Chapter 3
Stacks and Queues

By: Zahra Bahramian

Stacks and Queues

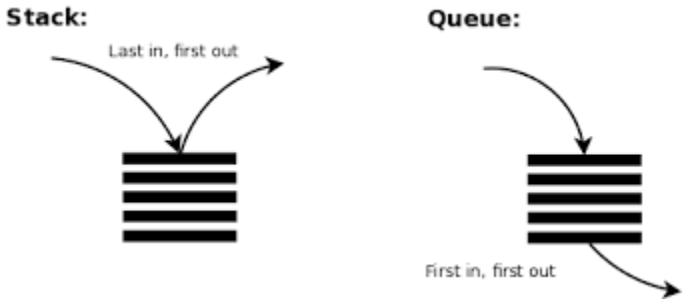
- Stack

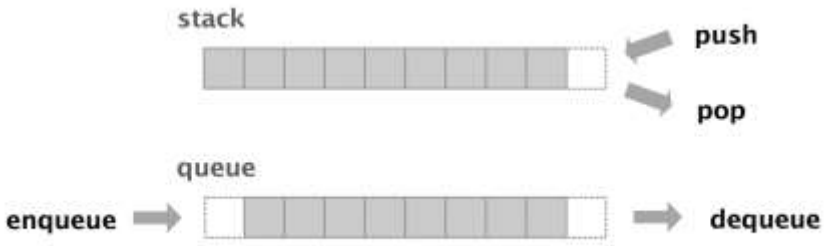
  LIFO: Last-In-First-Out

  FILO: First-In-Last-Out

- Queue

  FIFO: First-In-First-Out



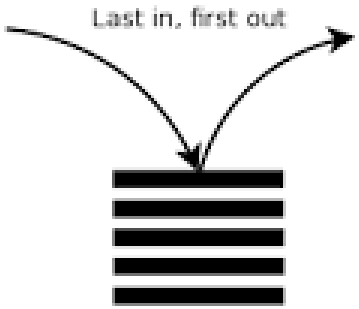


Part 1: Stacks

Stacks

- Stack
  - LIFO: Last-In-First-Out
  - FILO: First-In-Last-Out
- example: stacks of plates used in cafeterias

- $S.top$: indexes the most recently inserted element
- Underflow/Overflow



Stacks

- **Operations on stacks:**

$PUSH\ (S, x)$

$POP\ (S)$

$SIZE\ (S)$

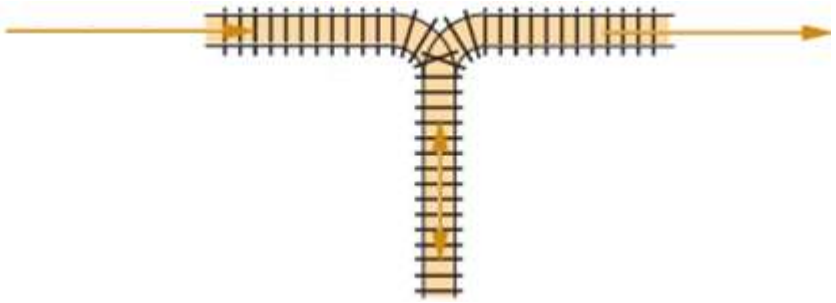
$ISEMPTY\ (S)$

$TOP\ (S)$

Stacks: Example 1

- Stack of trains

8 7 6 5 4 3 2 1



8 1 2 7 3 6 4 5

<5,4,6,3,7,2,1,8>?

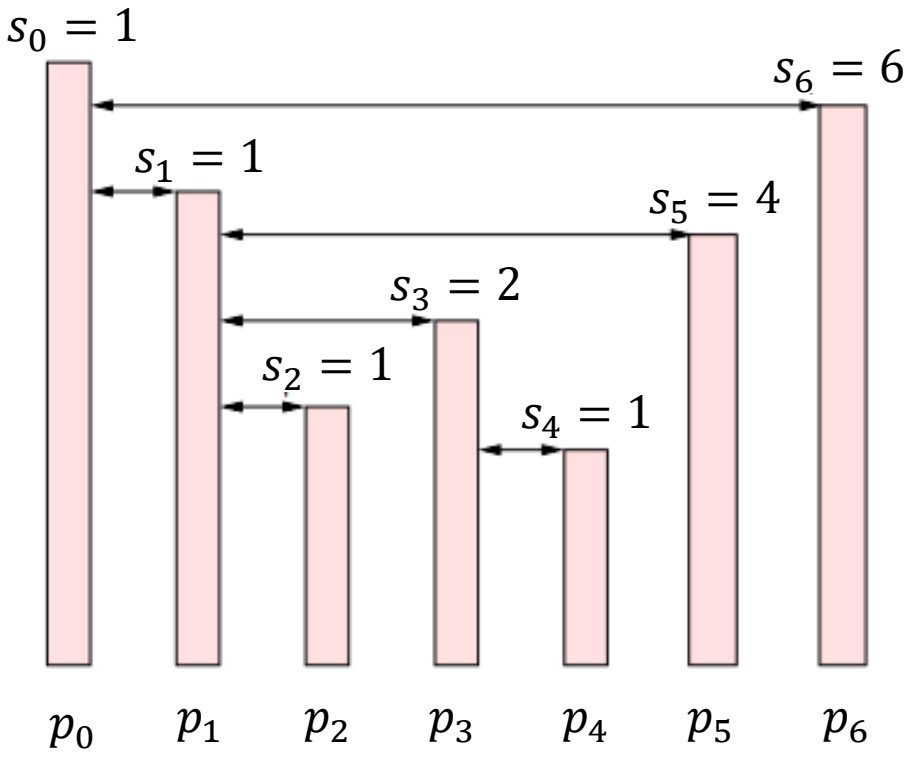
Push, Push, Push, Push, Push, Pop, Pop, Push, Pop, Pop, Push, Pop, Pop, Pop, Push, Pop

<4,3,7,8,6,2,,5,1>?

<1,8,3,6,2,7,4,5>?

Stacks: Example 2

- Compute Spans



Stacks: Example 2 (cont.)

- Compute Spans (cont.)

```
COMPUTESPANS(P)
      ▷ Input: n-element array P
      ▷ Output: n-element array S
 1  for i ← 0 to n − 1
 2      do k ← 0
 3          done ← false
 4          repeat if P[i − k] ≤ P[i]
 5                 then k ← k + 1
 6                 else done ← true
 7          until k = i or done
 8          S[i] ← k + 1
 9  return S
```

Stacks: Example 2 (cont.)

- Compute Spans (cont.)

```
COMPUTESPANS2(P)
      ▷ we use a stack D
 1  for i ← 0 to n − 1
 2     do done ← false
 3        while ( not ISEMPTY(D) or done)
 4              do if P[i] ≥ P[TOP(D)]
 5                    then POP(D)
 6                    else done ← true
 7        if ISEMPTY(D)
 8          then h ← −1
 9          else h ← TOP(D)
10        S[i] ← i − h
11        PUSH(D, i)
12  return S
```

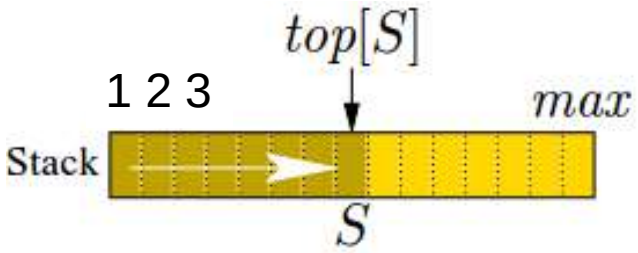
Implementation of Stack

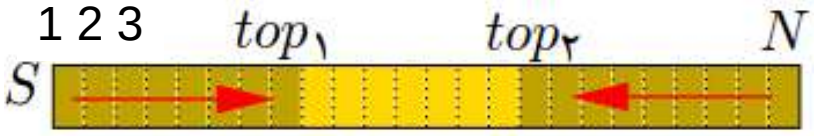
Implementation of stack by:

- Array
- Linked list

Implementation of Stack (cont.)

- **Implementation of Stack by Array**





Implementation of Stack (cont.)

- Implementation of Stack by Array (cont.)

```
SIZE(S)
    1  return  top[S]    ▷ assuming that initially top[S] = 0

ISEMPTY(S)
    1  return  SIZE(S) = 0

TOP(S)
    1  if ISEMPTY(S)
    2    then  error ("STACK IS EMPTY")
    3  return  S[top[S]]
```

Implementation of Stack (cont.)

- **Implementation of Stack by Array (cont.)**

```
PUSH(S, x)
1  if SIZE(S) = max
2    then error ("stack is full")
3  top[S] ← top[S] + 1
4  S[top[S]] ← x
```

```
POP(S)
1  if isEmpty()
2    then error ("stack is empty")
3  e ← S[top[S]]
4  top[S] ← top[S] − 1
5  return e
```

Implementation of Stack (cont.)

- **Implementation of Stack by Linked List**

```
SIZE (S)
    1  return  size[S]

ISEMPTY (S)
    1  return  (size[S] = 0)

TOP (S)
    1  if isEmpty(S)
    2    then  error ("stack is empty")
    3  return  top[S]
```

Implementation of Stack (cont.)

- **Implementation of Stack by Linked List (cont.)**

```
PUSH(S, x)
1  top[S] ← ALLOCATE-NODE(x, top[S])
2  size[S] ← size[S] + 1
```

```
POP(S)
1  if ISEMPTY(S)
2    then error ("STACK IS EMPTY")
3  n ← top[S]
4  temp ← element[n]
5  top[S] ← next[n]
6  size[S] ← size[S] − 1
7  FREE-OBJECT(n)
8  return temp
```

Part 2: Queues

Elementary Data Structures

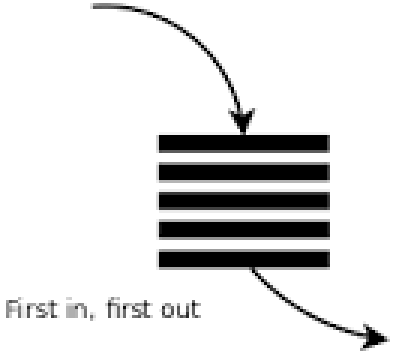
- Lists
- Stacks
- **Queues**
- Trees

Queues

- Queue:

  FIFO: First-In-First-Out

- example: a line of customers waiting to pay a cashier
- front/head
- rear/tail
- Underflows/Overflows



Queues (cont.)

- **Operations on queues**:

$ENQUEUE\ (Q, x)$

$DEQUEUE\ (Q, x)$

$SIZE\ (Q)$

$ISEMPTY\ (Q)$

$FRONT - ELEMENT\ (Q)$

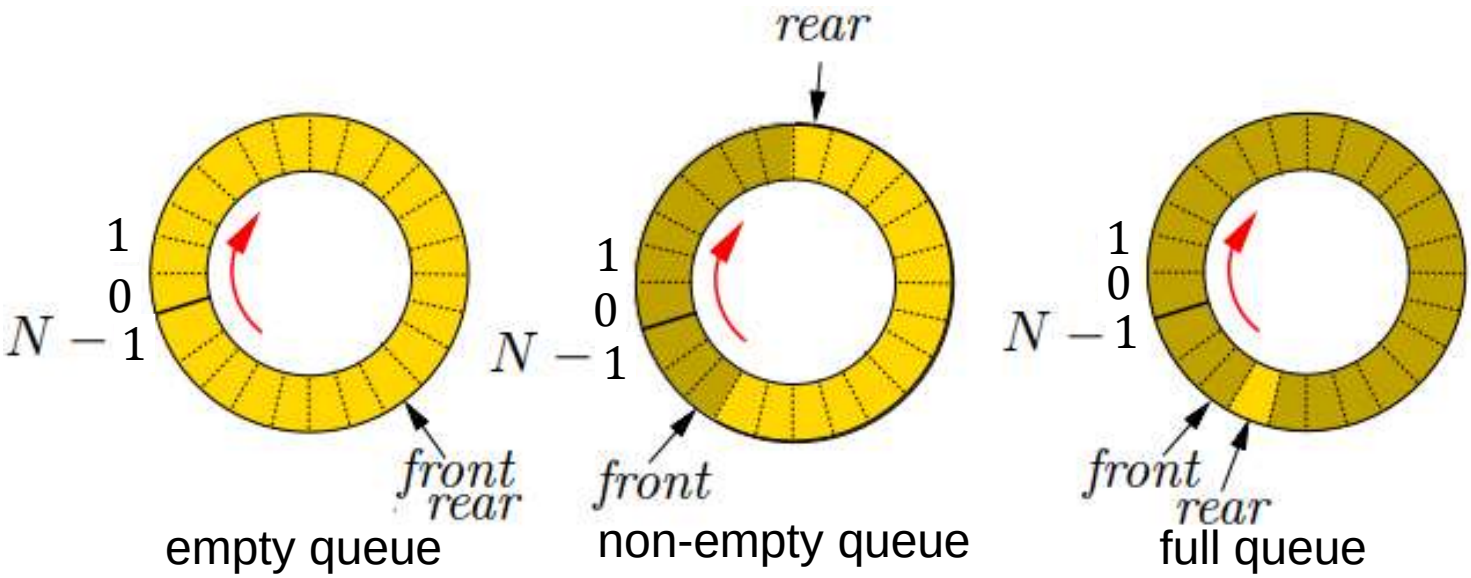
Implementation of Queue

Implementation of Queue by

- Circular Array
- Linked List

Implementation of Queue (cont.)

- **Implementation of Queue by Circular Array**



At start: $front[Q] = rear[Q] = 0$

empty queue: $front[Q] = rear[Q]$

full queue: $(max - front[Q] + rear[Q]) \bmod max = max - 1$

Implementation of Queue (cont.)

- Implementation of Queue by Circular Array (cont.)

```
SIZE (Q)
  1  return  (max − front[Q] + rear[Q]) mod max

ISEMPTY (Q)
  1  return  (front[Q] = rear[Q])

FRONT-ELEMENT (Q)
  1  if ISEMPTY(Q)
  2    then  error "Queue is empty"
  3  return  Q[front[Q]]
```

Implementation of Queue (cont.)

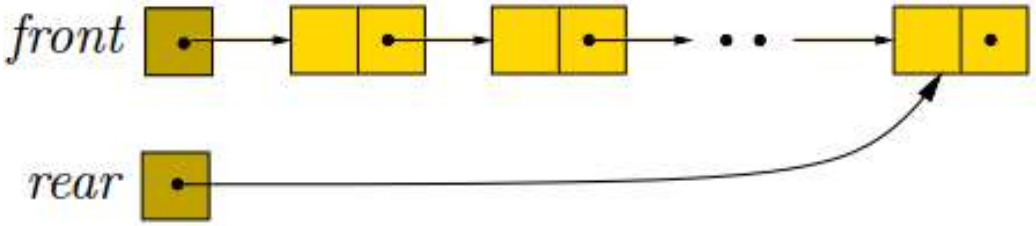
- **Implementation of Queue by Circular Array (cont.)**

```
ENQUEUE(Q, x)
1  if Size(Q) = max − 1
2    then  error "Queue is full"
3  Q[rear[Q]] ← x
4  rear[Q] ← (rear[Q] + 1) mod max
```

```
DEQUEUE(Q)
1  if isEmpty()
2    then error "Queue is empty"
3  temp ← Q[front[Q]]
4  front[Q] ← (front[Q] + 1) mod max
5  return temp
```

Implementation of Queue (cont.)

- Implementation of Queue by Linked List

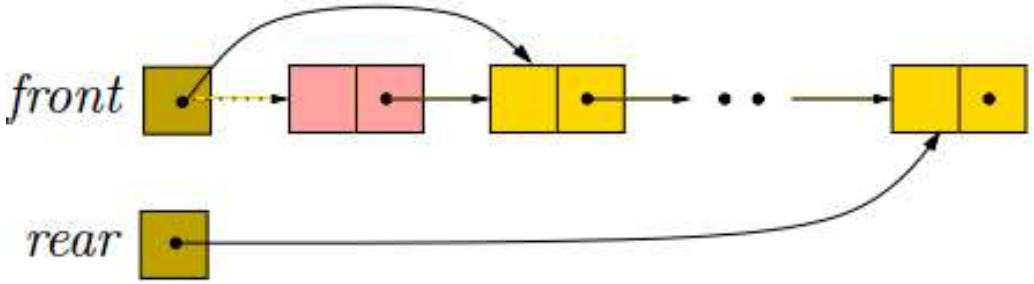



```
ISEMPTY(Q)
  1  return size[Q] = 0
```

Implementation of Queue (cont.)

- **Implementation of Queue by Linked List (cont.)**

ENQUEUE(Q, x)
1 $next[rear[Q]] \leftarrow$ ALLOCATE-NODE $(x, \textbf{null})$
2 $rear[Q] \leftarrow next[rear[Q]]$
3 $size[Q] \leftarrow size[Q] + 1$



Implementation of Queue (cont.)

- **Implementation of Queue by Linked List (cont.)**

```
DEQUEUE(Q)
1  if ISEMPTY(Q)
2    then error 'QUEUE IS EMPTY'
3  n ← front[Q]
4  x ← element[n]
5  front[Q] ← next[front[Q]]
6  FREE-NODE(n)
7  size[Q] ← size[Q] − 1
8  return x
```