Algorithms & Data Structures I
CSC 225

Ali Mashreghi

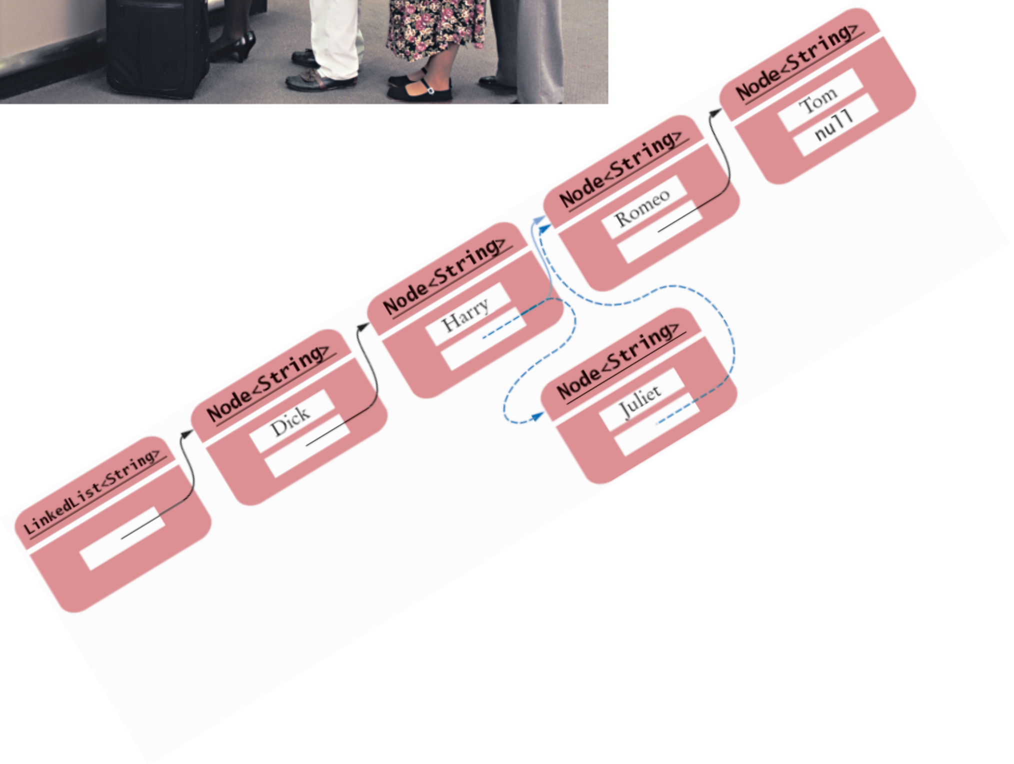
Fall 2018

Department of Computer Science, University of Victoria

Elementary Data Structures

- Stacks
- Queues
- Lists
- Arrays
- Resizable Arrays

X	12	3	7	24	4	1	1
A	12	7.5	7.3	11.5	10	8.5	7.4



Stack

A stack is a collection of items with two interesting operations:

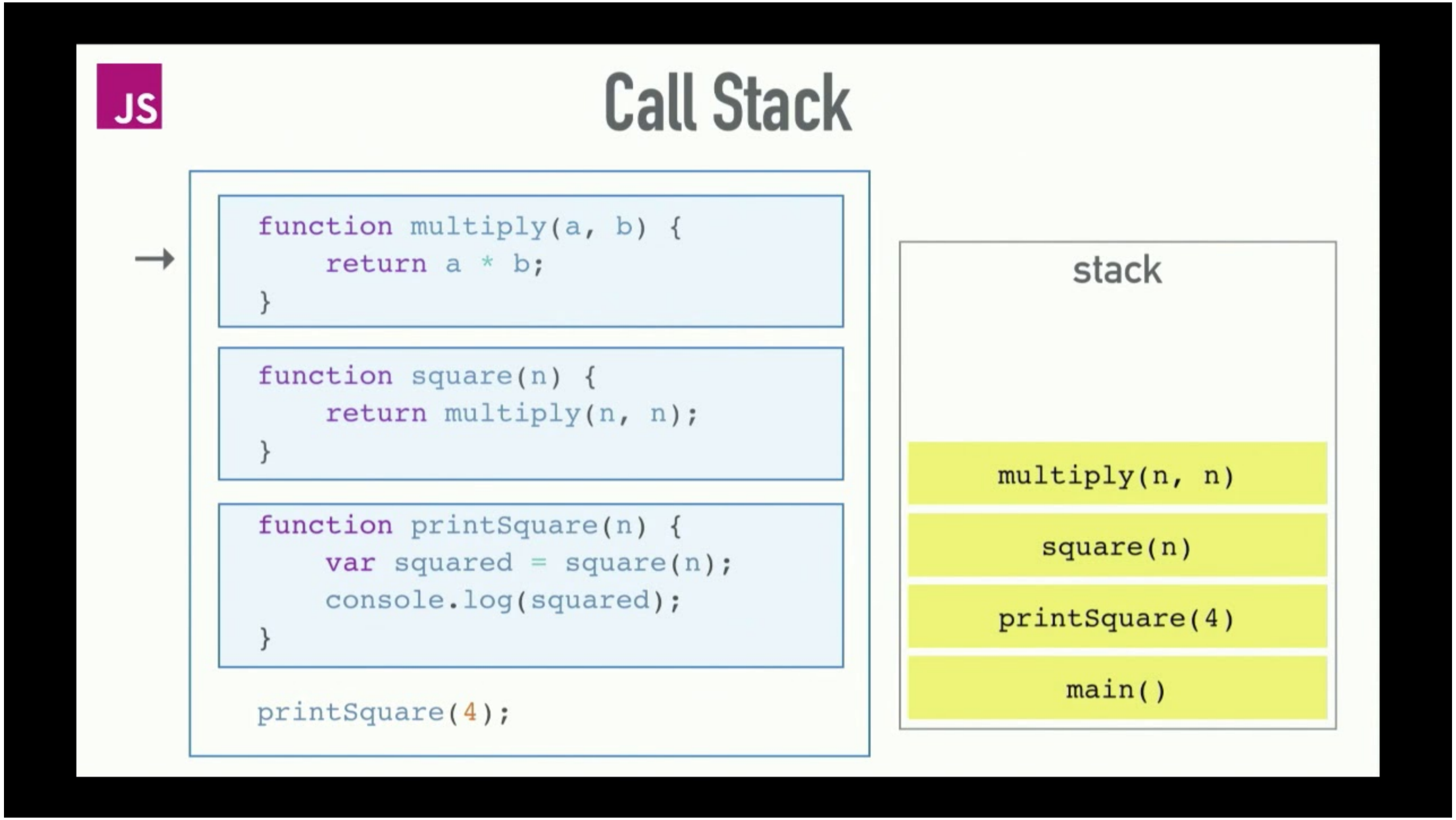
1. **Push:** Putting an item on **top** of the stack
2. **Pop:** Remove an item from the **top** of the stack

Examples in real life:

1. A stack of plates in the cafeteria
2. The undo operation in text editors
3. Keeping the path when browsing files and folders

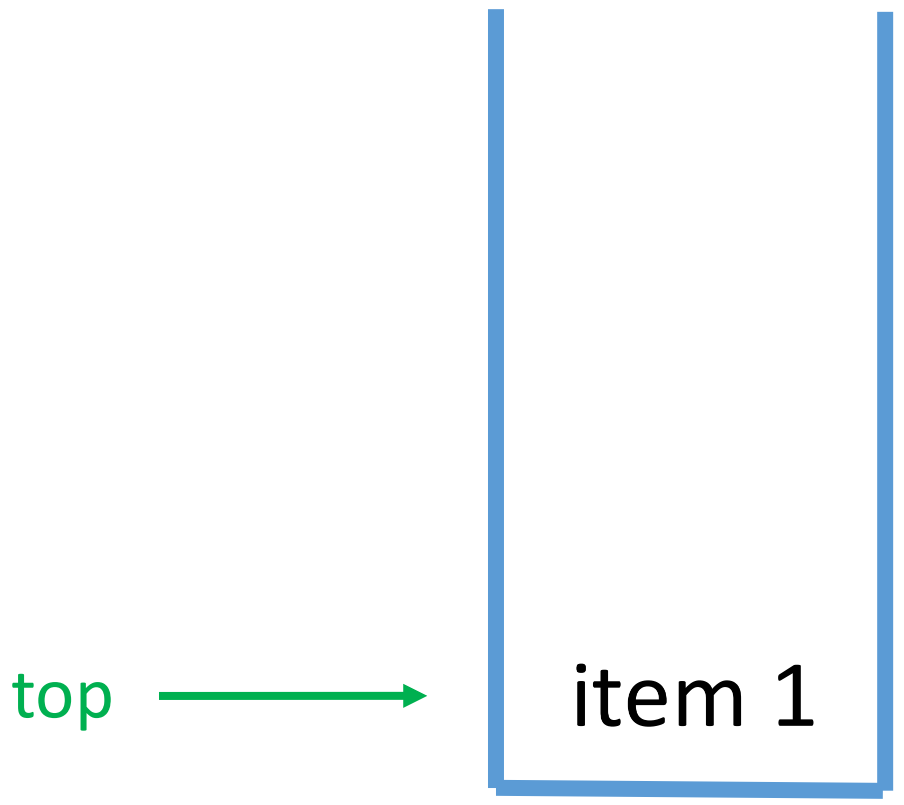
Stack

4. Making nested (or recursive) calls in programming

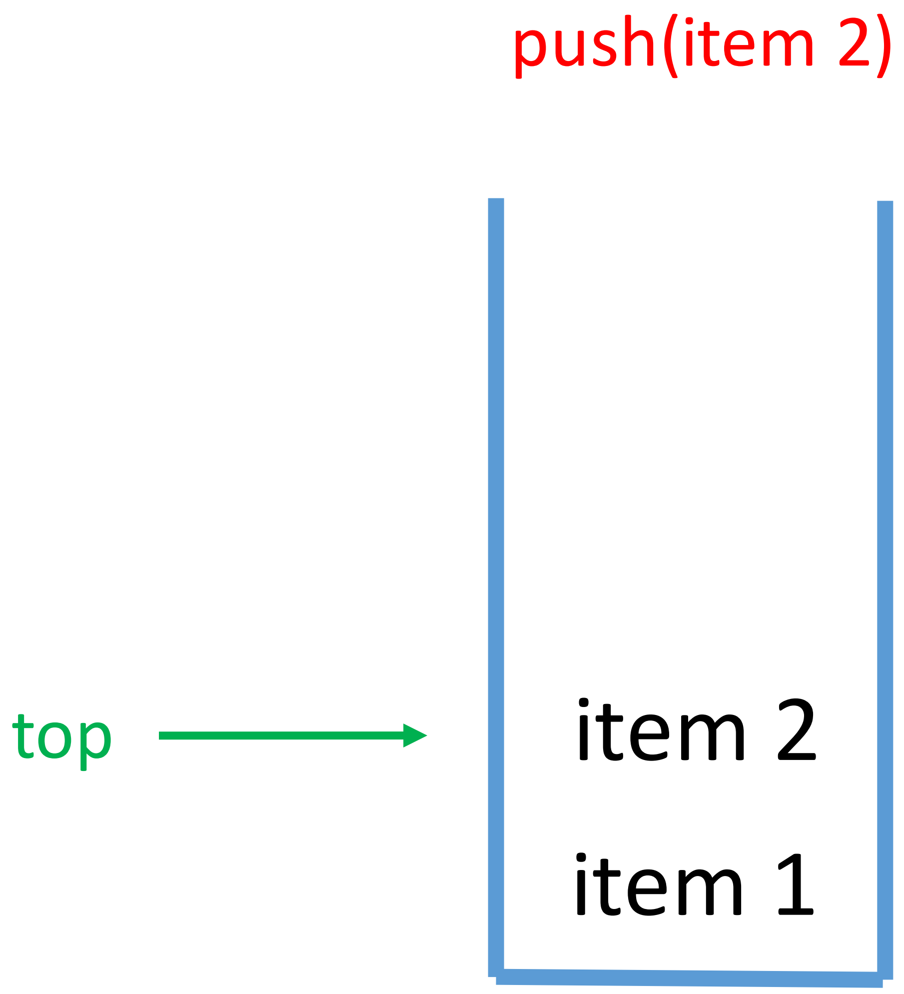


Stack

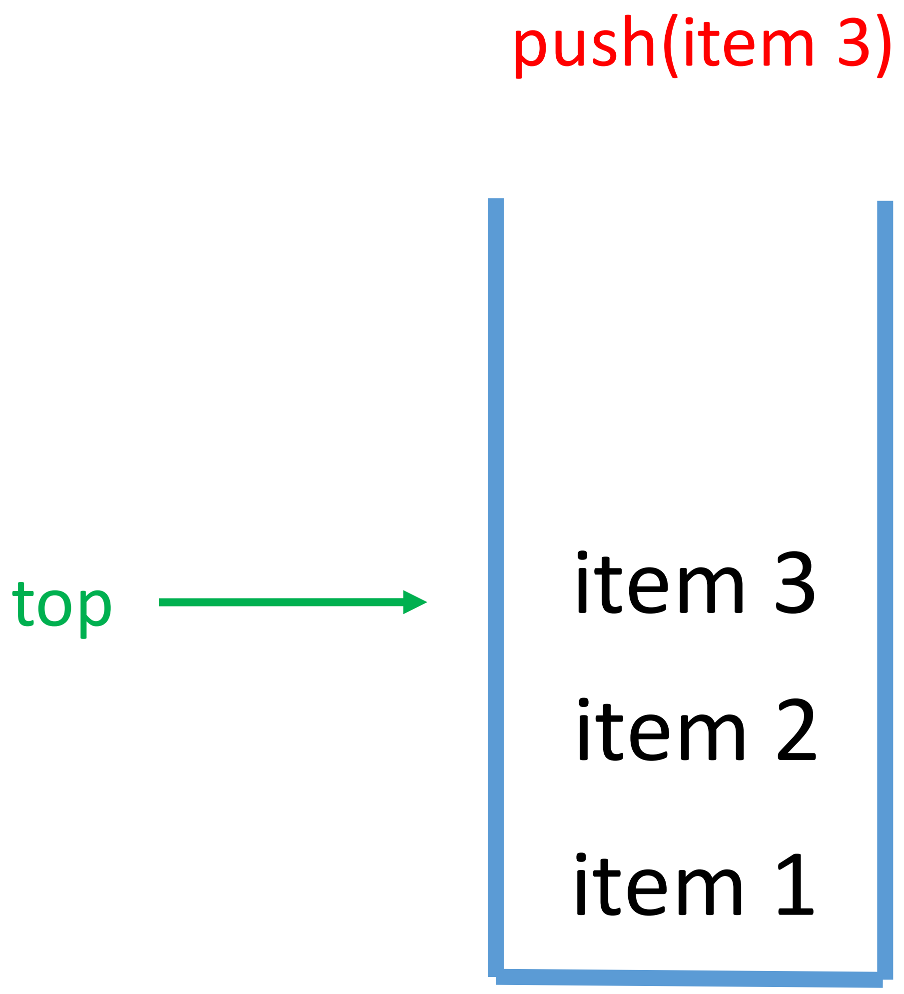
A stack with a single item in it



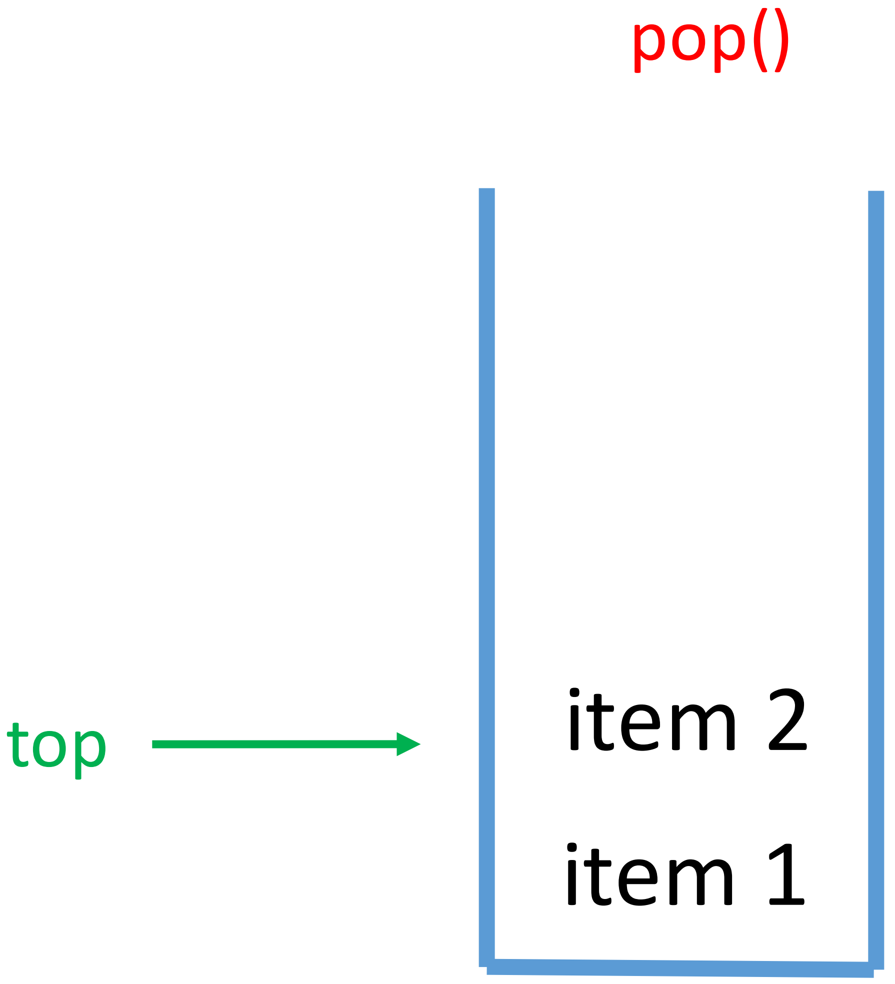
Stack



Stack



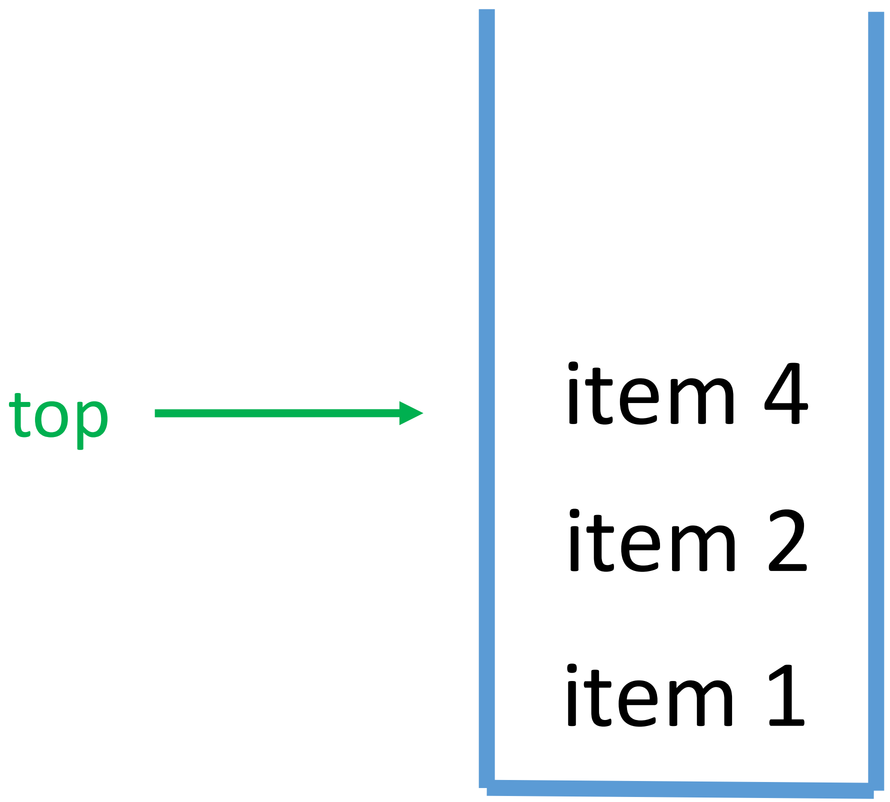
Stack



Stack follows the LIFO (Last In First Out) principle,
i.e. the last item pushed into the stack is the first item that’s popped.

Stack

push(item 4)



At this point you don’t have access to item 1 or item 2
Since only the top of the stack is accessible

Stack ADT

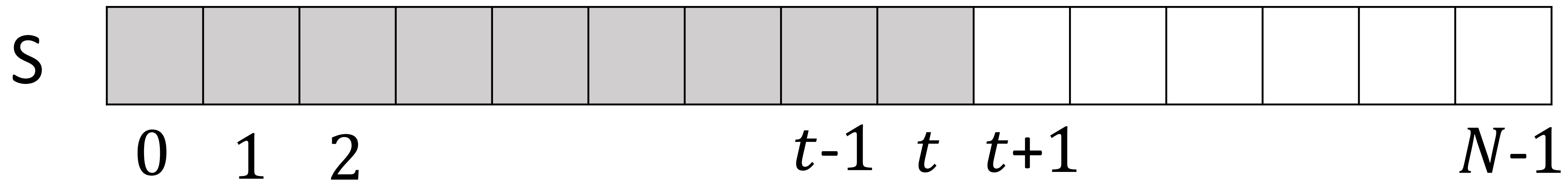
A stack supports the following operations:

- **PUSH(x):** Insert item x at the top of the stack.
- **POP():** Remove from the stack **and return** the item at the top of the stack. An error occurs if the stack is empty.
- **ISEMPTY():** Return True if the stack is empty, False otherwise.
- **TOP():** Return the top item on the stack **without removing** it; an error occurs if the stack is empty.
- **SIZE():** Return the number of items in the stack.

Array-based Implementation

- We can use an array to implement a stack as follows:

1. ***Array S:*** N-element array, with elements stored from $S[0]$ to $S[t]$
2. ***t:*** stack pointer; integer that gives the index of the top element in S
3. ***N***: specified max stack size (e.g., $N = 1000$)



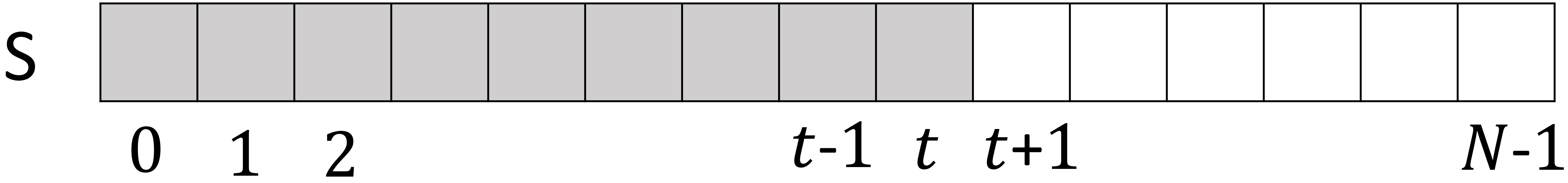
Top(), Size(), and IsEmpty()

Size()

return $t + 1$

Top()

return $S[t]$



Top(), Size(), and IsEmpty()

Size()

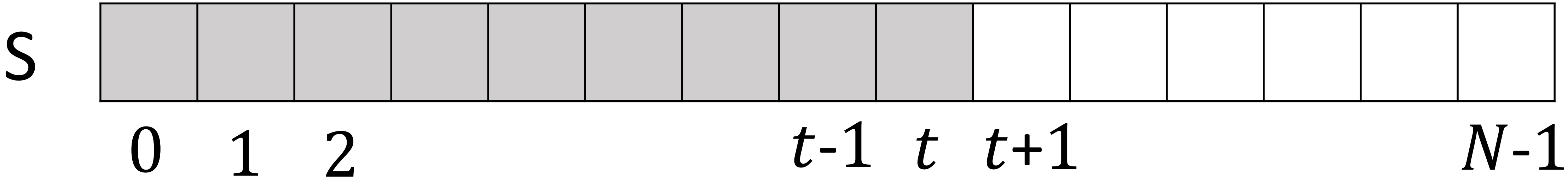
```
return t + 1
```

Top()

```
return S[t]
```

IsEmpty()

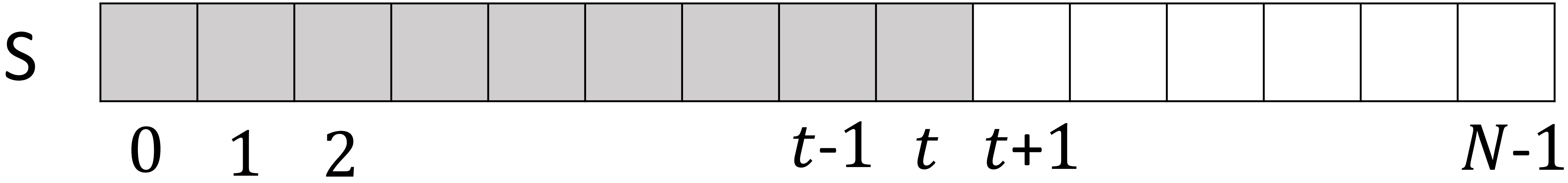
```
if Size() == 0
    return TRUE
return FALSE
```



PUSH(x)

PUSH(x)
if SIZE() == N
  error "stack is full"
  return
$t = t + 1$
$S[t] = x$

Also called overflow

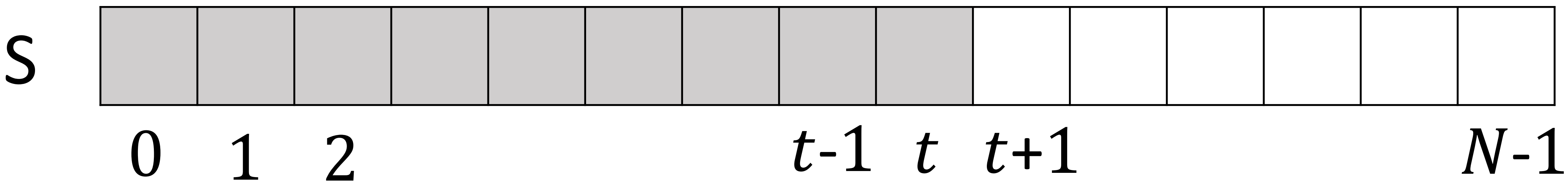


Pop()

```
Pop()
    if IsEmpty() == TRUE
        error "stack is empty"
        return NULL
    item = S[t]
    t = t − 1
    return item
```

Also called underflow

In an empty stack t is -1

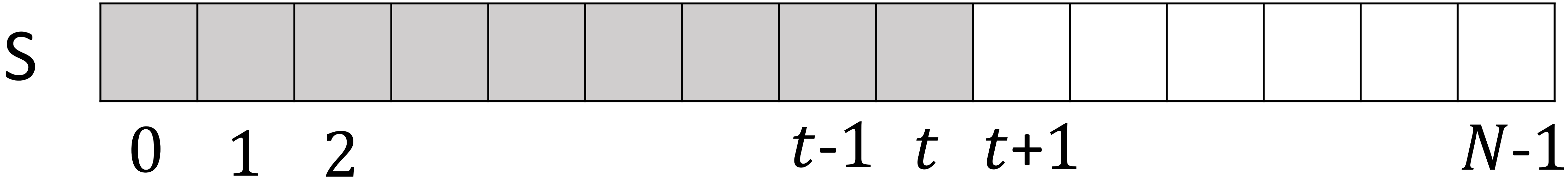


Pros and Cons

Pros: Simple and efficient: $O(1)$ per operation

Cons:

1. The stack ***must*** assume a fixed upper bound *N*
2. Memory might be wasted or a stack-full error can occur!
3. If a good estimate for the stack size is known, array is the best choice.



Queue

Queue

A queue is a collection of items with two interesting operations:

1. **Enqueue**
2. **Dequeue**

Queue

A queue is a collection of items with two interesting operations:

1. **Enqueue**
2. **Dequeue**

NOT

Queue

A queue is a collection of items with two interesting operations:

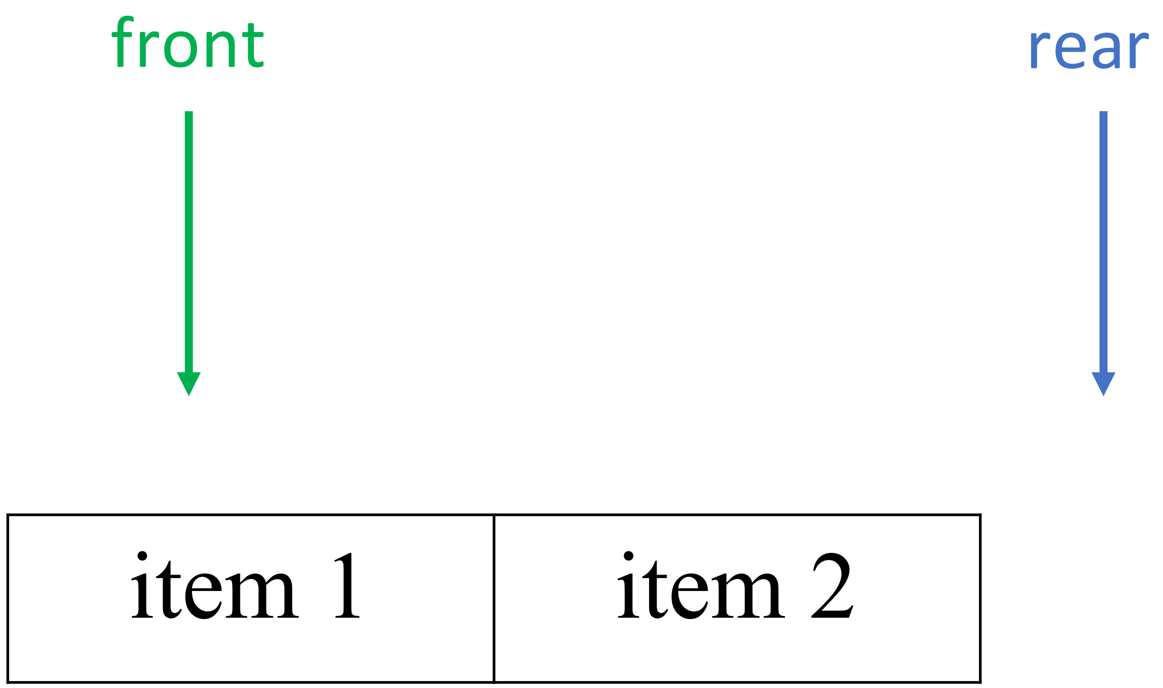
1. **Enqueue:** Insert an item at the **rear** of the queue
2. **Dequeue:** Remove an item from the **front** of the queue

Examples in real life:

1. A line up in a bank
2. Resource sharing in operating systems

A queue follows the FIFO (First In First Out) principle.

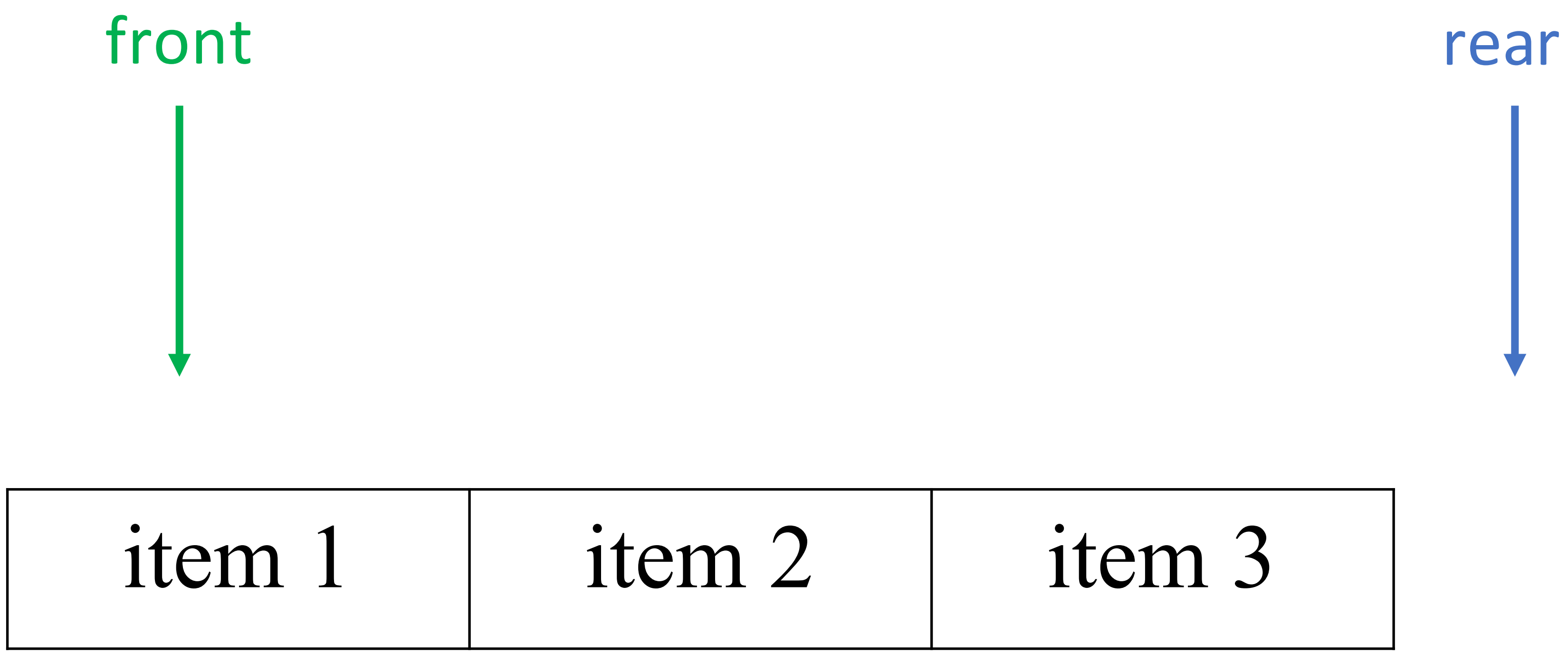
Queue



Note: rear does not point to a position in the queue but rather to a position that the next item can be inserted at.

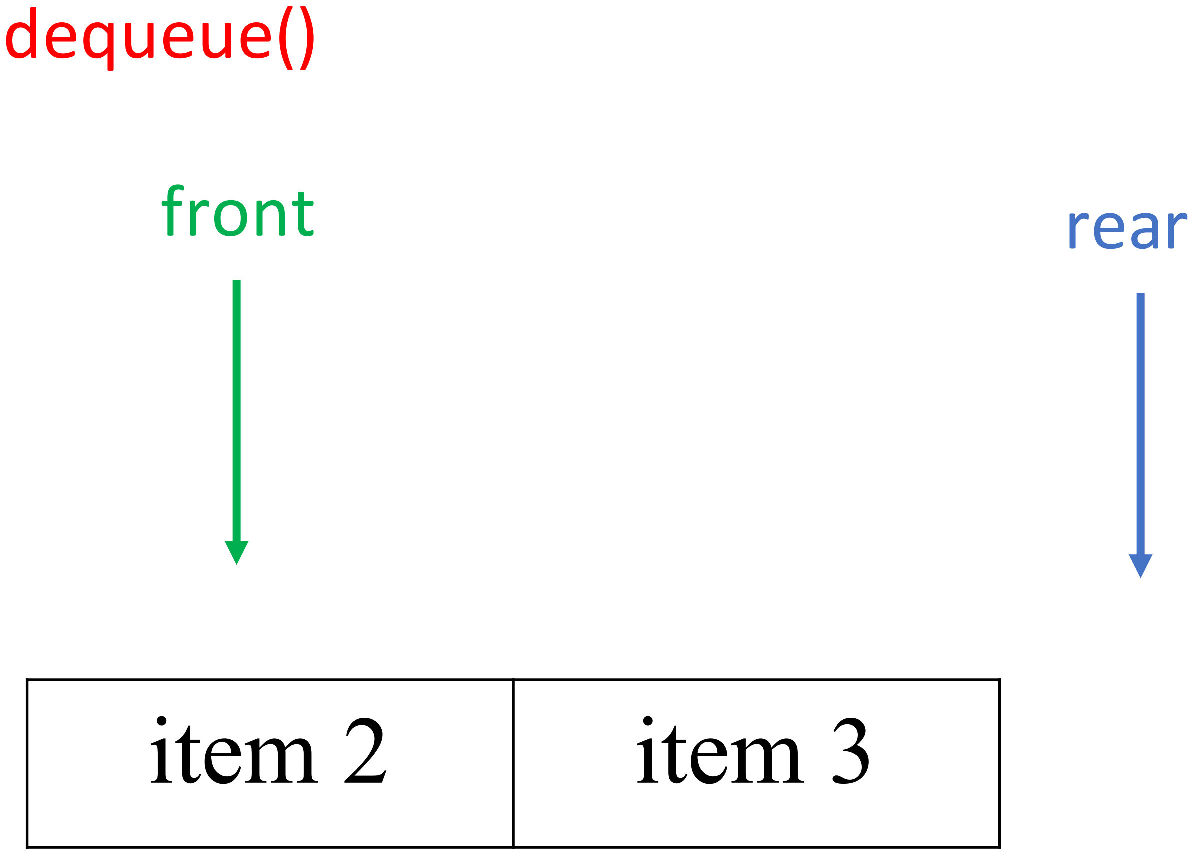
Queue

enqueue(item 3)

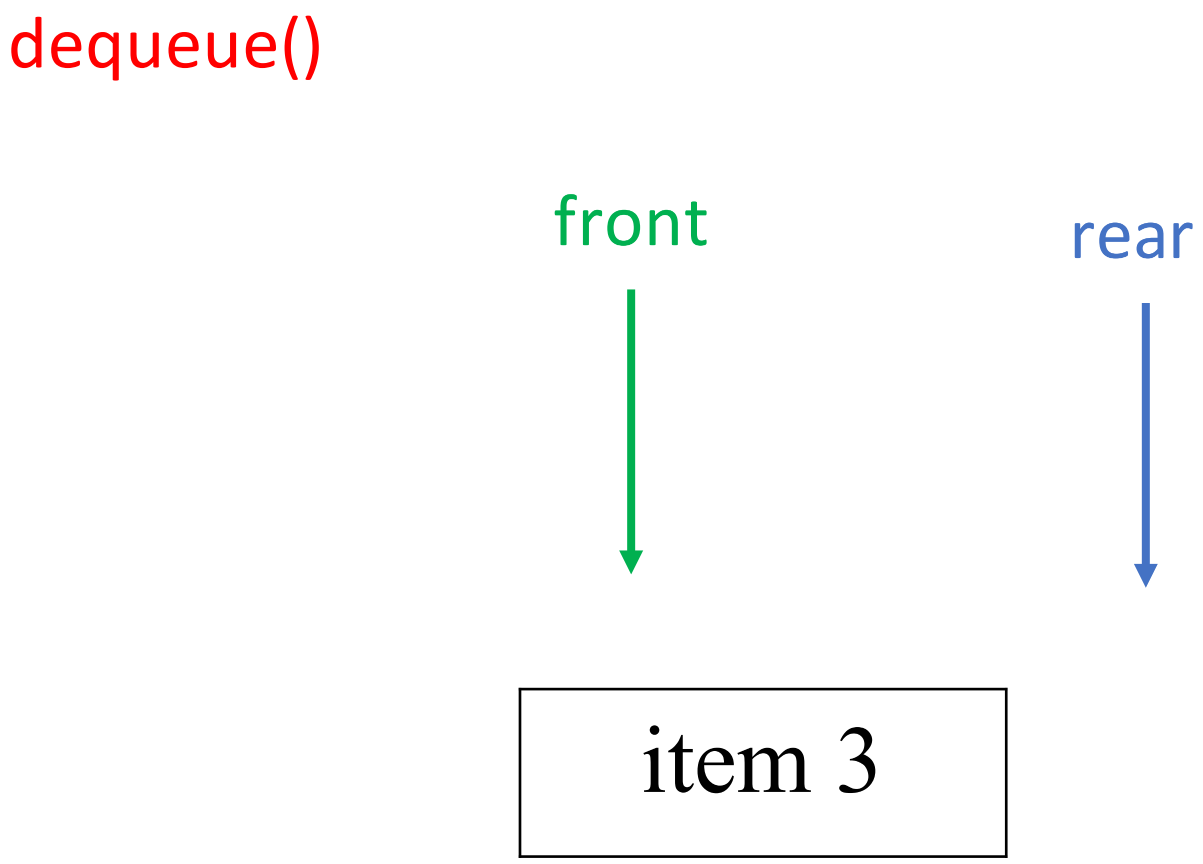


At this point we can’t remove item 2, or item 3

Queue

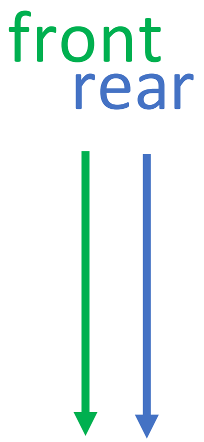


Queue



Queue

dequeue()



In an empty queue, front and rear point to the same position.

Queue ADT

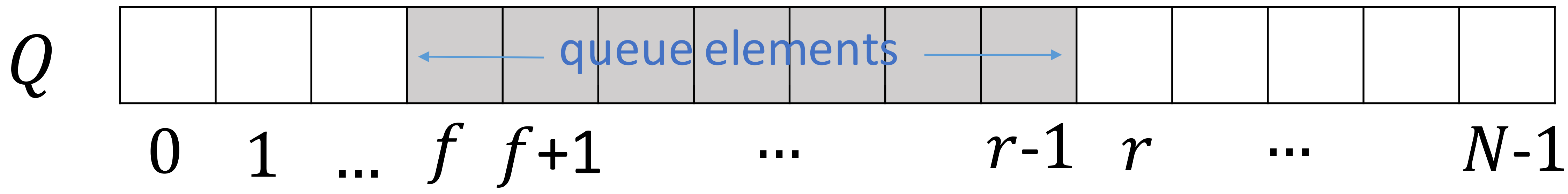
A queue supports the following operations:

- **ENQUEUE(x):** Insert item x at the rear of the queue.
- **DEQUEUE():** Remove **and return** the item at the front of the queue. An error occurs if the queue is empty.
- **ISEMPTY():** Return True if the queue is empty, False otherwise.
- **FRONT():** Return the front item in the queue **without removing** it; an error occurs if the queue is empty.
- **SIZE():** Return the number of items in the queue.

Array-based Implementation

- We can also use an array to implement a queue:

1. ***Array Q:*** N-element array, with elements stored from $Q[f]$ to $Q[r-1]$
2. ***f:*** pointer to the front position in Q
3. ***r:*** pointer to the rear position in Q, (next available position)
4. ***N***: specified max queue size

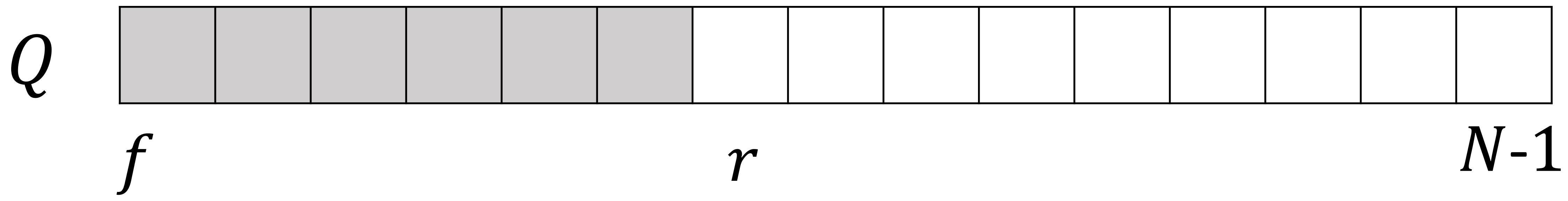


Simple Implementation

1. For **enqueue**, put the item at $Q[r]$, and increment r
2. For **dequeue**, remove from $Q[f]$, and increment f

- We say that queue is full when r reaches N.

- Example:

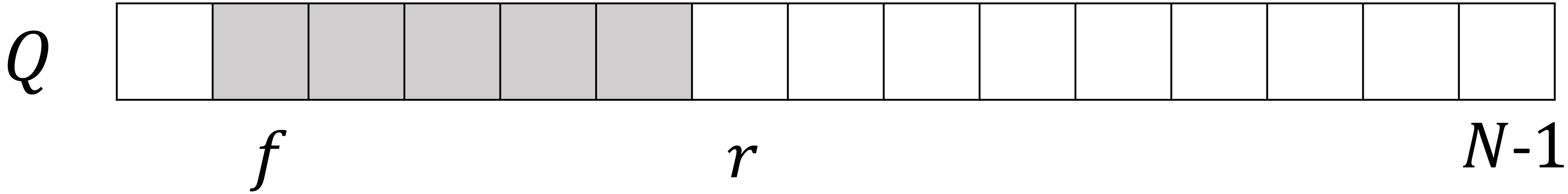


Simple Implementation

1. For **enqueue**, put the item at $Q[r]$, and increment r
2. For **dequeue**, remove from $Q[f]$, and increment f

- We say that queue is full when r reaches N.

- Example: dequeue

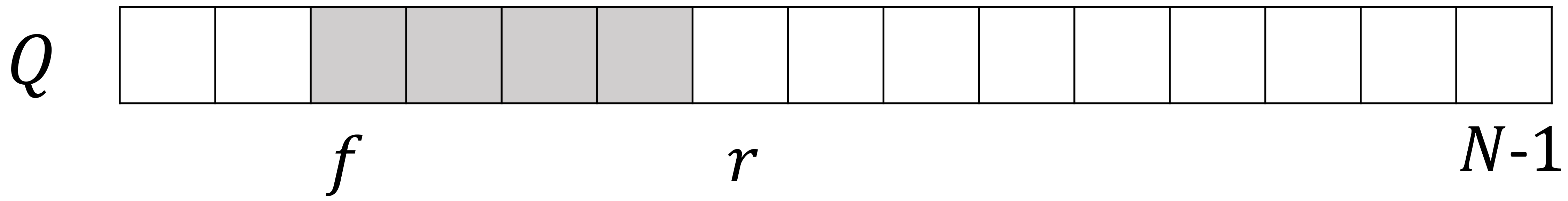


Simple Implementation

1. For **enqueue**, put the item at $Q[r]$, and increment r
2. For **dequeue**, remove from $Q[f]$, and increment f

- We say that queue is full when r reaches N.

- Example: dequeue

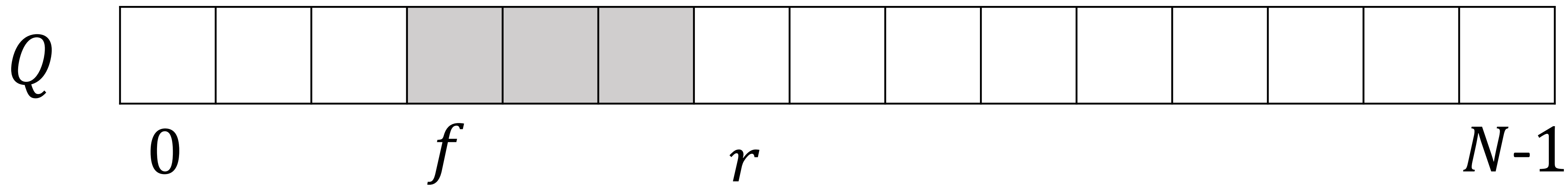


Simple Implementation

1. For **enqueue**, put the item at $Q[r]$, and increment r
2. For **dequeue**, remove from $Q[f]$, and increment f

- We say that queue is full when r reaches N.

- Example: dequeue

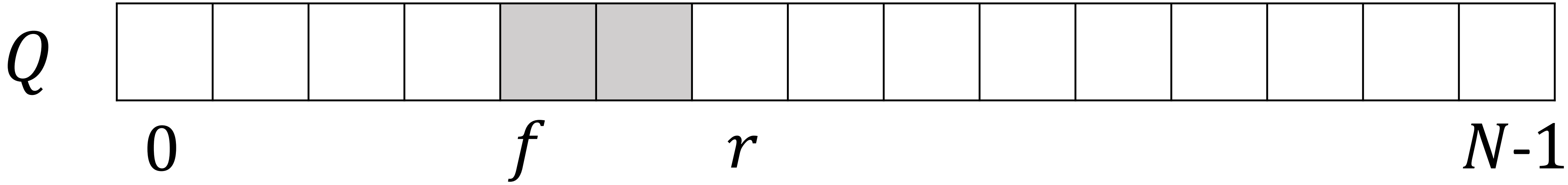


Simple Implementation

1. For **enqueue**, put the item at $Q[r]$, and increment r
2. For **dequeue**, remove from $Q[f]$, and increment f

- We say that queue is full when r reaches N.

- Example: dequeue

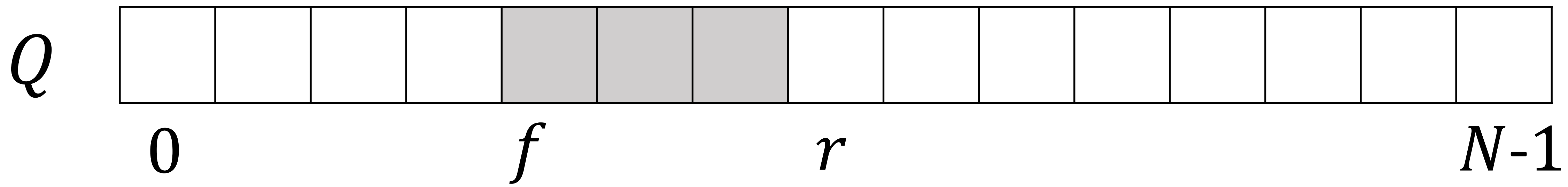


Simple Implementation

1. For **enqueue**, put the item at $Q[r]$, and increment r
2. For **dequeue**, remove from $Q[f]$, and increment f

- We say that queue is full when r reaches N.

- Example: enqueue

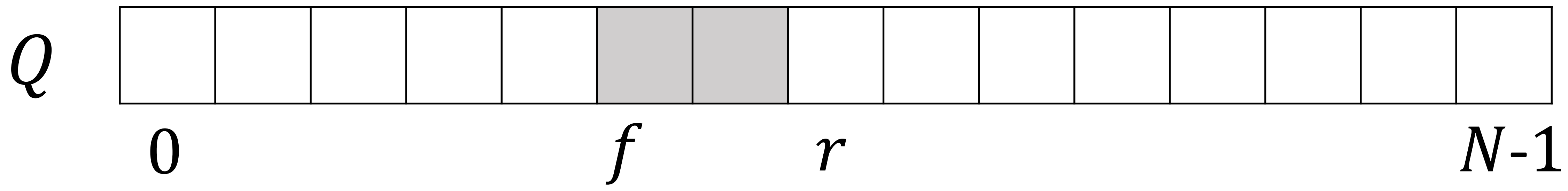


Simple Implementation

1. For **enqueue**, put the item at $Q[r]$, and increment r
2. For **dequeue**, remove from $Q[f]$, and increment f

- We say that queue is full when r reaches N.

- Example: dequeue

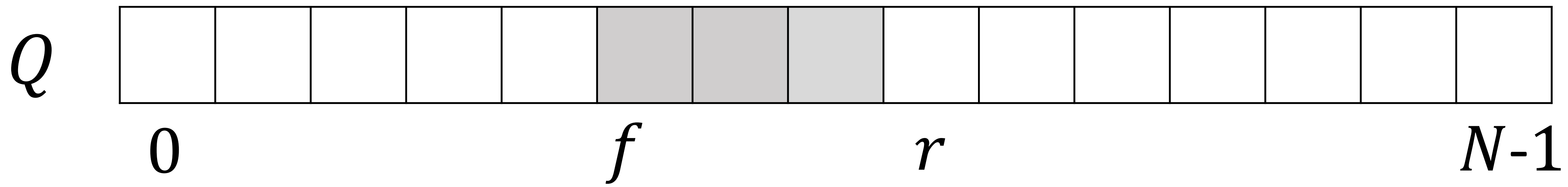


Simple Implementation

1. For **enqueue**, put the item at $Q[r]$, and increment r
2. For **dequeue**, remove from $Q[f]$, and increment f

- We say that queue is full when r reaches N.

- Example: enqueue

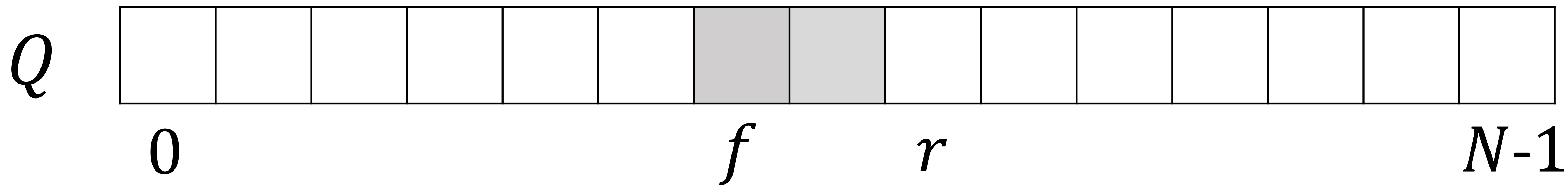


Simple Implementation

1. For **enqueue**, put the item at $Q[r]$, and increment r
2. For **dequeue**, remove from $Q[f]$, and increment f

- We say that queue is full when r reaches N.

- Example: dequeue

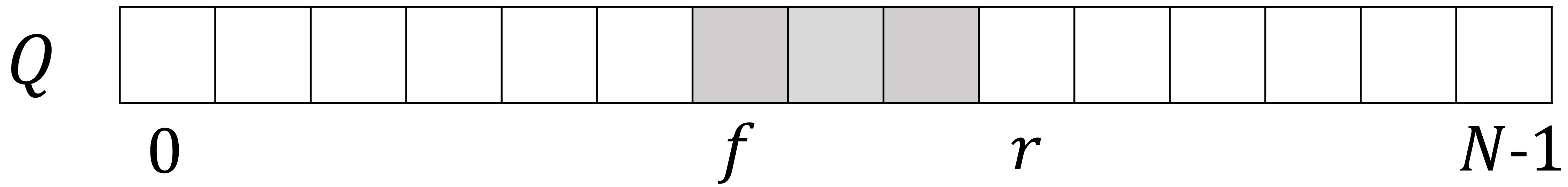


Simple Implementation

1. For **enqueue**, put the item at $Q[r]$, and increment r
2. For **dequeue**, remove from $Q[f]$, and increment f

- We say that queue is full when r reaches N.

- Example: enqueue

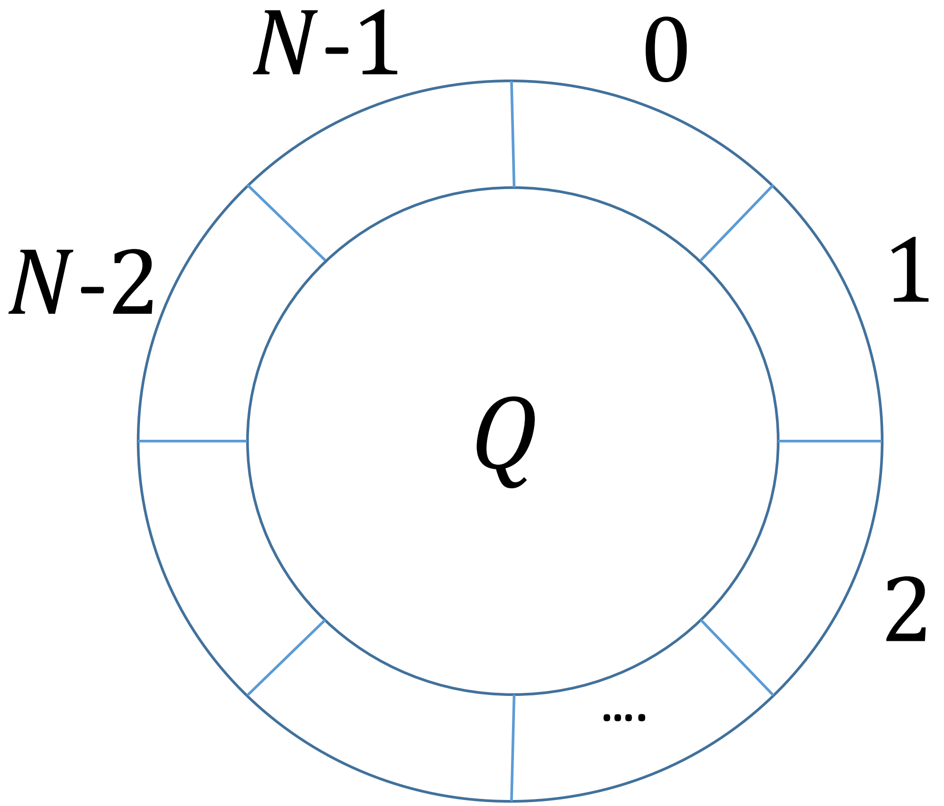


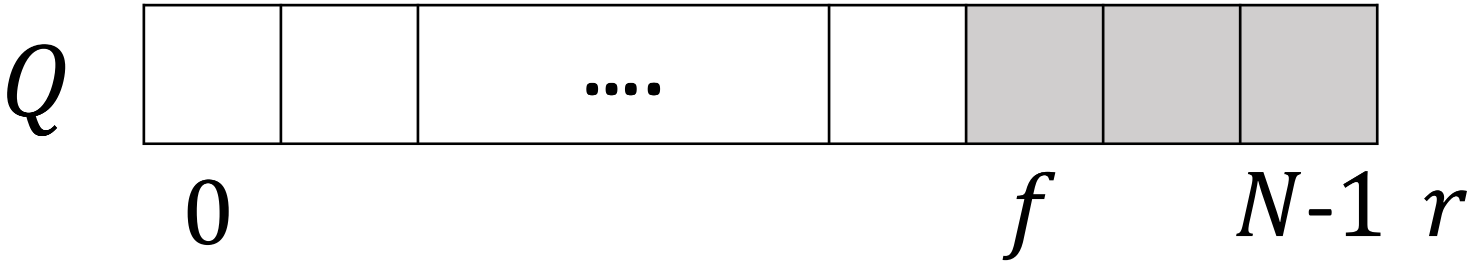
Simple Implementation

- However, this has a big problem!
- Say for example the size of the queue is at most 3 elements and we are doing a sequence of enqueues, and dequeues. Then, r could reach N while we still have plenty of space left in the array.

How to solve it?

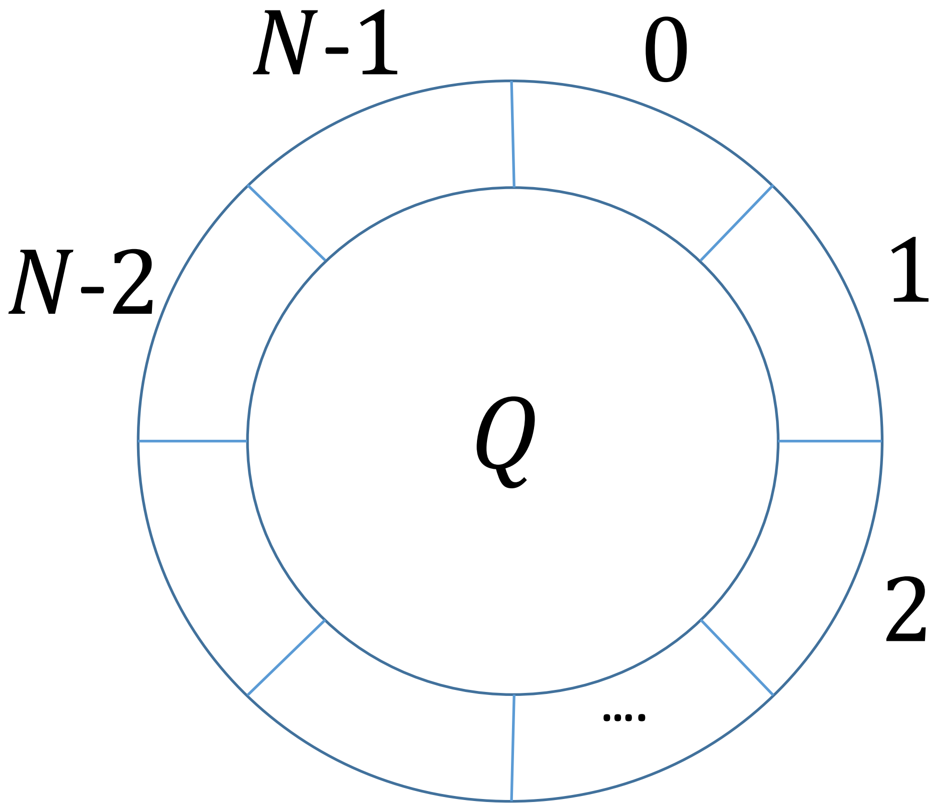
- The idea is to look at the array in a **circular way**.

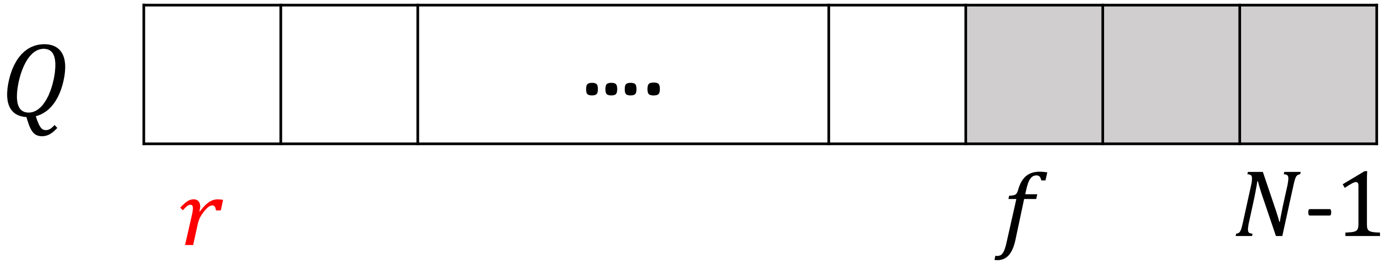




How to solve it?

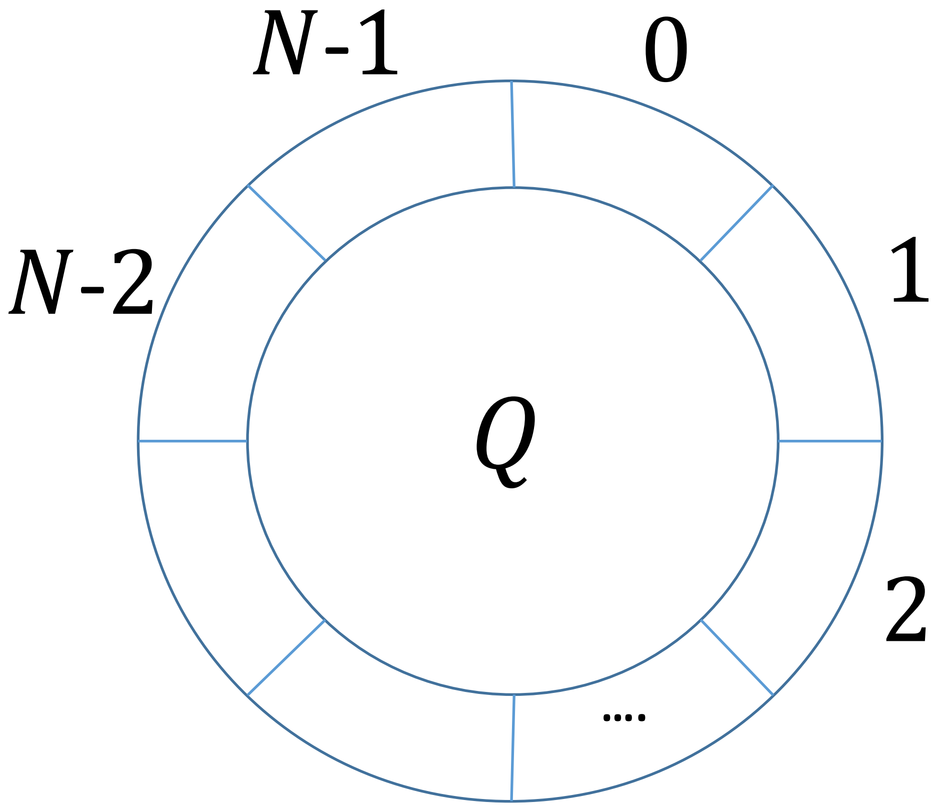
- The idea is to look at the array in a **circular way**.

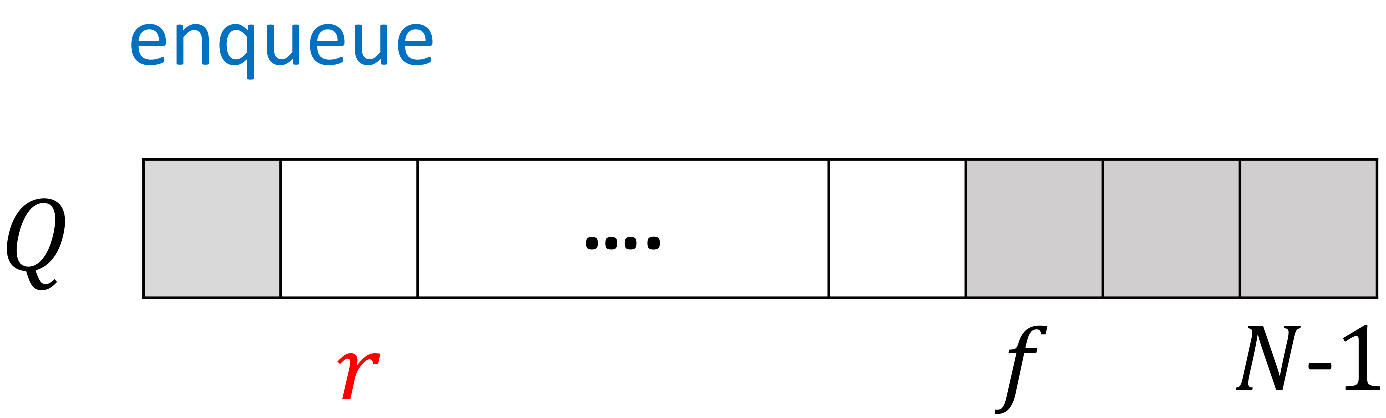




How to solve it?

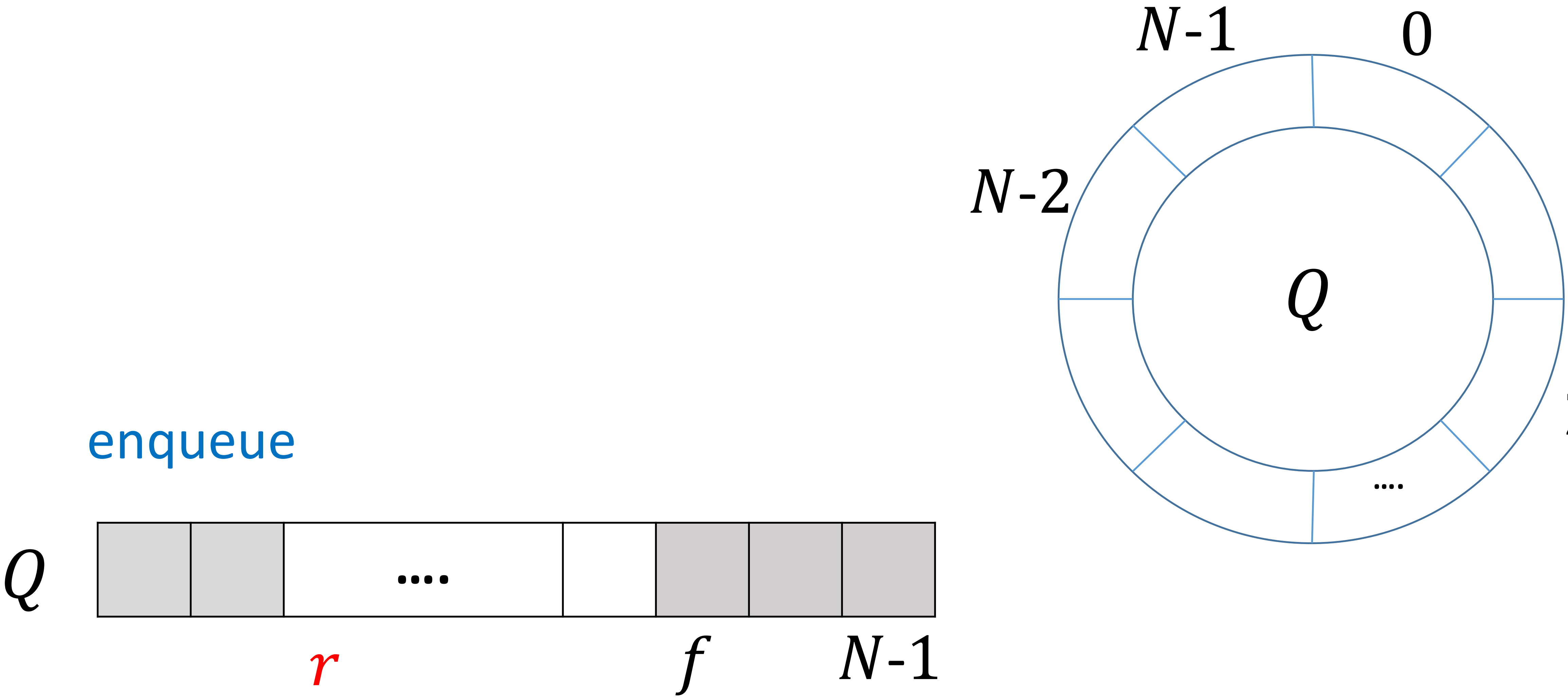
- The idea is to look at the array in a **circular way**





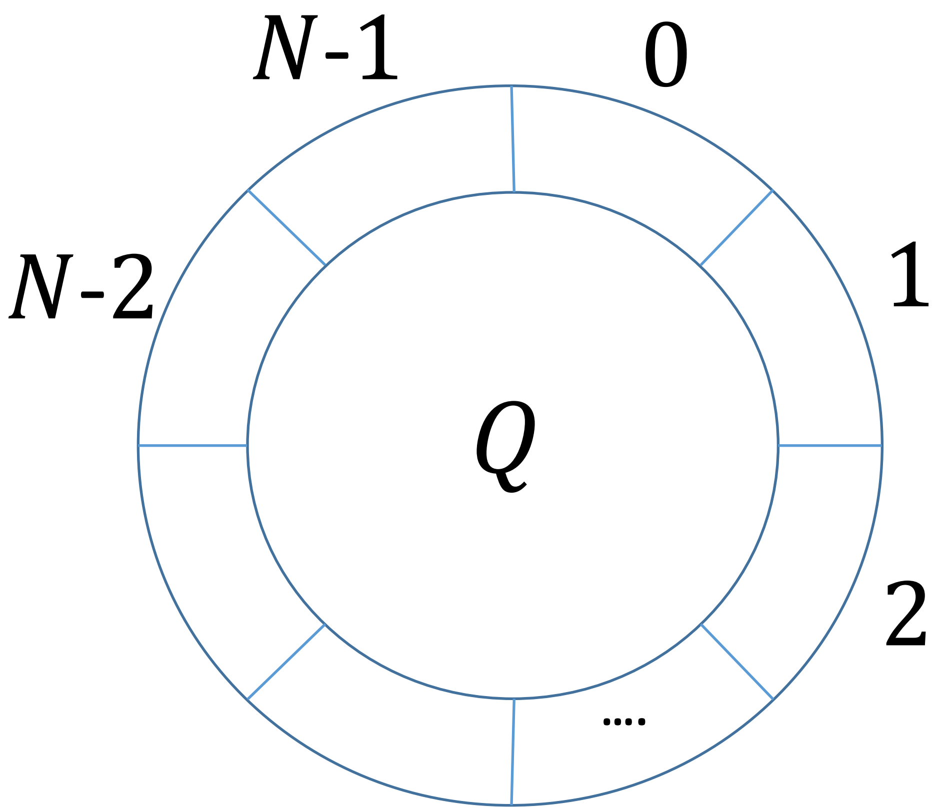
How to solve it?

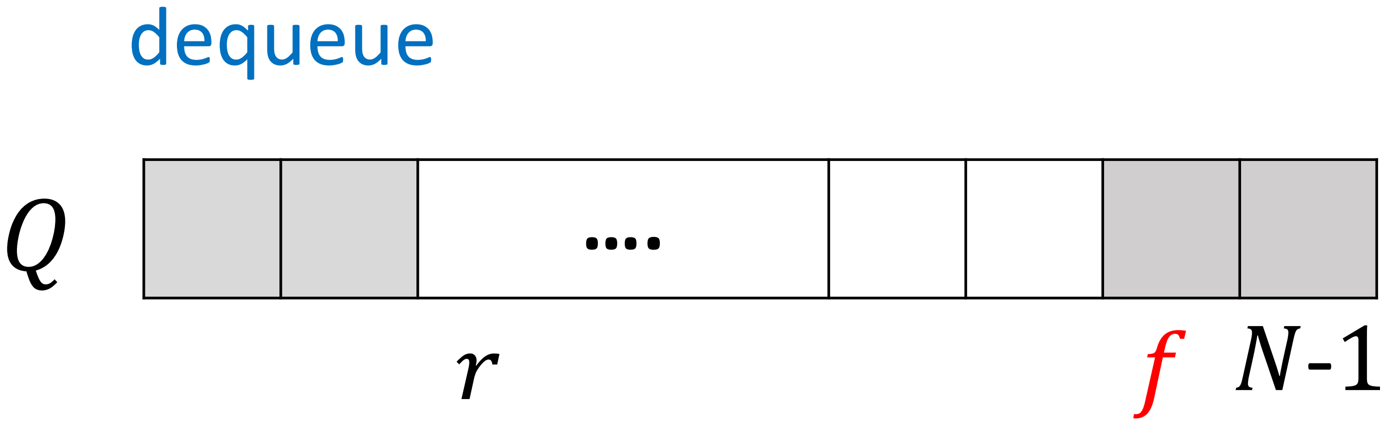
- The idea is to look at the array in a **circular way**.



How to solve it?

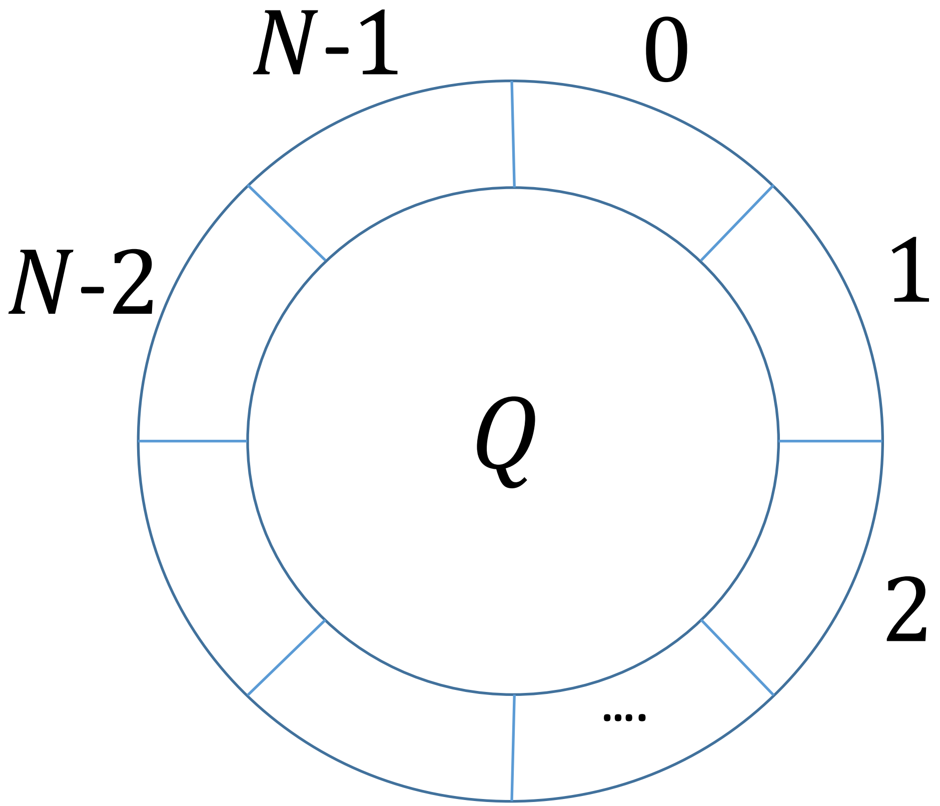
- The idea is to look at the array in a **circular way**.



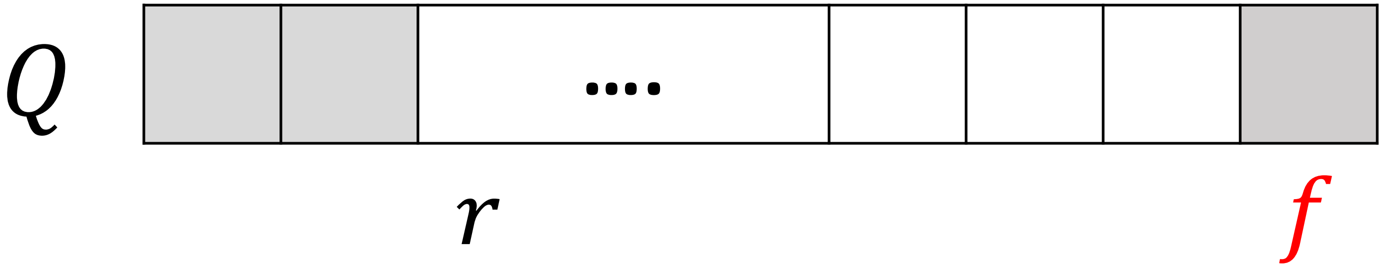


How to solve it?

- The idea is to look at the array in a **circular way**.

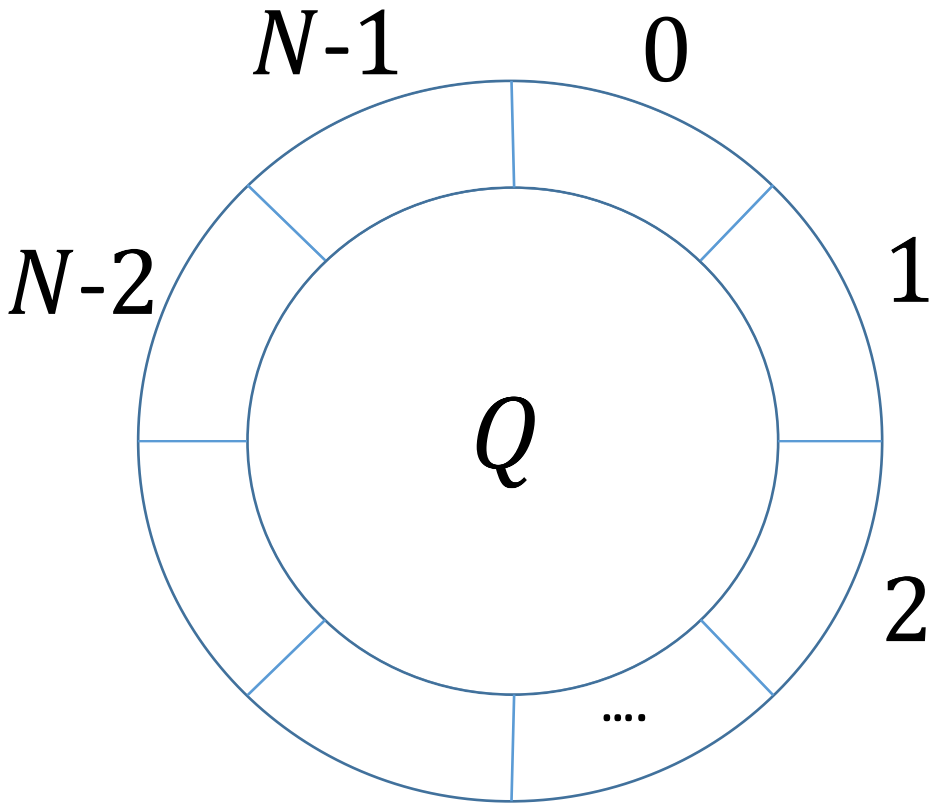


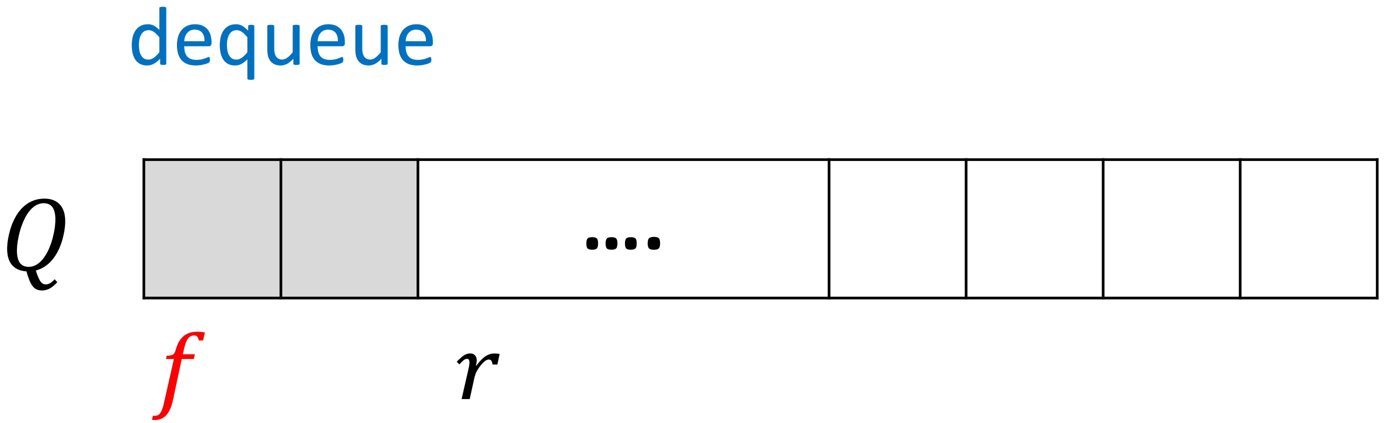
dequeue



How to solve it?

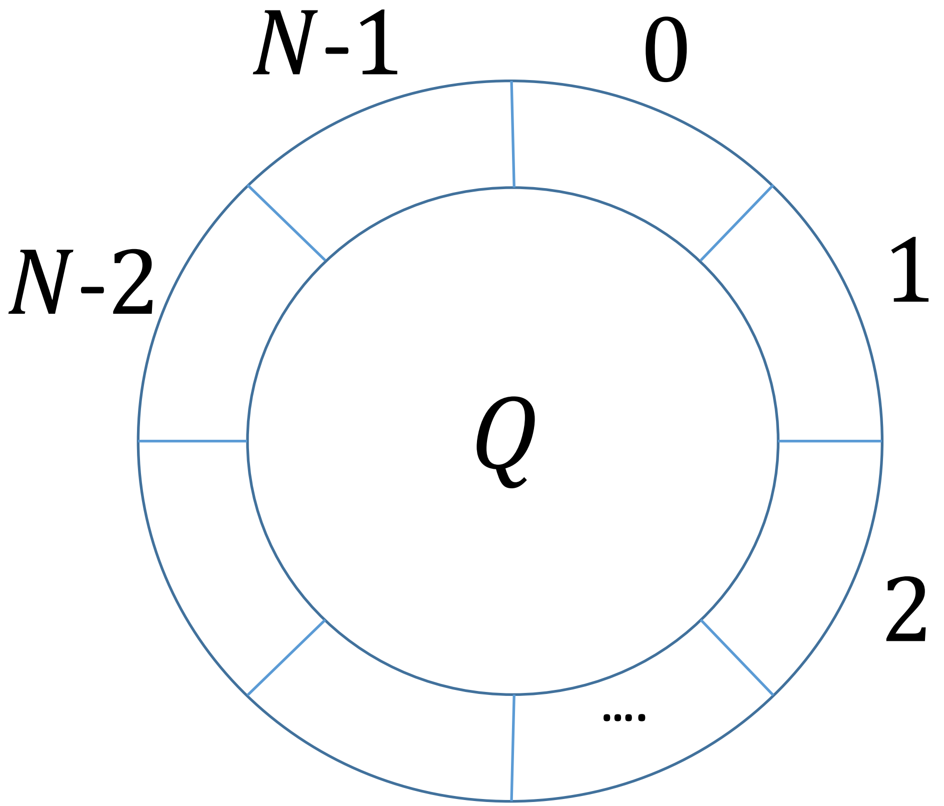
- As a result, when r or f reach N, they restart from 0, and we can use the full capacity of the array





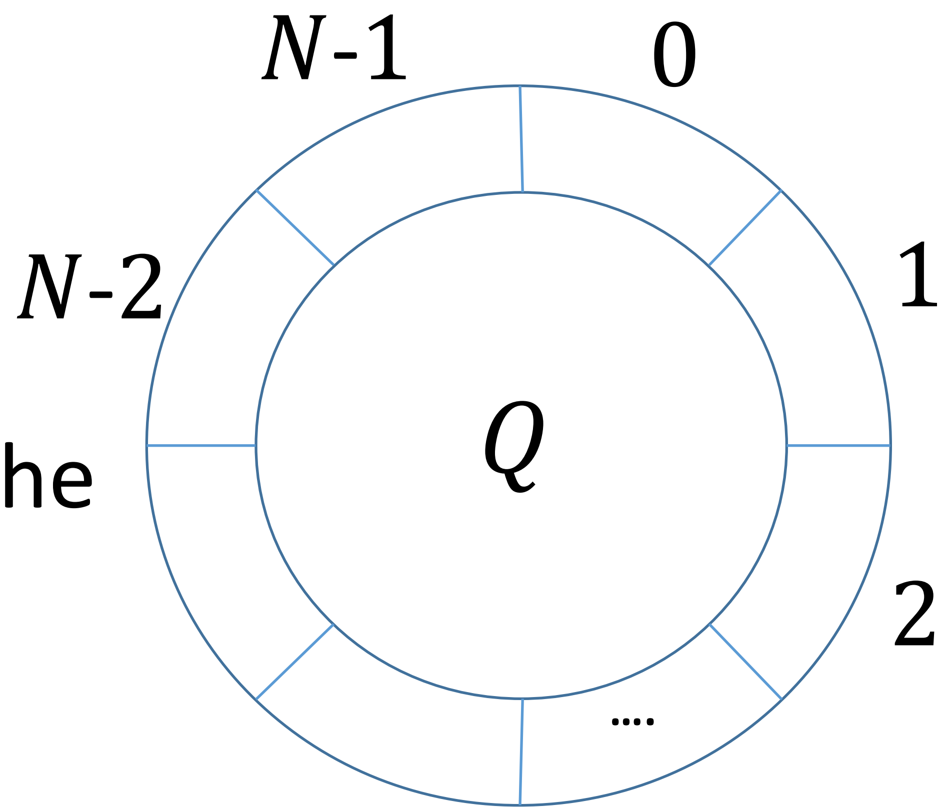
How to solve it?

- **Question:** What arithmetic operation can we apply to r and f for this purpose?



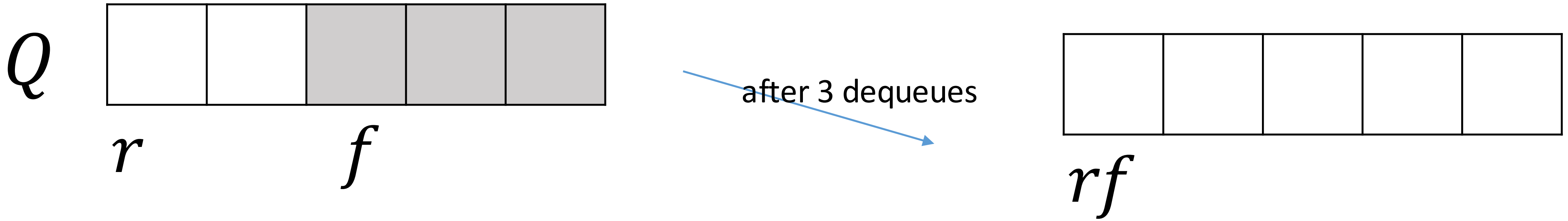
How to solve it?

- **Question:** What arithmetic operation can we apply to r and f for this purpose?
- **Answer:** Modulo. Instead of checking if r has reached N, we always say increment r, and take the result modulo N. The results is in $[0, N-1]$



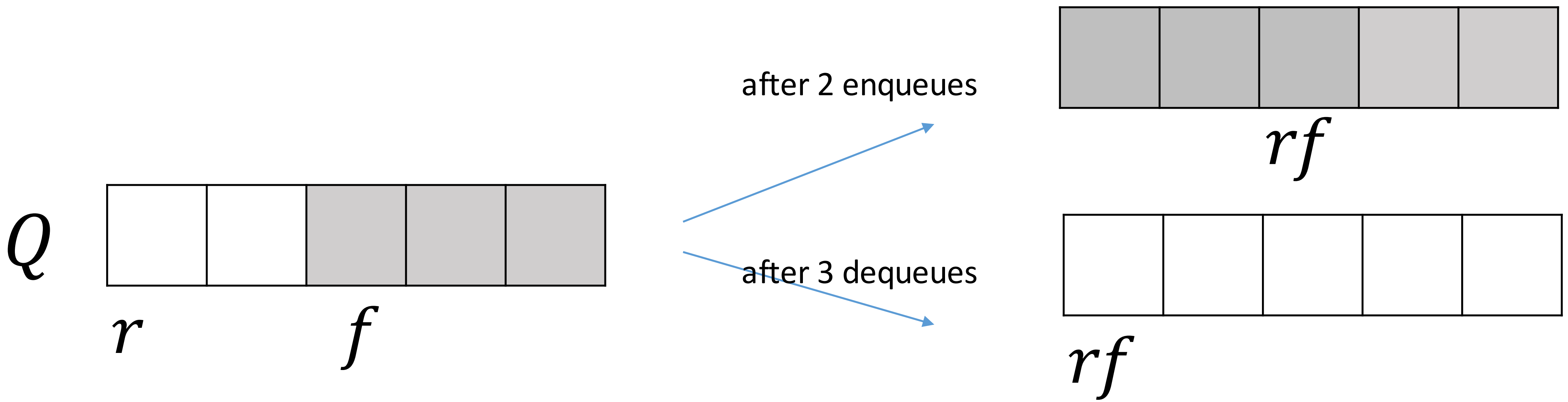
Problem with size

- **Question:** Before we said that $f == r$ means an empty queue. But what else could it mean in a circular implementation?



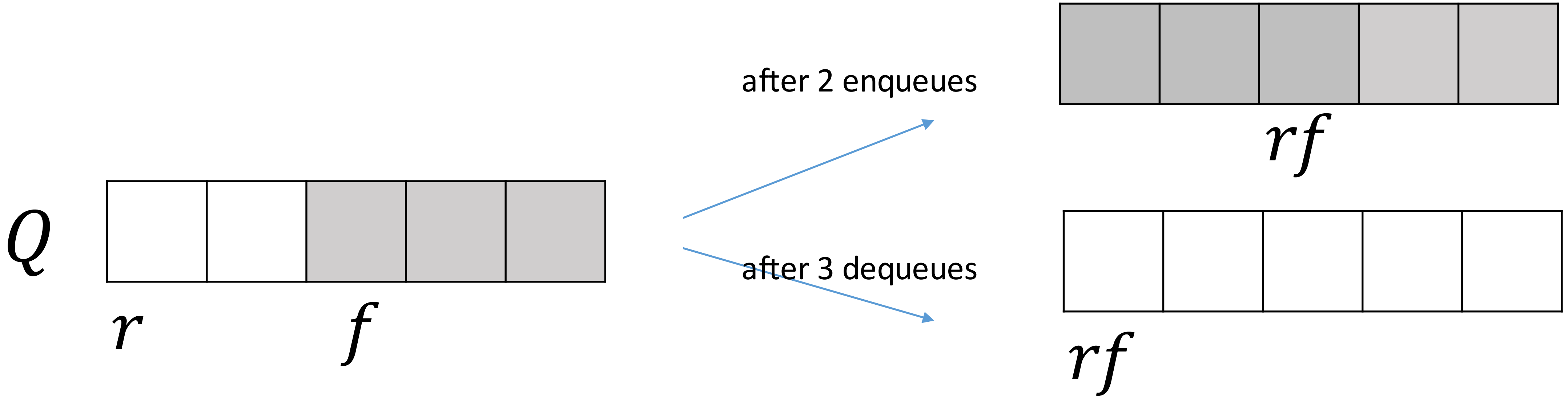
Problem with size

- **Answer:** Since we use the array in a circular way, whenever $f == r$, it could be that either the queue is **empty** or **full**!



Problem with size

- In order avoid problems like this we use a variable $size$. We **increment** $size$ upon an enqueue, and **decrement** it upon a dequeue.



Size (), IsEmpty (), and Front()

```
//We have a variable size which is
//updated in each operation. Initially,
//size is 0
Size()
    return size

Front()
    return Q[f]

IsEmpty()
    if Size() == 0
        return TRUE
    return FALSE
```

ENQUEUE(x), DEQUEUE()

```
ENQUEUE(x)
    if SIZE() == N
        error "queue is full"
        return
    Q[r] = x
    r = (r+1) mod N
    size = size + 1
```

```
DEQUEUE ()
    if ISEMPTY() == TRUE
        error "queue is empty"
        return NULL
    item = Q[f]
    f = (f +1) mod N
    size = size - 1
    return item
```