0
}

isEmpty(){
    if(head==tail){
        return TRUE
    }
    return FALSE
}

isFull(){
    if(tail==capacity){
        return TRUE
    }
    return FALSE
}
```

```
add ( val ) {
    if ( isFull() ){
        return ERROR
    }
    A[tail] = val
    tail ++
}
```

```
delete() {
    if(isEmpty()) {
        return ERROR
    }
    answer = A[ head ]
    head++
}
```