Abstract Data Types (ADT)

Abstract Data Types (ADT)

- "Abstract away" the details of *how* a particular data structure is implemented, and focus only on *what* it does
- In other words, we focus on the **interface**
- E.g., in a recent lecture we saw that we can use a *linked-list* ("how") in order to implement the FIFO behaviour of a *queue* ("what")

An Abstract Data Type (ADT) is a precise, logical model of a data structure that specifies:

the type of data stored,

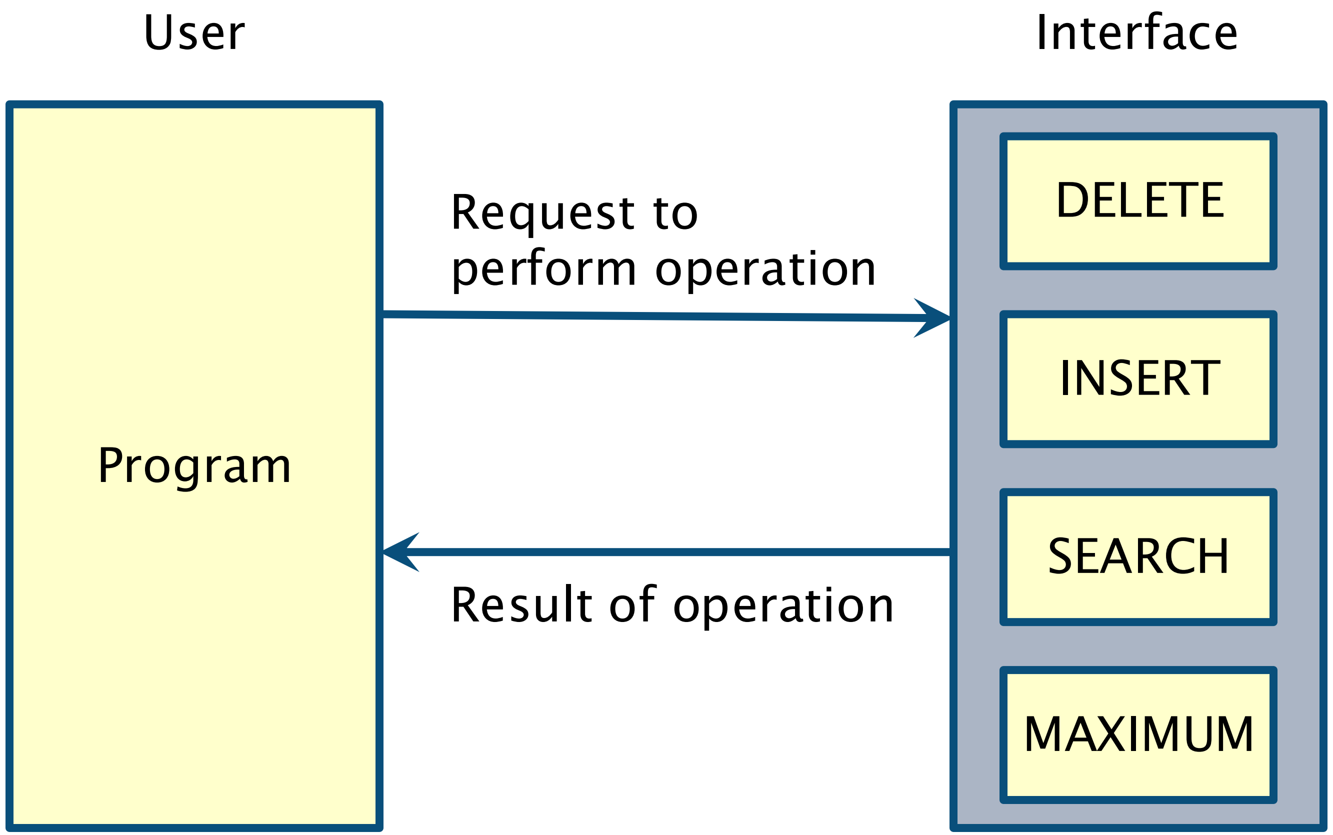
the operations supported on them, and

the types of the parameters (used in the operations)

Abstract Data Type vs Data Structures

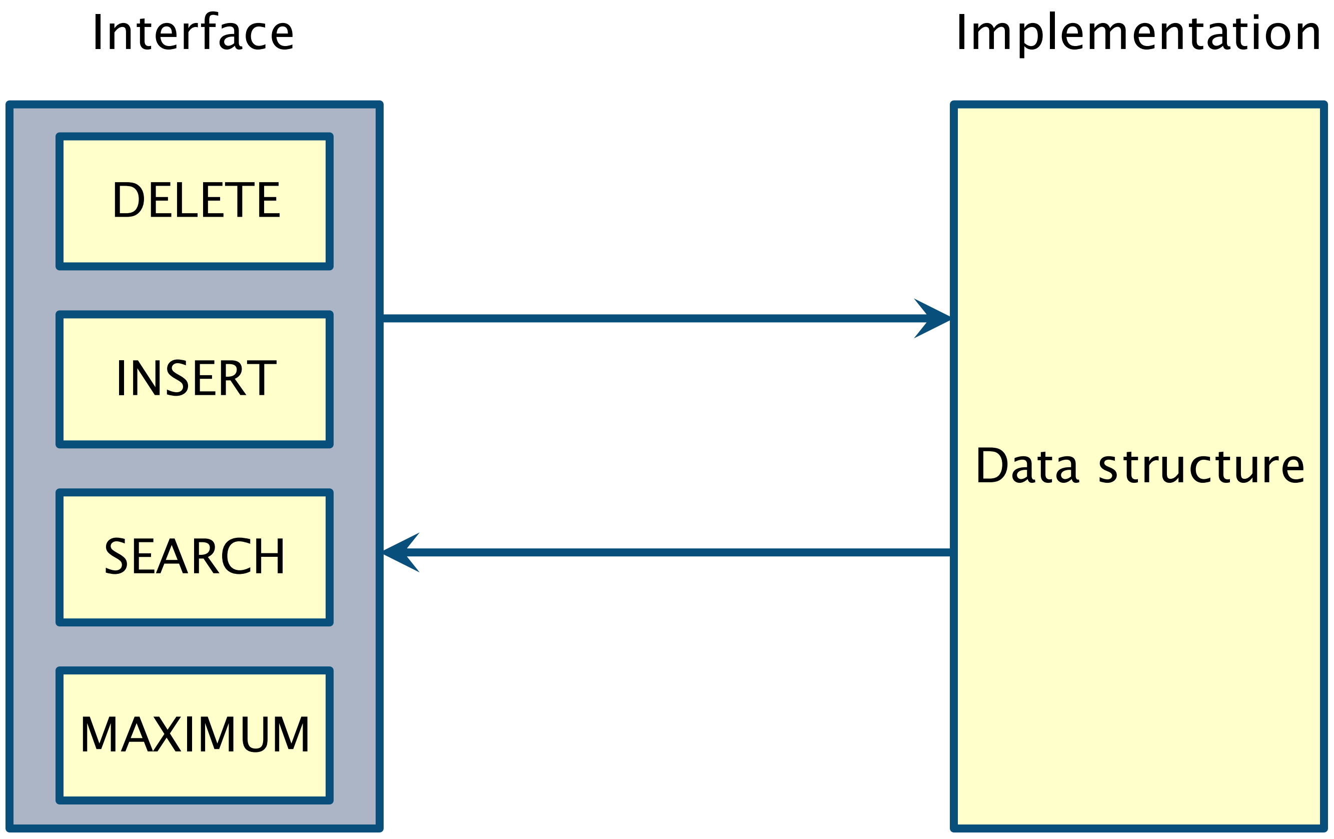
Abstract Data Type (the “what”)	Data Structures (the “how”)
Stack *(Last in, first out)*	Static Arrays Re-sizable Arrays Singly Linked Lists Doubly Linked Lists
Queue *(First in, first out)*	Static Arrays Re-sizable Arrays Singly Linked Lists Doubly Linked Lists
List *(Random access)*	Static Arrays Re-sizable Arrays Singly Linked Lists Doubly Linked Lists
Map *(“Dictionary”)*	Nested Arrays Hash Tables Tree Map

More on ADTs vs Data Structures



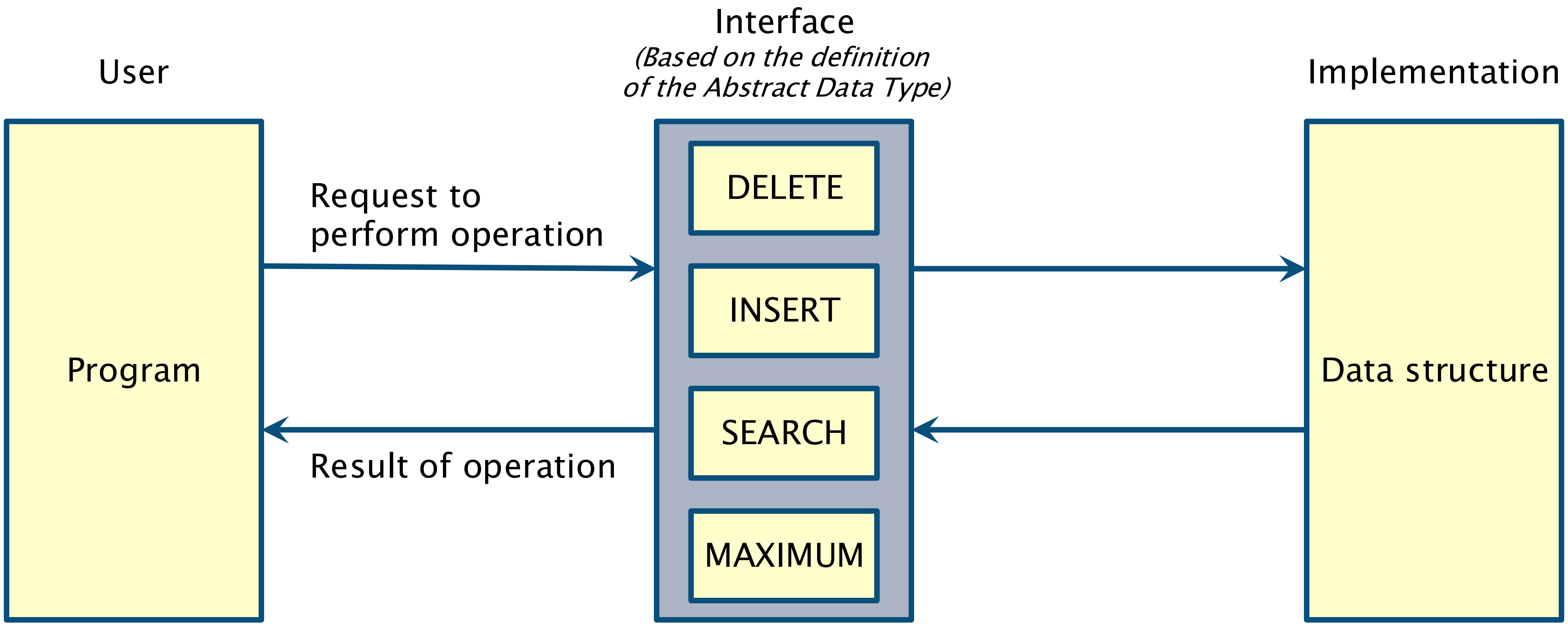
"The Wall" of ADT operations

More on ADTs vs Data Structures



“The Wall” of ADT operations

More on ADTs vs Data Structures



“The Wall” of ADT operations

STACKS

Stack

- The Stack ADT stores **arbitrary** elements
- Insertions and deletions follow the **LIFO (last-in-first-out)** policy

- Main stack operations
  - PUSH(S,x): insert element x in stack S
  - POP(S): remove and return the most recently inserted element from stack S

- Auxiliary stack operations
  - PEEK(S): return (but don't remove) the most recently inserted element from stack S (sometimes called TOP(S))
  - SIZE(S): return the number of elements stored in stack S
  - EMPTY(S): test if stack S is empty

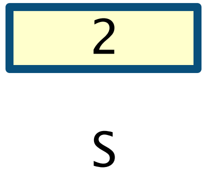
Stack: Example

- **What is the stack formed by the following sequence of instructions?**
  - PUSH(S,2)
  - PUSH(S,3)
  - PUSH(S,5)
  - POP(S)
  - PEEK(S)
  - POP(S)
  - PUSH(S,7)

S

Stack: Example

- What is the stack formed by carrying out the following sequence of instructions?
  - PUSH(S,2)
  - PUSH(S,3)
  - PUSH(S,5)
  - POP(S)
  - PEEK(S)
  - POP(S)
  - PUSH(S,7)

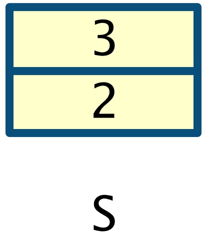


2

S

Stack: Example

- What is the stack formed by carrying out the following sequence of instructions?
  - PUSH(S,2)
  - PUSH(S,3)
  - PUSH(S,5)
  - POP(S)
  - PEEK(S)
  - POP(S)
  - PUSH(S,7)



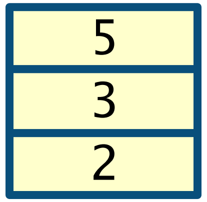
3

2

S

Stack: Example

- What is the stack formed by carrying out the following sequence of instructions?
  - PUSH(S,2)
  - PUSH(S,3)
  - PUSH(S,5)
  - POP(S)
  - PEEK(S)
  - POP(S)
  - PUSH(S,7)



5

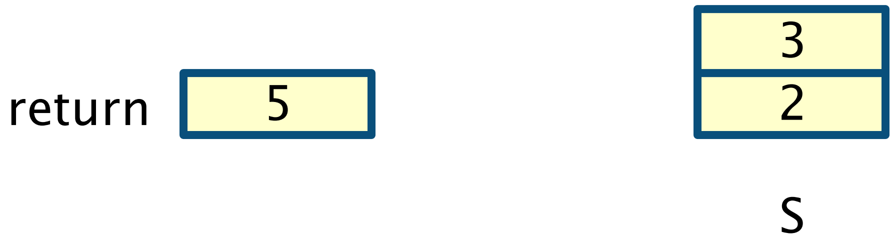
3

2

S

Stack: Example

- **What is the stack formed by carrying out the following sequence of instructions?**
  - PUSH(S,2)
  - PUSH(S,3)
  - PUSH(S,5)
  - POP(S)
  - PEEK(S)
  - POP(S)
  - PUSH(S,7)



return 5

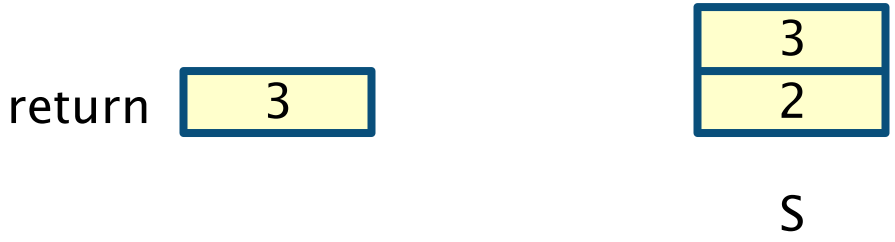
3

2

S

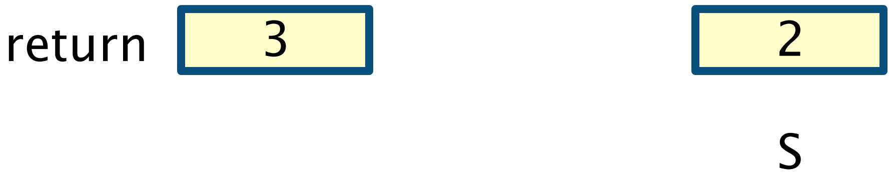
Stack: Example

- What is the stack formed by carrying out the following sequence of instructions?
  - PUSH(S,2)
  - PUSH(S,3)
  - PUSH(S,5)
  - POP(S)
  - PEEK(S)
  - POP(S)
  - PUSH(S,7)



Stack: Example

- What is the stack formed by carrying out the following sequence of instructions?
  - PUSH(S,2)
  - PUSH(S,3)
  - PUSH(S,5)
  - POP(S)
  - PEEK(S)
  - POP(S)
  - PUSH(S,7)



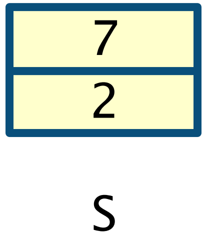
return 3

2

S

Stack: Example

- What is the stack formed by carrying out the following sequence of instructions?
  - PUSH(S,2)
  - PUSH(S,3)
  - PUSH(S,5)
  - POP(S)
  - PEEK(S)
  - POP(S)
  - PUSH(S,7)



7

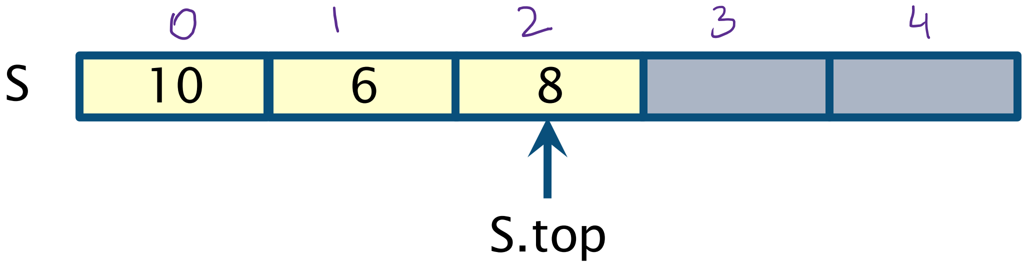
2

S

Stacks: array implementation

- A simple way of implementing a **bounded** stack is to use a (static) array*
  - Add/remove elements from the *right end* of the array
  - An attribute S.top keeps track of the index of the top element
    - S.top is updated accordingly when an element is removed/"popped" out of the stack

- Array S[0..n−1] implements a stack of at most n elements
- The stack consists of subarray S[0..S.top] where S.top < n
  - S[0] is the element at the bottom of the stack
  - S[S.top] is the element at the top
  - When S.top = −1 the stack is empty



*Think "fixed-size list" in Python

Stack Overflow

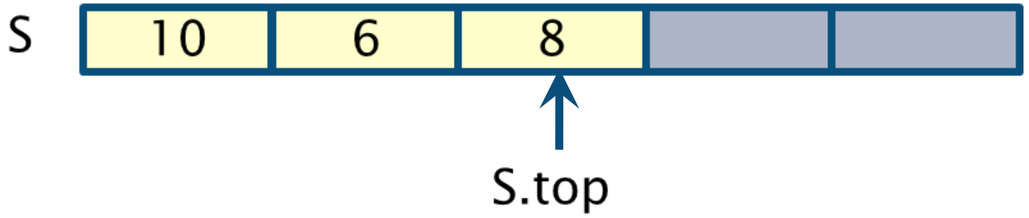
- The maximum size of the stack must be defined *a priori* and cannot be changed in run-time
- The array storing the stack elements may become full/empty
  - If we push into a full stack, the stack overflows
  - If we try to pop an empty stack, the stack underflows

- Overflows are a limitation of the static array implementation, not of the Stack ADT in general
  - In our pseudocode we will ignore stack overflows*

*Think “list” in Python

Array implementation of stacks: Operations

```
PUSH(S,x)
  S.top := S.top + 1
  S[S.top] := x
```



```
POP(S)
  if EMPTY(S)
    error “underflow”
  else
    S.top := S.top - 1
    return S[S.top + 1]
```

```
EMPTY(S)
  return S.top == -1
```

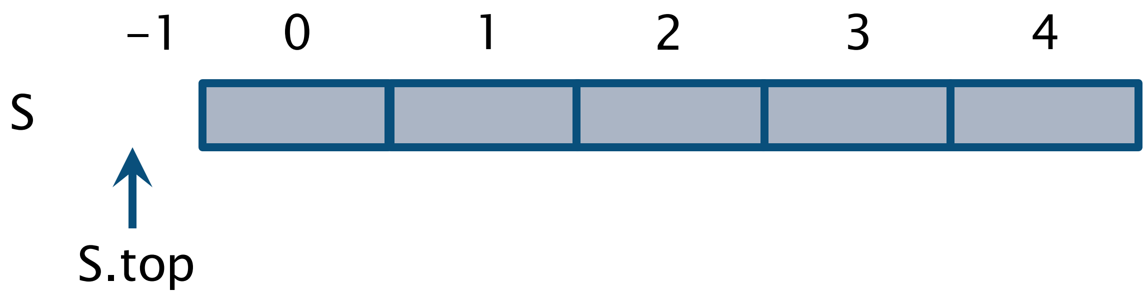
All operations run in constant time

Array implementation of stacks: Example

- **What is the stack formed by carrying out the following sequence of instructions?**
  - PUSH(S,2)
  - PUSH(S,3)
  - PUSH(S,5)
  - POP(S)
  - PEEK(S)
  - POP(S)
  - PUSH(S,7)

Initialise S

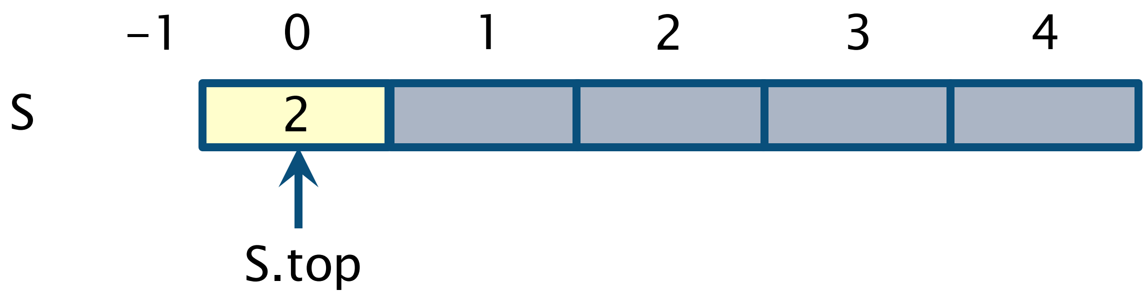
(S can contain at most 5 elements)



Array implementation of stacks: Example

- What is the stack formed by carrying out the following sequence of instructions?
  - PUSH(S,2)
  - PUSH(S,3)
  - PUSH(S,5)
  - POP(S)
  - PEEK(S)
  - POP(S)
  - PUSH(S,7)

```
PUSH(S,x)
  S.top := S.top + 1
  S[S.top] := x
```



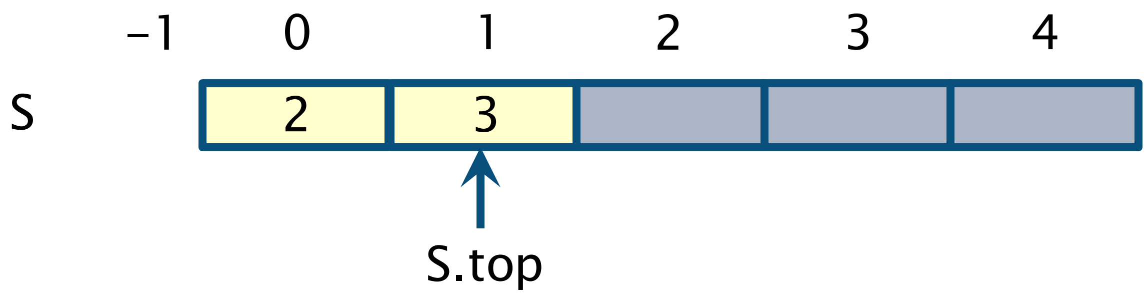
	−1	0	1	2	3	4
S		2				

S.top is incremented
Element 2 is stored in the array

Array implementation of stacks: Example

- What is the stack formed by carrying out the following sequence of instructions?
  - PUSH(S,2)
  - PUSH(S,3)
  - PUSH(S,5)
  - POP(S)
  - PEEK(S)
  - POP(S)
  - PUSH(S,7)

```
PUSH(S,x)
  S.top := S.top + 1
  S[S.top] := x
```

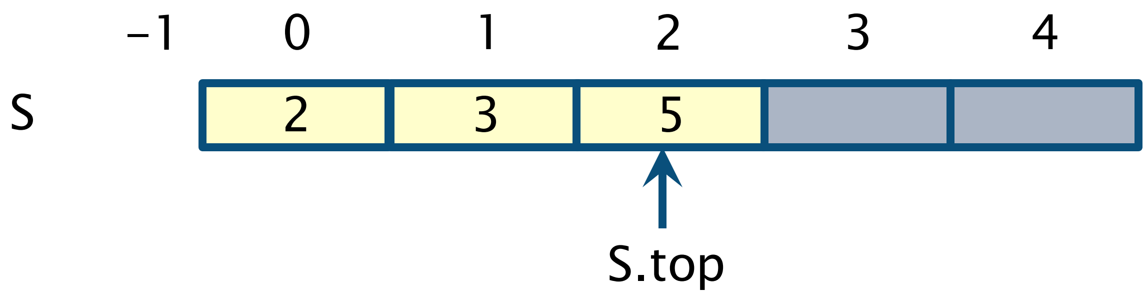


S.top is incremented
Element 3 is stored in the array

Array implementation of stacks: Example

- What is the stack formed by carrying out the following sequence of instructions?
  - PUSH(S,2)
  - PUSH(S,3)
  - PUSH(S,5)
  - POP(S)
  - PEEK(S)
  - POP(S)
  - PUSH(S,7)

```
PUSH(S,x)
  S.top := S.top + 1
  S[S.top] := x
```



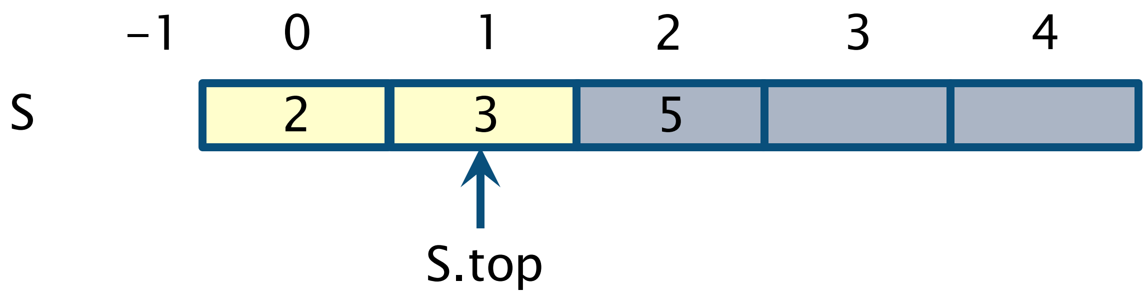
	–1	0	1	2	3	4
S		2	3	5		

S.top is incremented
Element 5 is stored in the array

Array implementation of stacks: Example

- What is the stack formed by carrying out the following sequence of instructions?
  - PUSH(S,2)
  - PUSH(S,3)
  - PUSH(S,5)
  - POP(S)
  - PEEK(S)
  - POP(S)
  - PUSH(S,7)

```
POP(S)
  if STACK-EMPTY(S)
    error “underflow”
  else S.top := S.top - 1
    return S[S.top + 1]
```

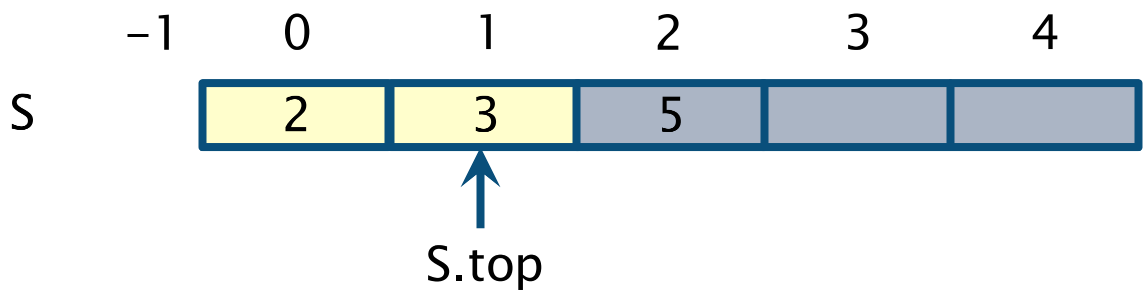


S.top is decremented
Element 5 is returned

Array implementation of stacks: Example

- What is the stack formed by carrying out the following sequence of instructions?
  - PUSH(S,2)
  - PUSH(S,3)
  - PUSH(S,5)
  - POP(S)
  - PEEK(S)
  - POP(S)
  - PUSH(S,7)

```
PEEK(S)
  if STACK-EMPTY(S)
    error "underflow"
  else
    return S[S.top]
```

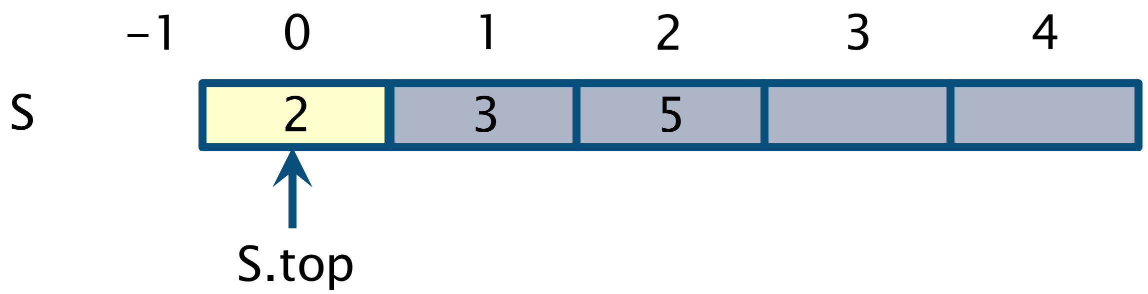


Element 3 is returned

Array implementation of stacks: Example

- What is the stack formed by carrying out the following sequence of instructions?
  - PUSH(S,2)
  - PUSH(S,3)
  - PUSH(S,5)
  - POP(S)
  - PEEK(S)
  - POP(S)
  - PUSH(S,7)

```
POP(S)
  if STACK-EMPTY(S)
    error “underflow”
  else S.top := S.top - 1
    return S[S.top + 1]
```



	−1	0	1	2	3	4
S		2	3	5		

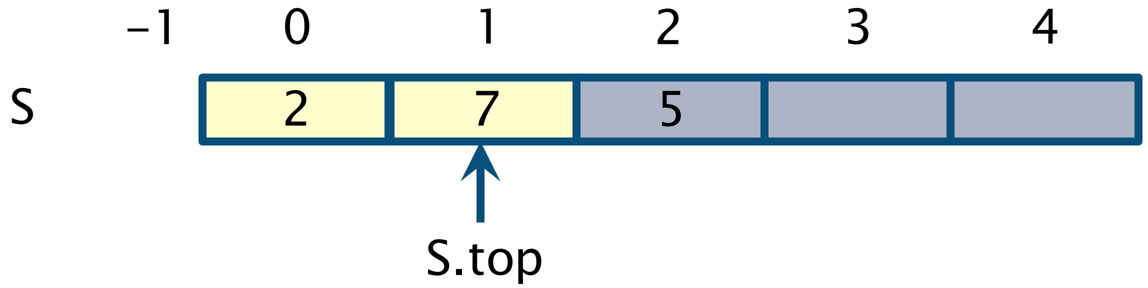
S.top

S.top is decremented
Element 3 is returned

Array implementation of stacks: Example

- What is the stack formed by carrying out the following sequence of instructions?
  - PUSH(S,2)
  - PUSH(S,3)
  - PUSH(S,5)
  - POP(S)
  - PEEK(S)
  - POP(S)
  - PUSH(S,7)

```
PUSH(S,x)
  S.top := S.top + 1
  S[S.top] := x
```



	−1	0	1	2	3	4
S		2	7	5		

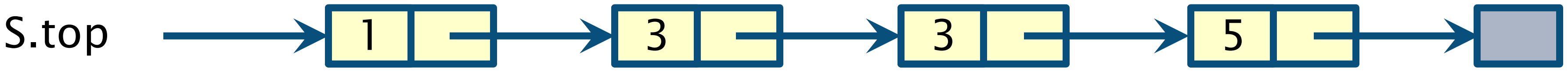
S.top is incremented
Element 7 is stored in the array

Stacks: Linked list implementation

- A stack S can be easily implemented with a (singly) linked list L:
  - L.head implements S.top
  - PUSH is implemented by INSERT at the head
  - POP is implemented by DELETE at the head
- Both operations can be performed in **constant** time
- No overflows: new elements are dynamically allocated

```
PUSH(S,node)
  node.next := S.top
  S.top := node
```

```
POP(S)
  if S.top != NIL
    node := S.top
    S.top := S.top.next
    return node
  else
    error "underflow"
```

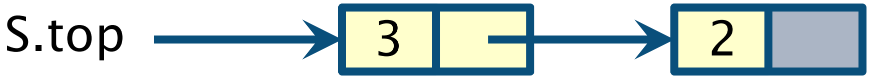


Stacks via linked lists: Example

- We perform the following operations on the stack below:
  - PUSH(S,5)
  - POP(S)

```
PUSH(S,node)
  node.next := S.top
  S.top := node
```

```
POP(S)
  if S.top != NIL
    node := S.top
    S.top := S.top.next
    return node
  else
    error “underflow”
```

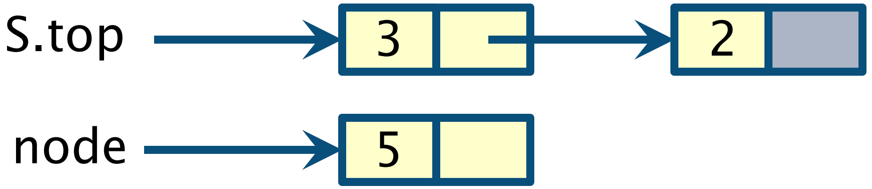


Stacks via linked lists: Example

- We perform the following operations on the stack below:
  - PUSH(S,5)
  - POP(S)

```
PUSH(S,node)
  node.next := S.top
  S.top := node
```

```
POP(S)
  if S.top != NIL
    node := S.top
    S.top := S.top.next
    return node
  else
    error “underflow”
```

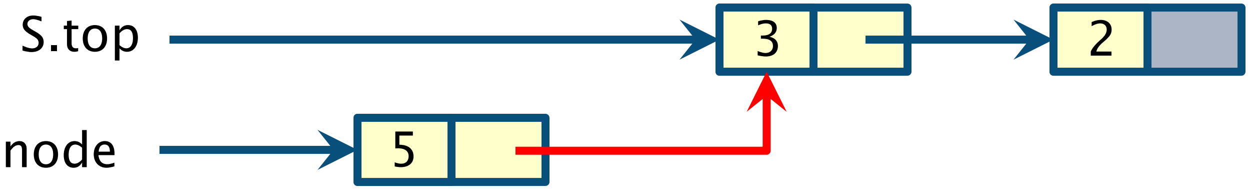


Stacks via linked lists: Example

- We perform the following operations on the stack below
  - PUSH(S,5)
  - POP(S)

```
PUSH(S,node)
  node.next := S.top
  S.top := node
```

```
POP(S)
  if S.top != NIL
    node := S.top
    S.top := S.top.next
    return node
  else
    error “underflow”
```

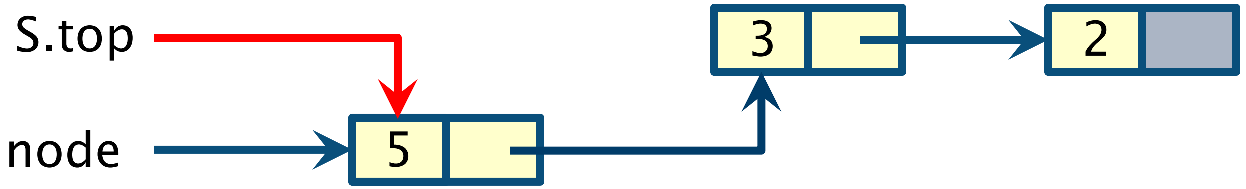


Stacks via linked lists: Example

- We perform the following operations on the stack below
  - PUSH(S,5)
  - POP(S)

```
PUSH(S,node)
   node.next := S.top
   S.top := node
```

```
POP(S)
  if S.top != NIL
    node := S.top
    S.top := S.top.next
    return node
  else
    error "underflow"
```

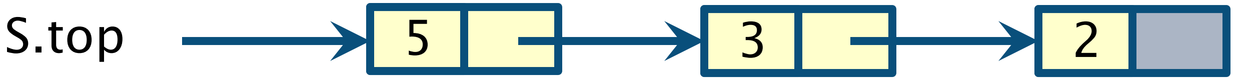


Stacks via linked lists: Example

- We perform the following operations on the stack below
  - PUSH(S,5)
  - POP(S)

```
PUSH(S,node)
  node.next := S.top
  S.top := node
```

```
POP(S)
  if S.top != NIL
    node := S.top
    S.top := S.top.next
    return node
  else
    error “underflow”
```

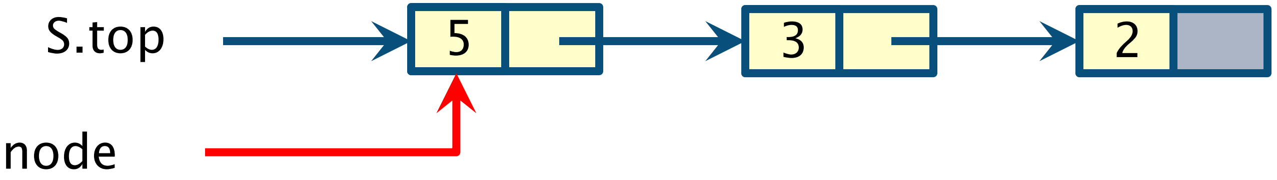


Stacks via linked lists: Example

- We perform the following operations on the stack below
  - PUSH(S,5)
  - POP(S)

```
PUSH(S,node)
  node.next := S.top
  S.top := node
```

```
POP(S)
  if S.top != NIL
    node := S.top
    S.top := S.top.next
    return node
  else
    error “underflow”
```

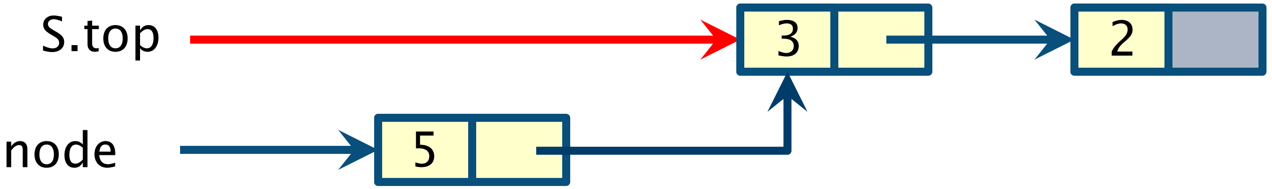


Stacks via linked lists: Example

- We perform the following operations on the stack below
  - PUSH(S,5)
  - POP(S)

```
PUSH(S,node)
   node.next := S.top
   S.top := node
```

```
POP(S)
   if S.top != NIL
      node := S.top
      S.top := S.top.next
      return node
   else
      error “underflow”
```



QUEUES

Queue

- The Queue ADT stores arbitrary elements
- Insertions and deletions follow the FIFO (first-in-first-out) policy

- Main queue operations
  - ENQUEUE(Q,x): insert element x at the *end* (rear, tail) of queue Q
  - DEQUEUE(Q): remove and return the element from the *front* (head) of queue Q

- Auxiliary queue operations
  - FRONT(Q): return the element at the front of queue Q, without removing it
  - SIZE(Q): return the number of elements stored in queue Q
  - EMPTY(S): test if queue Q is empty

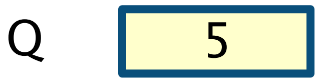
Queues: Example

- What is the queue formed by carrying out the following sequence of instructions?
  - ENQUEUE(Q,5)
  - ENQUEUE(Q,3)
  - ENQUEUE(Q,7)
  - DEQUEUE(Q)
  - DEQUEUE(Q)
  - FRONT(Q)
  - DEQUEUE(Q)
  - DEQUEUE(Q)
  - EMPTY(Q)

Q

Queues: Example

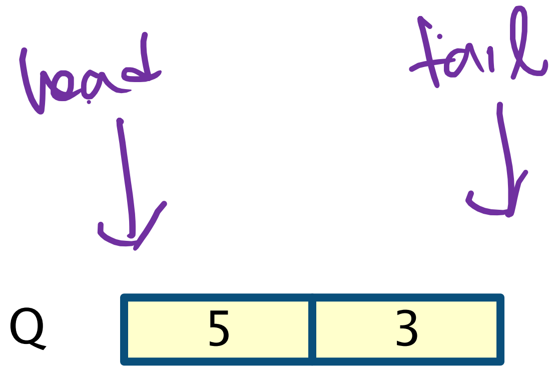
- What is the queue formed by carrying out the following sequence of instructions?
  - ENQUEUE(Q,5)
  - ENQUEUE(Q,3)
  - ENQUEUE(Q,7)
  - DEQUEUE(Q)
  - DEQUEUE(Q)
  - FRONT(Q)
  - DEQUEUE(Q)
  - DEQUEUE(Q)
  - EMPTY(Q)



Q 5

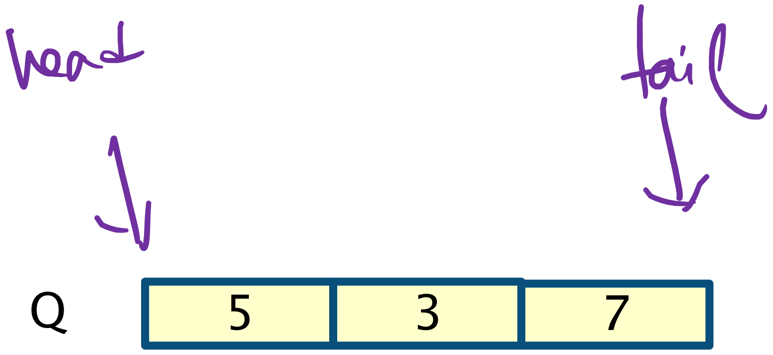
Queues: Example

- What is the queue formed by carrying out the following sequence of instructions?
  - ENQUEUE(Q,5)
  - ENQUEUE(Q,3)
  - ENQUEUE(Q,7)
  - DEQUEUE(Q)
  - DEQUEUE(Q)
  - FRONT(Q)
  - DEQUEUE(Q)
  - DEQUEUE(Q)
  - EMPTY(Q)



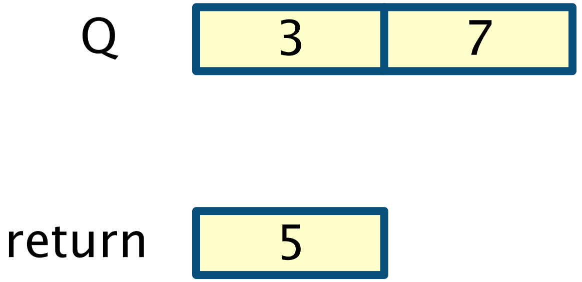
Queues: Example

- What is the queue formed by carrying out the following sequence of instructions?
  - ENQUEUE(Q,5)
  - ENQUEUE(Q,3)
  - ENQUEUE(Q,7)
  - DEQUEUE(Q)
  - DEQUEUE(Q)
  - FRONT(Q)
  - DEQUEUE(Q)
  - DEQUEUE(Q)
  - EMPTY(Q)



Queues: Example

- What is the queue formed by carrying out the following sequence of instructions?
  - ENQUEUE(Q,5)
  - ENQUEUE(Q,3)
  - ENQUEUE(Q,7)
  - DEQUEUE(Q)
  - DEQUEUE(Q)
  - FRONT(Q)
  - DEQUEUE(Q)
  - DEQUEUE(Q)
  - EMPTY(Q)

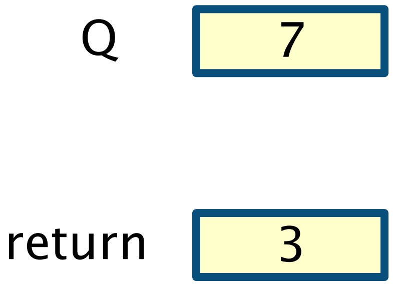


Q | 3 | 7

return | 5

Queues: Example

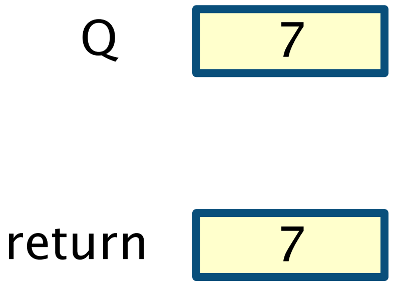
- What is the queue formed by carrying out the following sequence of instructions?
  - ENQUEUE(Q,5)
  - ENQUEUE(Q,3)
  - ENQUEUE(Q,7)
  - DEQUEUE(Q)
  - DEQUEUE(Q)
  - FRONT(Q)
  - DEQUEUE(Q)
  - DEQUEUE(Q)
  - EMPTY(Q)


Q 7

return 3

Queues: Example

- What is the queue formed by carrying out the following sequence of instructions?
  - ENQUEUE(Q,5)
  - ENQUEUE(Q,3)
  - ENQUEUE(Q,7)
  - DEQUEUE(Q)
  - DEQUEUE(Q)
  - FRONT(Q)
  - DEQUEUE(Q)
  - DEQUEUE(Q)
  - EMPTY(Q)



Q 7

return 7

Queues: Example

- What is the queue formed by carrying out the following sequence of instructions?
  - ENQUEUE(Q,5)
  - ENQUEUE(Q,3)
  - ENQUEUE(Q,7)
  - DEQUEUE(Q)
  - DEQUEUE(Q)
  - FRONT(Q)
  - DEQUEUE(Q)
  - DEQUEUE(Q)
  - EMPTY(Q)

Q

return 7

Queues: Example

- What is the queue formed by carrying out the following sequence of instructions?
  - ENQUEUE(Q,5)
  - ENQUEUE(Q,3)
  - ENQUEUE(Q,7)
  - DEQUEUE(Q)
  - DEQUEUE(Q)
  - FRONT(Q)
  - DEQUEUE(Q)
  - DEQUEUE(Q)
  - EMPTY(Q)

Q

underflow!

Queues: Example

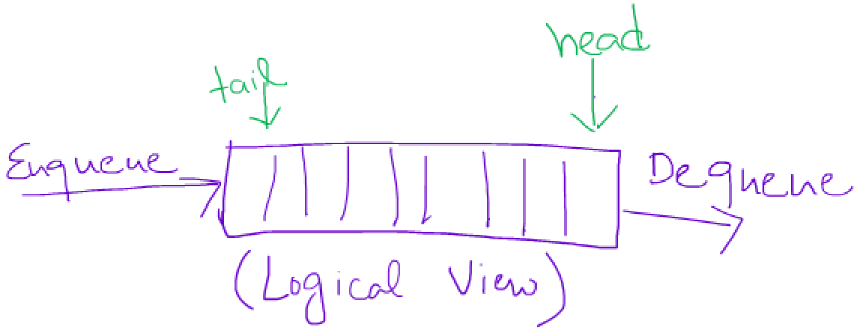
- What is the queue formed by carrying out the following sequence of instructions?
  - ENQUEUE(Q,5)
  - ENQUEUE(Q,3)
  - ENQUEUE(Q,7)
  - DEQUEUE(Q)
  - DEQUEUE(Q)
  - FRONT(Q)
  - DEQUEUE(Q)
  - DEQUEUE(Q)
  - EMPTY(Q)

Q

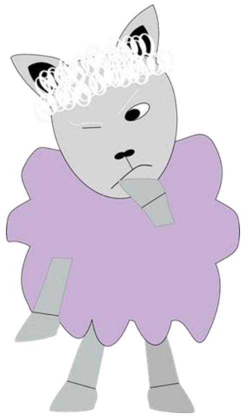
return True

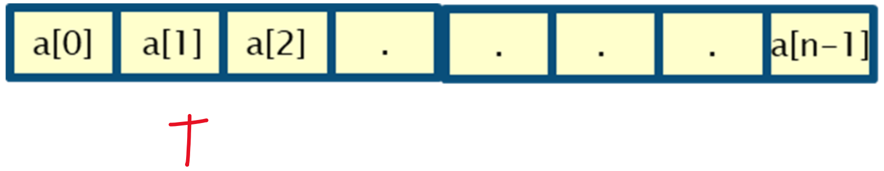
Array implementation: The Problem

- **The Problem:**
  - A Queue grows at one end (ENQUEUE operations are at the tail)
  - A Queue shrinks at the other end (DEQUEUE operations are at the head)



  - How do we use a fixed-size *array* to represent a Queue?





a[0]	a[1]	a[2]	.	.	.	.	a[n−1]

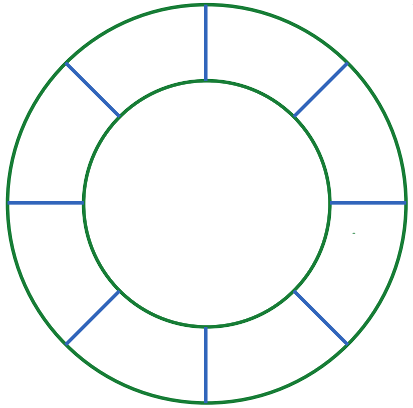
I have a cunning plan.

First: Let’s convert our arrays into *circular* arrays!

Circular arrays

- What we need is a *logically* “circular” array:

Think: Clocks!



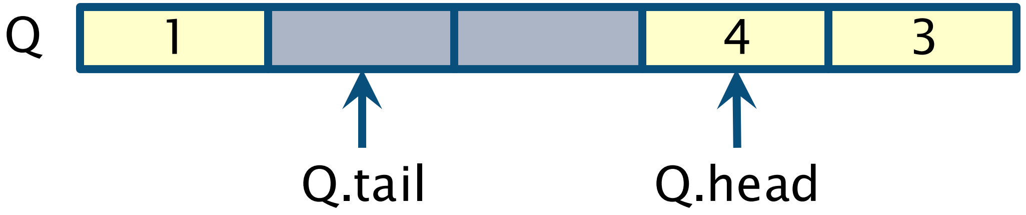
% = “mod” operator
That is, the **remainder** after
a number is divided by the **modulus**

Queues: Array implementation

- A **bounded** queue can be implemented using an array in a **circular** way
  - Wrapped-around array: location 0 immediately follows location n – 1 in a circular order

- We need two attributes of queues:
  - Q.head indexes the element at its head
  - Q.tail indexes the *next* location at which a new element will be inserted into the queue

- Array Q[0..n–1] implements a queue of at most n – 1 elements
  - Q[Q.tail] is a dummy/“empty” element
  - When Q.head = Q.tail the queue is empty
  - When Q.head is one place ahead of Q.tail, queue is full



Queues: Array implementation (cont'd)

- Elements are added to queue **Q** in **Q.tail**–th position of the array; **Q.tail** is updated accordingly (increases by 1)
  - Wrapping around: when Q.tail = n – 1, then Q.tail = 0 after enqueing

- Elements are removed from the **Q.head**–th position of the array; **Q.head** is updated (increases by 1)
  - Wrapping around: When Q.head = n – 1, then Q.head becomes 0 after the update

- Overflows are a limitation of the array–based implementation in queues as well…

Array implementation of queues: Operations

- We use the modulo operator % to wrap-around

```
EMPTY(Q)
  return Q.head = Q.tail
```

```
FULL(Q)
  return (   (Q.head = Q.tail + 1 )
          OR (Q.head = 0 and Q.tail=n-1)
          )
```

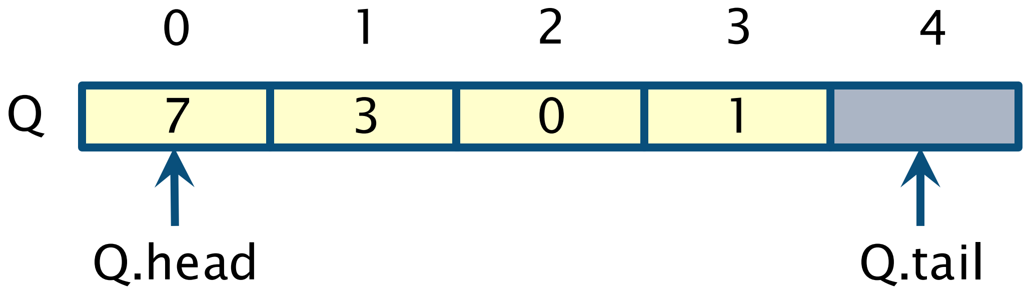
Each operation runs in constant time!

```
ENQUEUE(Q,x)
  if FULL(Q)
    error "overflow"
  else
    Q[Q.tail] := x
    Q.tail := (Q.tail + 1) % n
```

```
DEQUEUE(S)
  if EMPTY(S)
    error "underflow"
  else
    x := Q[Q.head]
    Q.head := (Q.head + 1) % n
    return x
```

Array implementation of queues: Example

- What is the queue formed by carrying out the following sequence of instructions?
  - DEQUEUE(Q)
  - ENQUEUE(Q,4)
  - DEQUEUE(Q)
  - DEQUEUE(Q)
  - ENQUEUE(Q,5)

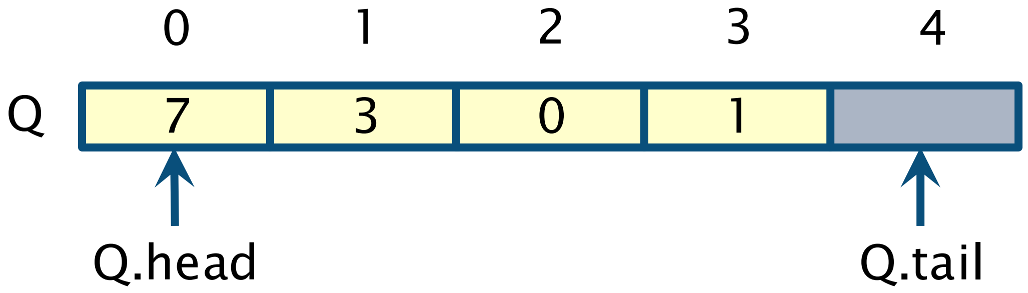


	0	1	2	3	4
Q	7	3	0	1	

Array implementation of queues: Example

- What is the queue formed by carrying out the following sequence of instructions?
  - DEQUEUE(Q)
  - ENQUEUE(Q,4)
  - DEQUEUE(Q)
  - DEQUEUE(Q)
  - ENQUEUE(Q,5)

```
DEQUEUE(S)
  if EMPTY(S)
    error “underflow”
  else x := Q[Q.head]
    Q.head := (Q.head + 1) % n
    return x
```



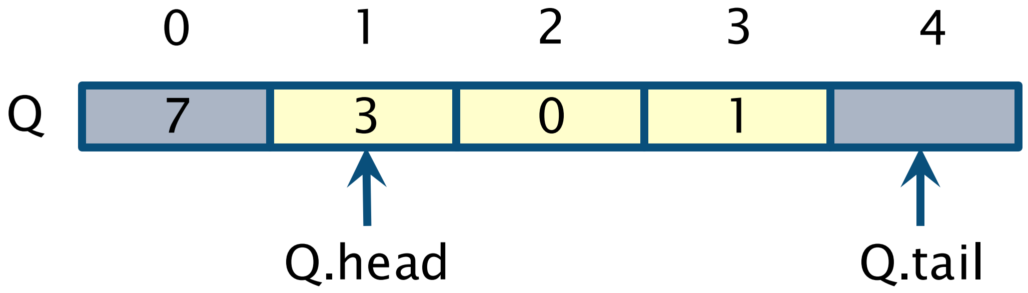
	0	1	2	3	4
Q	7	3	0	1	

Array implementation of queues: Example

- What is the queue formed by carrying out the following sequence of instructions?
  - DEQUEUE(Q)
  - ENQUEUE(Q,4)
  - DEQUEUE(Q)
  - DEQUEUE(Q)
  - ENQUEUE(Q,5)

```
DEQUEUE(S)
  if EMPTY(S)
    error “underflow”
  else x := Q[Q.head]
    Q.head := (Q.head + 1) % n
    return x
```

  - Return 7
  - Q.head is (0 + 1) mod 5 = 1

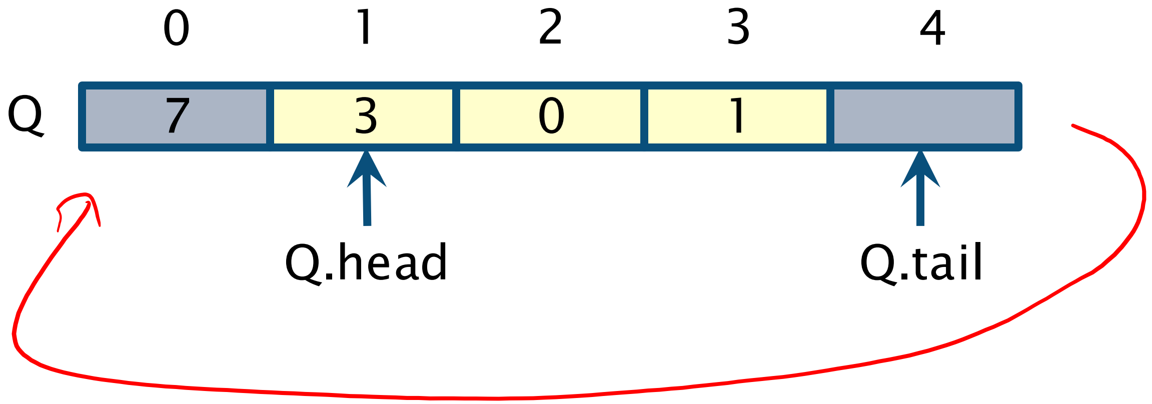


	0	1	2	3	4
Q	7	3	0	1	
		Q.head			Q.tail

Array implementation of queues: Example

- What is the queue formed by carrying out the following sequence of instructions?
  - DEQUEUE(Q)
  - ENQUEUE(Q,4)
  - DEQUEUE(Q)
  - DEQUEUE(Q)
  - ENQUEUE(Q,5)

```
ENQUEUE(Q,x)
  if FULL(Q)
    error “overflow”
  else Q[Q.tail] := x
    Q.tail := (Q.tail + 1) % n
```

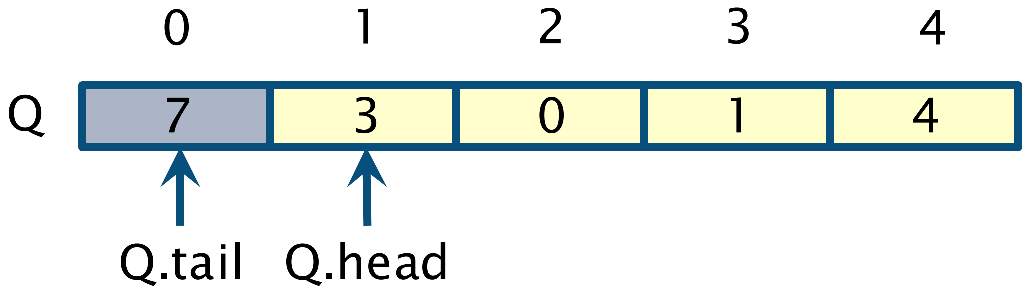


Array implementation of queues: Example

- What is the queue formed by carrying out the following sequence of instructions?
  - DEQUEUE(Q)
  - ENQUEUE(Q,4)
  - DEQUEUE(Q)
  - DEQUEUE(Q)
  - ENQUEUE(Q,5)

```
ENQUEUE(Q,x)
  if FULL(Q)
    error “overflow”
  else Q[Q.tail] := x
    Q.tail := (Q.tail + 1) % n
```

  - Q.tail is (4 + 1) mod 5 = 0



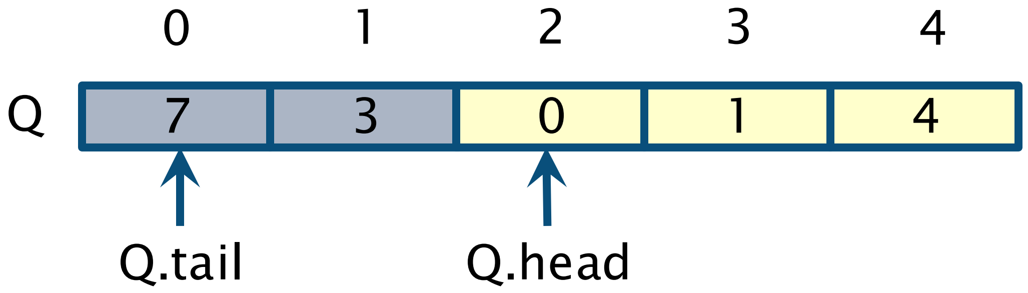
	0	1	2	3	4
Q	7	3	0	1	4

Array implementation of queues: Example

- What is the queue formed by carrying out the following sequence of instructions?
  - DEQUEUE(Q)
  - ENQUEUE(Q,4)
  - DEQUEUE(Q)
  - DEQUEUE(Q)
  - ENQUEUE(Q,5)

  - Return 3
  - Q.head is (1 + 1) mod 5 = 2

```
DEQUEUE(S)
  if EMPTY(S)
    error "underflow"
  else x := Q[Q.head]
    Q.head := (Q.head + 1) % n
    return x
```



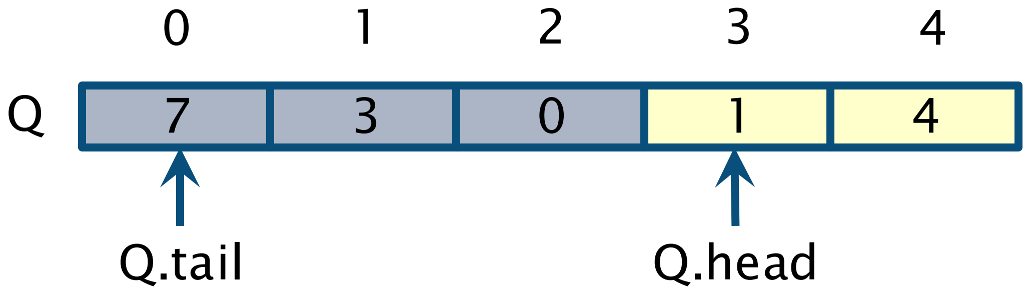
	0	1	2	3	4
Q	7	3	0	1	4

Array implementation of queues: Example

- What is the queue formed by carrying out the following sequence of instructions?
  - DEQUEUE(Q)
  - ENQUEUE(Q,4)
  - DEQUEUE(Q)
  - DEQUEUE(Q)
  - ENQUEUE(Q,5)

```
DEQUEUE(S)
  if EMPTY(S)
    error “underflow”
  else x := Q[Q.head]
    Q.head := (Q.head + 1) % n
    return x
```

  - Return 0
  - Q.head is (2 + 1) mod 5 = 3



	0	1	2	3	4
Q	7	3	0	1	4

Q.tail

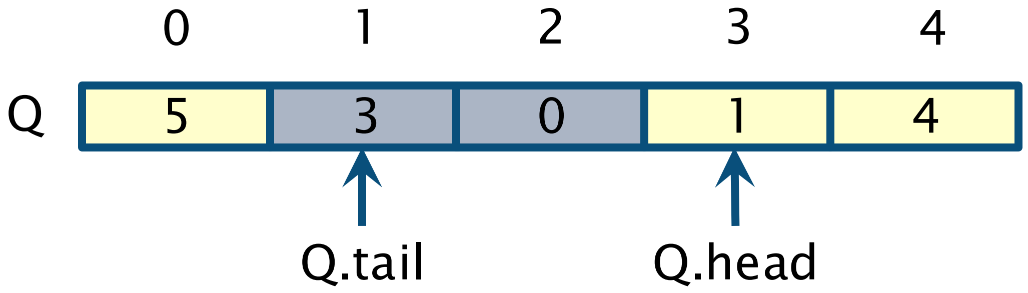
Q.head

Array implementation of queues: Example

- What is the queue formed by carrying out the following sequence of instructions?
  - DEQUEUE(Q)
  - ENQUEUE(Q,4)
  - DEQUEUE(Q)
  - DEQUEUE(Q)
  - ENQUEUE(Q,5)

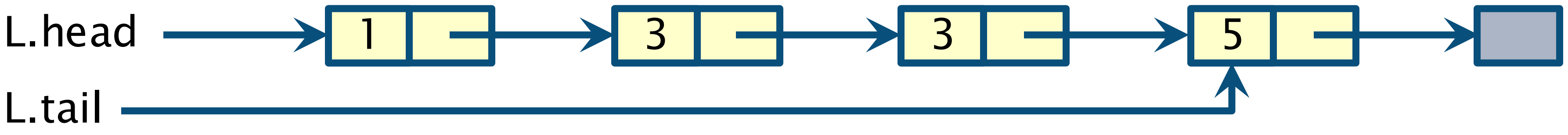
```
ENQUEUE(Q,x)
  if FULL(Q)
    error “overflow”
  else Q[Q.tail] := x
    Q.tail := (Q.tail + 1) % n
```

  - Q.tail is (0 + 1) mod 5 = 1



	0	1	2	3	4
Q	5	3	0	1	4

Queues: linked list implementation



- We will see the details in the Problem Set

- Very briefly:
    - (Singly) linked list with (head and) tail pointer
    - Insert at tail
    - Delete at head
    - No need to “circularize” (unlike the array implementation before)

Lists, Sets, and Maps

OTHER ABSTRACT DATA TYPES

The List ADT

- We have been using “lists” extensively in Python already.
  - However, it can also be viewed *more abstractly* as an ADT

- The List ADT stores a sequence of arbitrary elements
  - Can insert elements at *any location* (compare with Stack, Queue)

- Fundamental data type in most functional programming languages

- Main list operations
  - GET(L,i): return the element of list L at index i, without removing it
  - SET(L,i,x): replace the element of list L at index i, with x
  - ADD(L,x): insert element x to the end of list L
  - ADD-AT(L,i,x): insert element x at index i in list L, shifting all elements after this
  - REMOVE(L,i): remove and return the element of list L at index i, shifting all elements after this

The Set ADT

- We have come across SET as a mathematical concept
  - An unordered collection of elements without repetition
  - In the computing world, they can also be viewed as an abstract data type

- A Set ADT will define the following methods:
  - ADD (S,x) #add the element x to the set, if not already there
  - REMOVE (S,x) #remove the element x from the set, if present
  - CONTAINS (S,x) #checks if the set S contains the element x
  - SIZE (S) #returns the cardinality of the set
  - ISEMPTY(S) #checks if the set is empty

- Python has a built-in `set` *data structure* that implements this Set *ADT*

The Map ADT

- Lists are useful for (linearly) ordered data that can be accessed by <u>position</u>

- In many applications, ordering our data in such a way is irrelevant: what about allowing for more general indexing by “keys”
  - Eg: storing a number against each month of the year (number of customers, items sold, etc)

key : value

```
January : 123
February: 112
March   : 99
…
```

- Python readily provides the *dictionary* data type to achieve this

The Map ADT (cont'd)

- A map models a searchable collection of (key,value) pairs
  - Other names: associative array, symbol table, dictionary
  - Multiple entries with the same key are not allowed
    - keys must form a *set*

- Main map operations
  - INSERT(M,x): add a pair x = (k,v) to map M
  - DELETE(M,x): remove a pair x = (k,v) from map M
  - LOOKUP(M,k): if a pair with key k exists in M, return its value v

- Auxiliary map operations
  - EMPTY(M): test if map M is empty