Stacks and Queues

Subhabrata Samajder

IIIT, Delhi
Winter Semester,
22nd March, 2023

Stacks and Queues

- Stacks and Queues are dynamic sets where the DELETE operation is prespecified.
  - Stack: Last-in, first-out (LIFO) or first-in, last-out (FILO).
  - Queue: First-in, first-out (FIFO) or last-in, last-out (LILO).

Stacks and Queues

- Stacks and Queues are dynamic sets where the DELETE operation is prespecified.
  - Stack: Last-in, first-out (LIFO) or first-in, last-out (FILO).
  - Queue: First-in, first-out (FIFO) or last-in, last-out (LILO).
- There are several efficient ways to implement stacks and queues.
- Here we will use arrays.

Stack

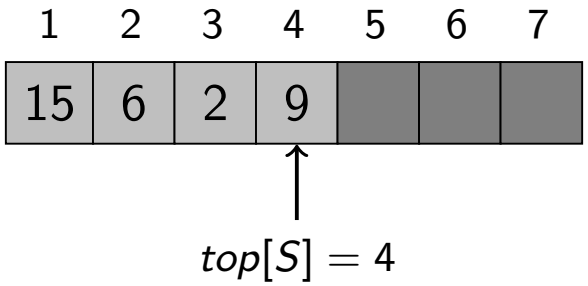
Stack

- The INSERT is called PUSH.
- The DELETE operation is called POP.
  - **Note:** POP does not take an element as argument,

Stack

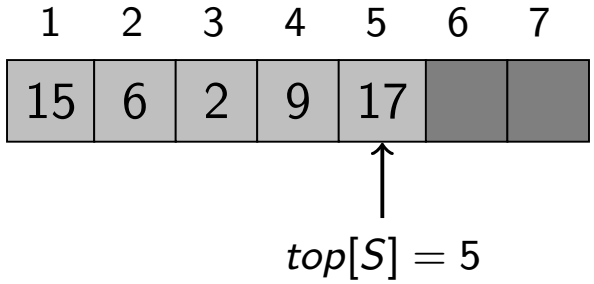
- The INSERT is called PUSH.
- The DELETE operation is called POP.
  - **Note:** POP does not take an element as argument,
- An array $S[1 \ldots n]$ denotes a stack of at most n elements.
- $top[S]$: Points to the most recently inserted element.
- The stack consists of elements $S[1 \ldots top[S]]$, where
  - $S[1]$ is the element at the bottom of the stack and
  - $S[top[S]]$ is the element at the top.
- **Empty Stack:** $top[S] = 0$.
- **Stack Underflow:** Empty stack is popped.
- **Stack Overflow:** $top[S] > n$.

Stack: An Example



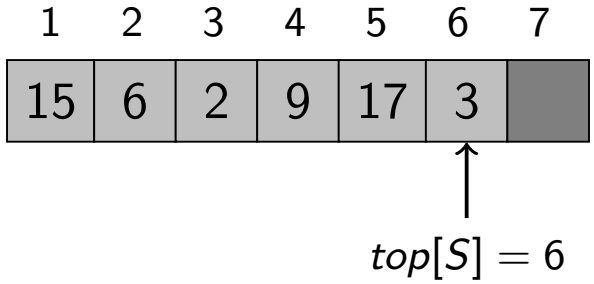
Stack: An Example

PUSH$(S, 17)$:



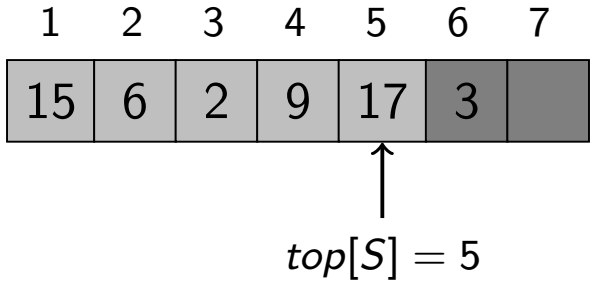
Stack: An Example

PUSH$(S, 3)$:



Stack: An Example

POP(S):



PUSH and POP

```
STACK-EMPTY(S)
Begin
  If (top[S] = 0)
    return TRUE;
  Else
    return FALSE;
End
```

Complexity: $\mathcal{O}(1)$

PUSH and POP

```
STACK-EMPTY(S)
Begin
  If (top[S] = 0)
    return TRUE;
  Else
    return FALSE;
End
```

Complexity: $\mathcal{O}(1)$

```
PUSH(S, x)
Begin
  If (top[S] = n)
    error "overflow";
  Else
    top[S] ← top[S] + 1;
    S[top[S]] ← x;
End
```

Complexity: $\mathcal{O}(1)$

PUSH and POP

```
STACK-EMPTY(S)
Begin
  If (top[S] = 0)
    return TRUE;
  Else
    return FALSE;
End
```

Complexity: $\mathcal{O}(1)$

```
PUSH(S, x)
Begin
  If (top[S] = n)
    error "overflow";
  Else
    top[S] ← top[S] + 1;
    S[top[S]] ← x;
End
```

Complexity: $\mathcal{O}(1)$

```
POP(S)
Begin
  If (STACKEMPTY(S))
    error "underflow";
  Else
    top[S] ← top[S] − 1;
    return S[top[S] + 1];
End
```

Complexity: $\mathcal{O}(1)$

Prefix, Postfix and Infix Expressions

Prefix, Postfix and Infix Expressions

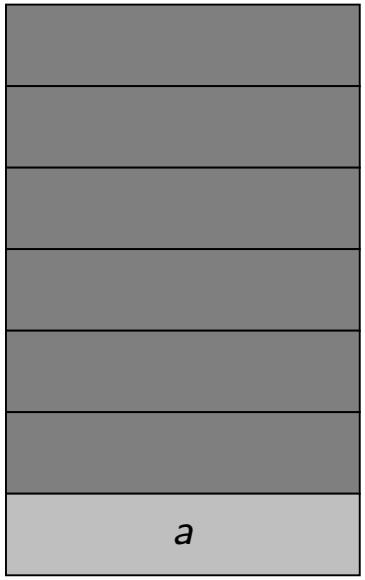
- **Infix:** The operators in the expression are placed in between the operands on which the operator works.
  - **Example:** $a + b * c$
  - Easy to read, write and understand by humans.
  - But not by computer.
  - It's costly, in terms of time and space, to process Infix expressions
- **Postfix:** The operators are placed after the operands on which the operator works.
  - **Example:** $abc * +$
  - It's most used to notation for evaluating arithmetic expression.
- **Prefix:** The operators are placed before the operands on which the operator works.
  - **Example:** $+a * bc$

Postfix To Prefix: An Example

I/P: $a \ b \ c \ / \ - \ a \ d \ / \ e \ - \ *$

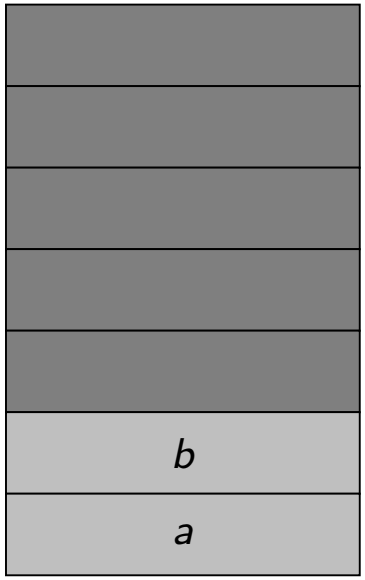
Postfix To Prefix: An Example

I/P: $a\ b\ c\ /\ -\ a\ d\ /\ e\ -\ *$



Postfix To Prefix: An Example

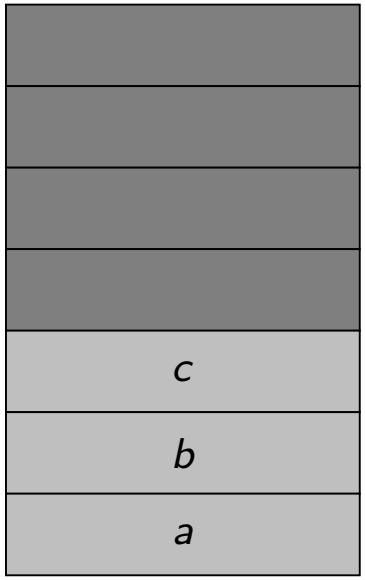
I/P: $a \quad \underset{\uparrow}{b} \quad c \quad / \quad - \quad a \quad d \quad / \quad e \quad - \quad *$



Postfix To Prefix: An Example

I/P: $a\ b\ c\ /\ -\ a\ d\ /\ e\ -\ *$

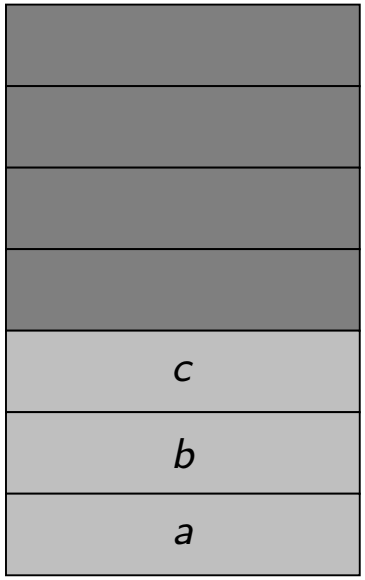
$\uparrow$



Postfix To Prefix: An Example

I/P: $a \quad b \quad c \quad / \quad - \quad a \quad d \quad / \quad e \quad - \quad *$

$\uparrow$

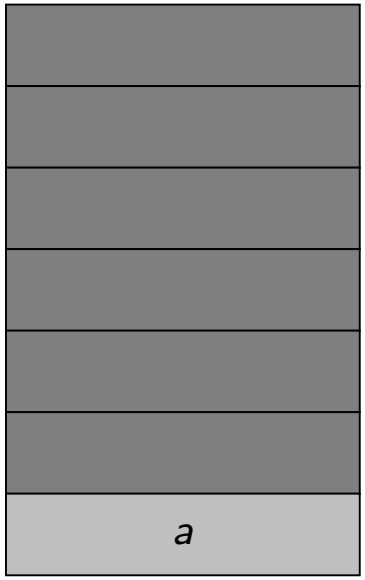


Postfix To Prefix: An Example

I/P: $a \quad b \quad c \quad / \quad - \quad a \quad d \quad / \quad e \quad - \quad *$

↑

POP two
from the stack

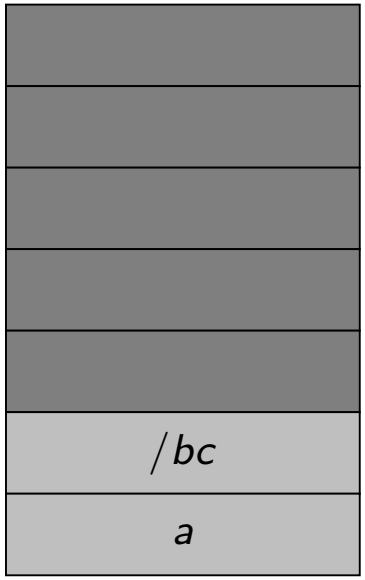


Postfix To Prefix: An Example

I/P: $a \quad b \quad c \quad / \quad - \quad a \quad d \quad / \quad e \quad - \quad *$

$\uparrow$

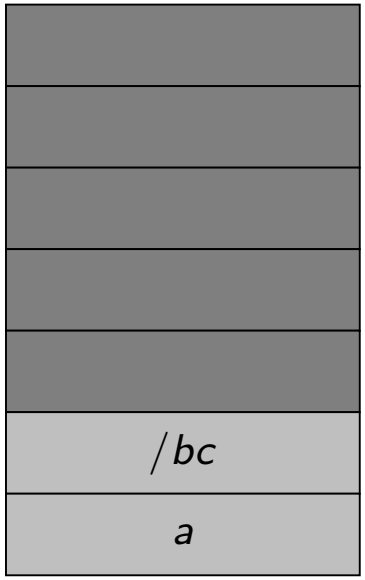
PUSH "/bc"



Postfix To Prefix: An Example

I/P: $a \quad b \quad c \quad / \quad - \quad a \quad d \quad / \quad e \quad - \quad *$

$\uparrow$

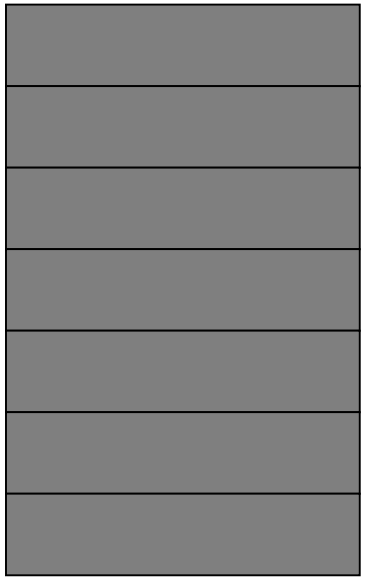


Postfix To Prefix: An Example

I/P: $a \; b \; c \; / \; - \; a \; d \; / \; e \; - \; *$

↑

POP two
from the stack

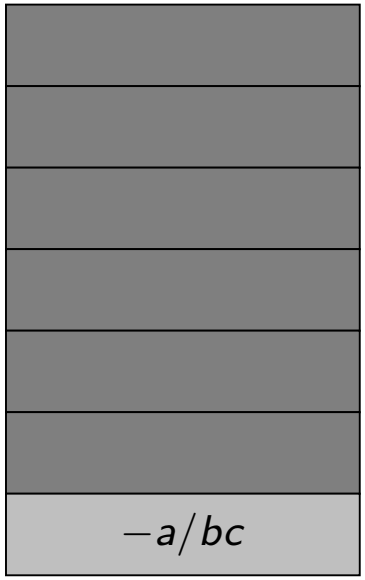


Postfix To Prefix: An Example

I/P: $a \quad b \quad c \quad / \quad - \quad a \quad d \quad / \quad e \quad - \quad *$

$\uparrow$

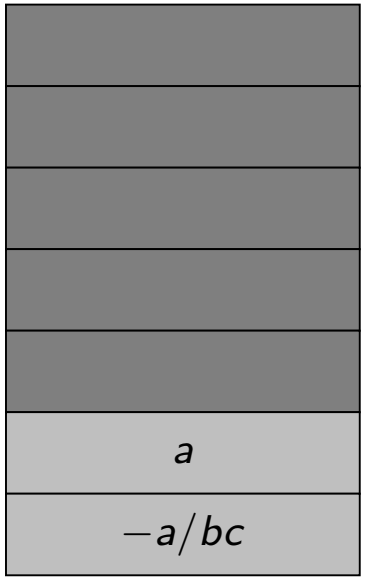
PUSH "-a/bc"



Postfix To Prefix: An Example

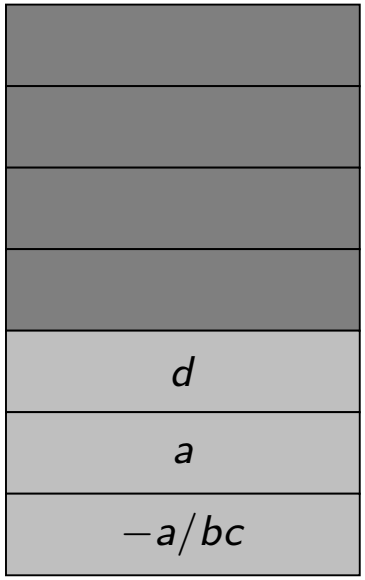
I/P: $a \quad b \quad c \quad / \quad - \quad a \quad d \quad / \quad e \quad - \quad *$

$\uparrow$



Postfix To Prefix: An Example

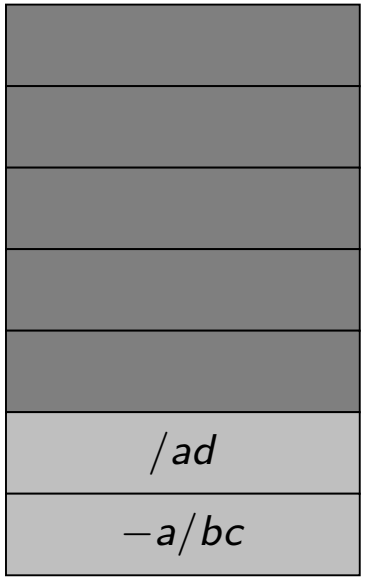
I/P: $a \quad b \quad c \quad / \quad - \quad a \quad d \quad / \quad e \quad - \quad *$



Postfix To Prefix: An Example

I/P: $a \quad b \quad c \quad / \quad - \quad a \quad d \quad / \quad e \quad - \quad *$

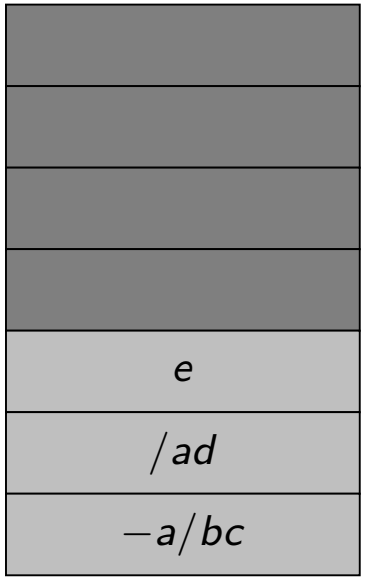
$\uparrow$



Postfix To Prefix: An Example

I/P: $a \quad b \quad c \quad / \quad - \quad a \quad d \quad / \quad e \quad - \quad *$

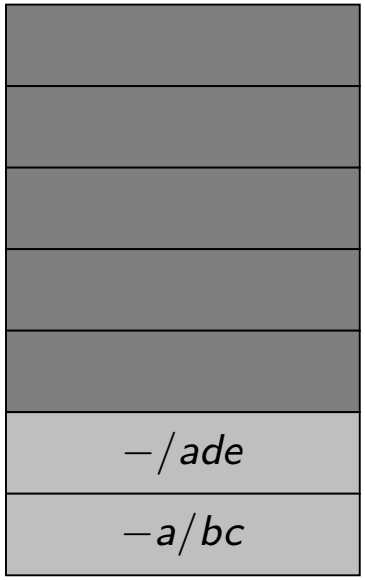
$\uparrow$



Postfix To Prefix: An Example

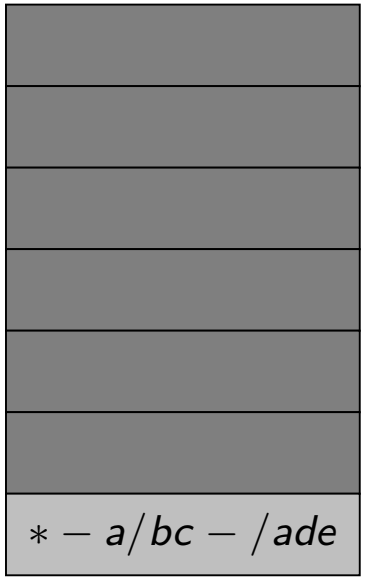
I/P: $a \quad b \quad c \quad / \quad - \quad a \quad d \quad / \quad e \quad - \quad *$

$\uparrow$



Postfix To Prefix: An Example

I/P: $a \;\; b \;\; c \;\; / \;\; - \;\; a \;\; d \;\; / \;\; e \;\; - \;\; *$
$\uparrow$



Queue

Queue

Queue

Queue

Queue

Queue

Queue ADT

- Queues store arbitrary objects.
- **Insertions:** At the end of the queue.
- **Removals:** From the front of the queue.
- The queue has a **head** and a **tail**.

Queue ADT (Cont.)

Main Operations:

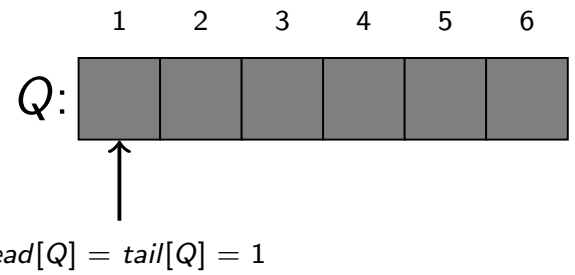
- ENQUEUE(Q, x): Inserts an element at the end.
- DEQUEUE(Q): Removes and returns the element at the front.

Queue ADT (Cont.)

Auxiliary Operations:

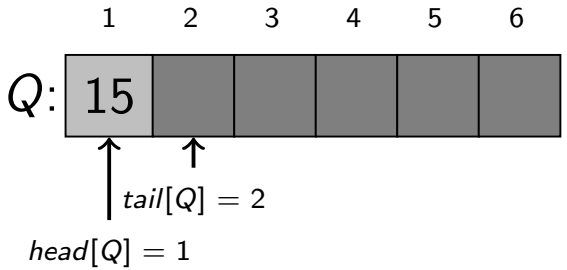
- FRONT(): Returns the element at the front without removing it.
- SIZE(): Returns the number of elements stored.
- ISEMPTY(): Returns a boolean value indicating if the queue is empty or not.

An Integer Array Implementation



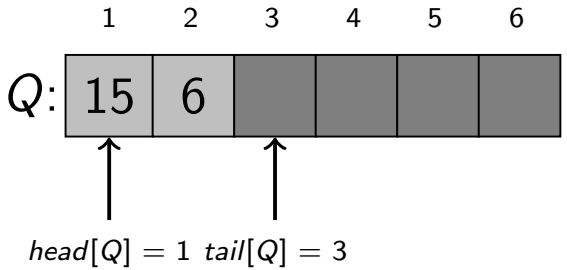
An Integer Array Implementation

ENQUEUE$(Q, 15)$:



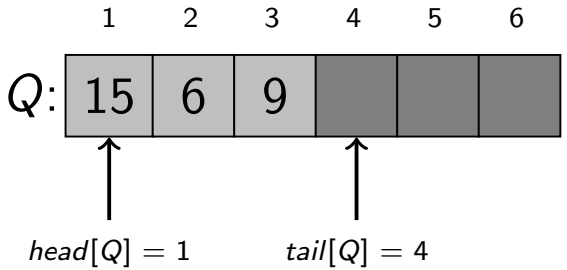
An Integer Array Implementation

ENQUEUE$(Q, 6)$:



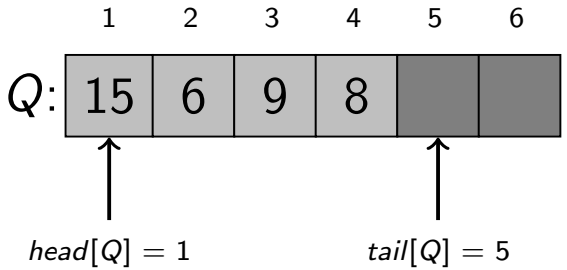
An Integer Array Implementation

ENQUEUE$(Q, 9)$:



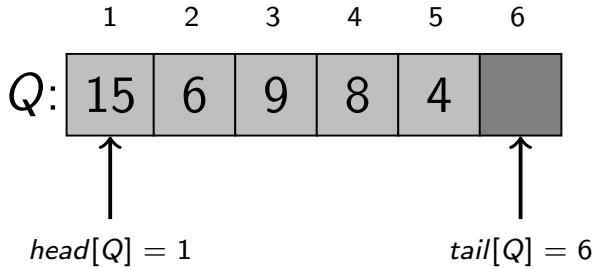
An Integer Array Implementation

ENQUEUE$(Q, 8)$:



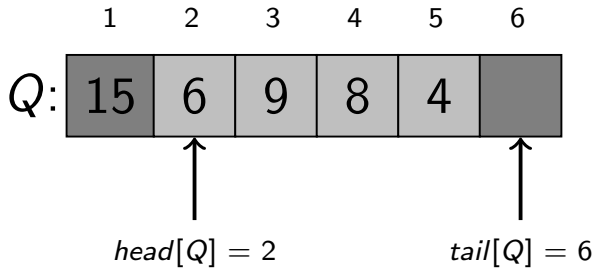
An Integer Array Implementation

ENQUEUE$(Q, 4)$:



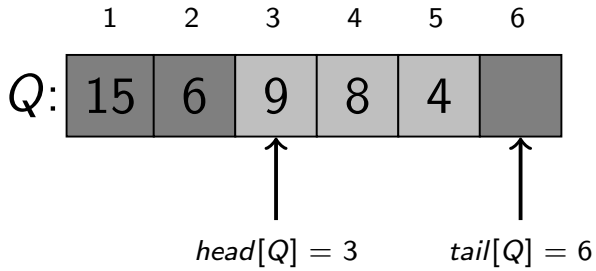
An Integer Array Implementation

DEQUEUE(Q):



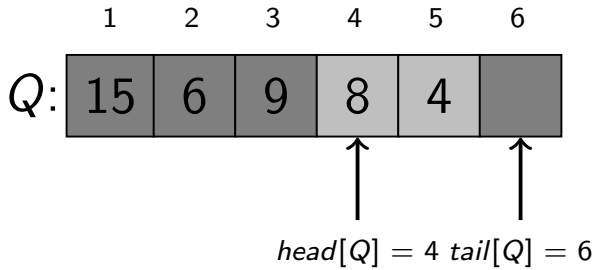
An Integer Array Implementation

DEQUEUE(Q):



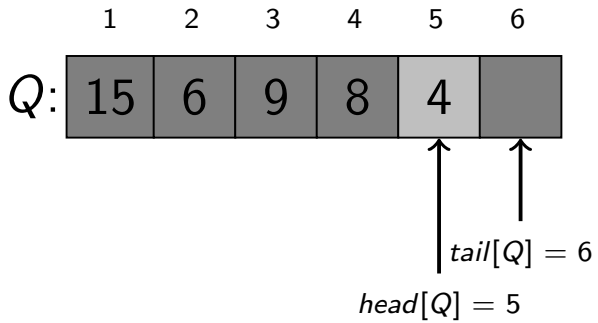
An Integer Array Implementation

DEQUEUE(Q):



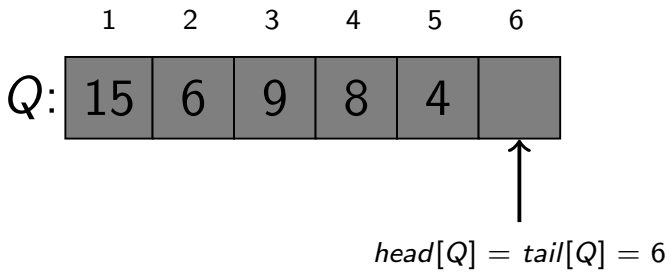
An Integer Array Implementation

DEQUEUE(Q):



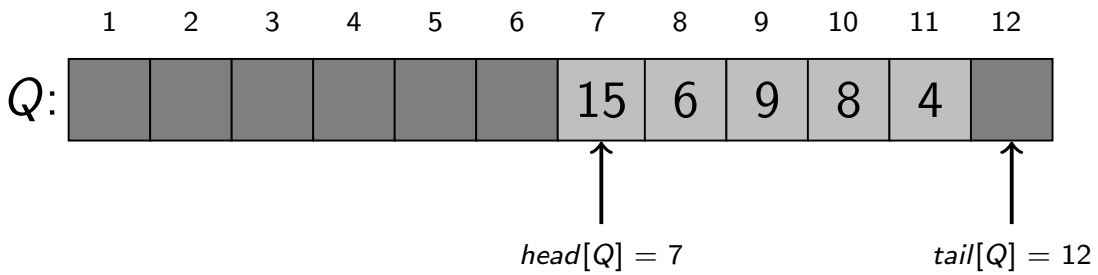
An Integer Array Implementation

DEQUEUE(Q):



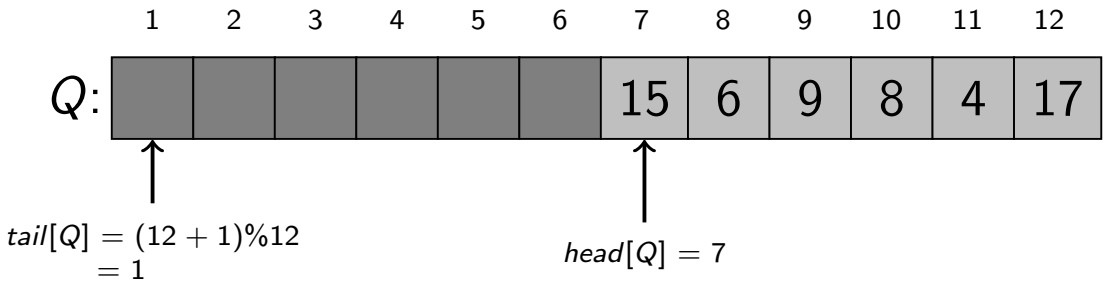
Queue Empty!!

An Improvement



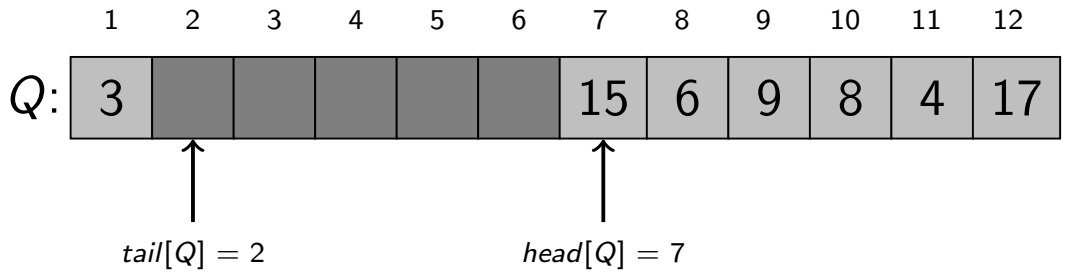
An Improvement

ENQUEUE$(Q, 17)$:



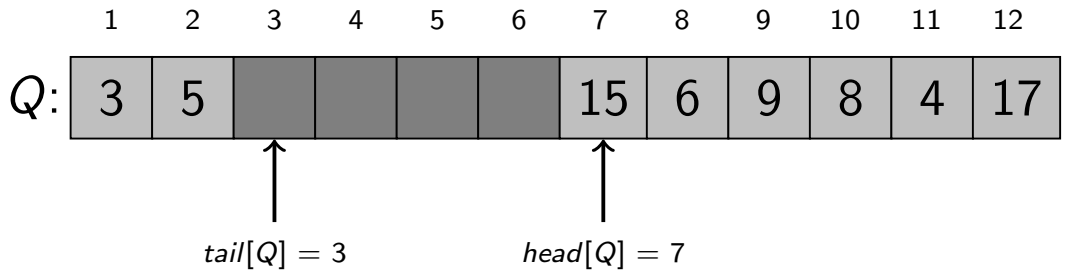
An Improvement

ENQUEUE$(Q, 3)$:



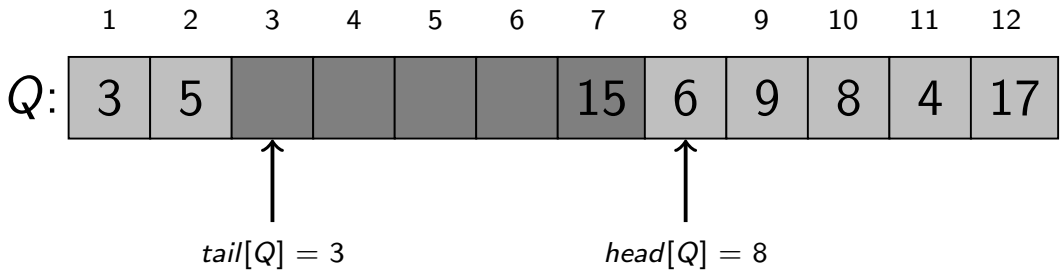
An Improvement

ENQUEUE$(Q, 5)$:



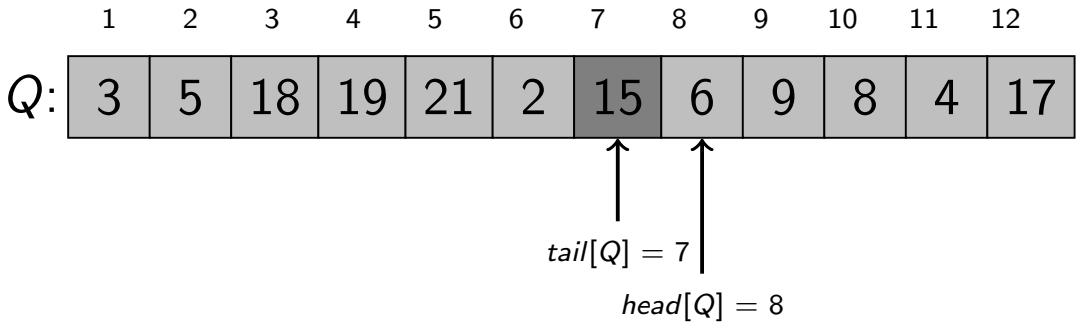
An Improvement

DEQUEUE(Q):



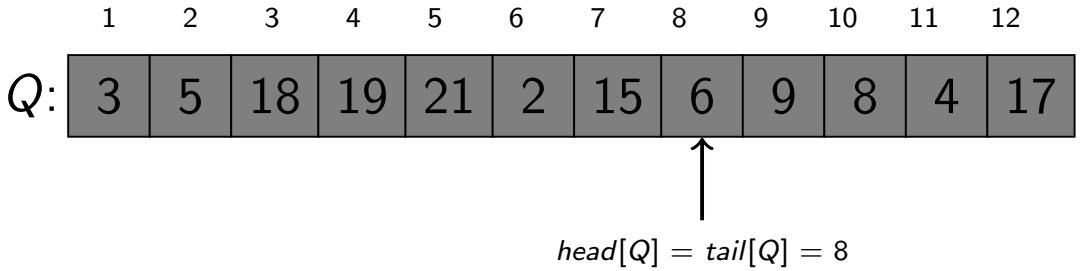
An Improvement

Queue Full: $head[Q] = (tail[Q] + 1) \mod n$.
Error: Overflow.



An Improvement

Queue Empty: $head[Q] = tail[Q]$.
Error: Underflow.



ENQUEUE(Q, x)

Begin
  If $(head[Q] = (tail[Q] + 1) \mod n)$
    return "**overflow error**"

  $Q[tail[Q]] \leftarrow x$;

  If $(tail[Q] = length[Q])$
    $tail[Q] \leftarrow 1$;
  Else
    $tail[Q] \leftarrow tail[Q] + 1$;

End

DEQUEUE(Q)

```
Begin
  If (head[Q] = tail[Q])
    return "underflow error"

  x ← Q[head[Q]];

  If (tail[Q] = length[Q])
    head[Q] ← 1;
  Else
    head[Q] ← head[Q] + 1;

  return x;
End
```

Applications of Queues

- Access to shared resources (e.g., printer).
- Simulations of read world situations of waiting lines (bank teller, flight bookings).
- To efficiently maintain a First-in-first out (FIFO) order on some entities
- In a multitasking operating system, the CPU cannot run all jobs at once, so jobs must be batched up and then scheduled according to order in a queue.
- User input in a game

A C Implemention of a Queue Using An Array

Initialization

```
/* Queue */
int main() {
   int head, tail;
   int Q[len];

   /* Initialisation */
   head = tail = 0;
   .
   .
}
```

Enqueue

Insert an element at the tail of the queue Q and redefine tail:

```
/* Enqueue */
int Enqueue(int data, int *Q) {
   /* check if queue is full or not */
   if (head == (tail + 1)% length) {
      printf("\n ERROR: Queue is full\n");
      return FLAG;
   }
   /* insert element at the tail */
   else {
      Q[tail] = data;
      tail = (tail + 1)% length;
   }
   return 0;
}
```

Dequeue

Delete and return the element pointed by head of the queue:

```
/* Dequeue */
int Dequeue(int *Q) {
   int x;

   if (head == tail) {    // if queue is empty
      printf("\n ERROR: Queue is empty\n");
      return FLAG;
      }
   /* delete element from the head */
   else {
      x = Q[head];
      head = (head + 1)% length;
   }
   return x;
}
```

Front

Return the front element from the queue (if queue is not empty) but do not remove it.

```
/* prints the head of the queue */
void Front() {
  if (head == tail) {
    printf("\n Q is Empty\n");
    return FLAG;
  }

  printf("\n Front Element is: %d", Q[head]);
  return 0;
}
```

Exercise

Describe the output and final structure of the queue after the following operations:

- ENQUEUE(8)
- ENQUEUE(3)
- DEQUEUE()
- ENQUEUE(2)
- ENQUEUE(5)
- DEQUEUE()
- DEQUEUE()
- ENQUEUE(9)
- ENQUEUE(1)

Books Consulted

1. Chapter 10.1 & 10.2 of *Introduction to Algorithms* by Thomas H Cormen, Charles E Leiserson, Ronald L Rivest, Clifford Stein.

Thank You for your kind attention!

Questions!!