FALL 2018 CISC 311

# OBJECT & STRUCTURE & ALGORITHM I

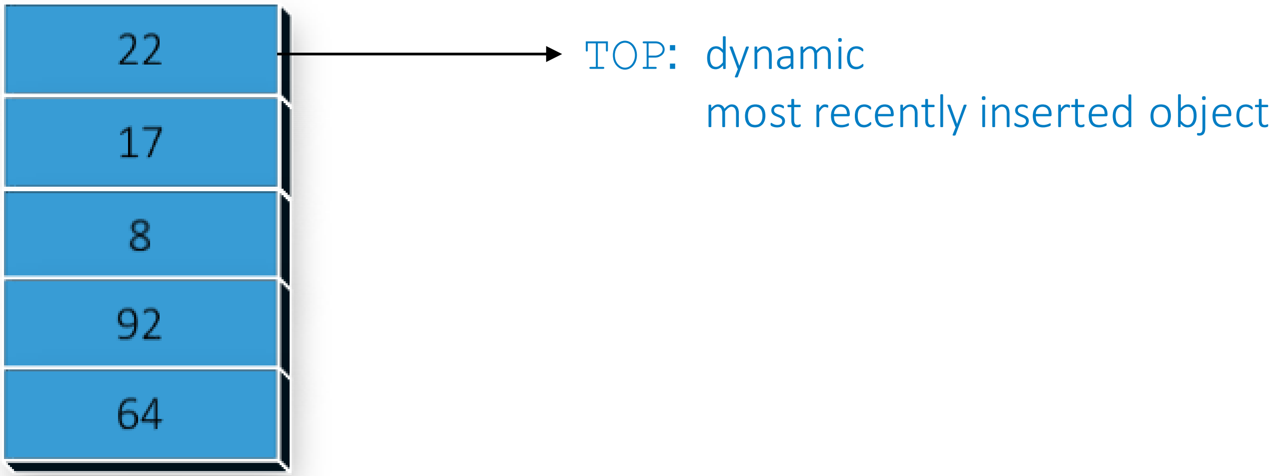
– Chapter 5: Stacks and Queues

# Outline

- The stack ADT
  - Singly linked list-based implementation
  - Array-based implementation
- The queue ADT
  - Singly linked list-based implementation
  - Circular array-based implementation
- The deque ADT
  - Doubly linked list-based implementation
  - Circular array-based implementation
- Applications

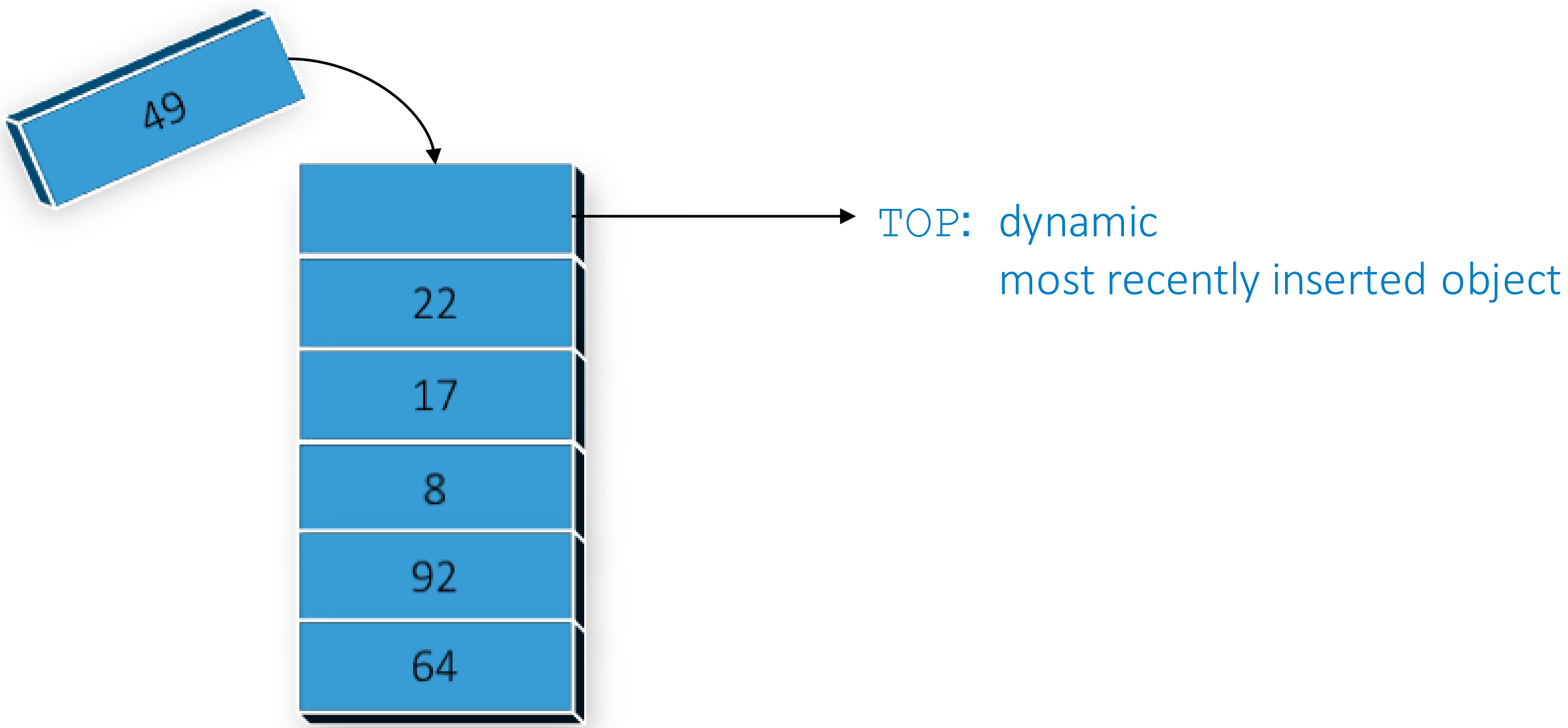
# The Stack ADT

- A stack is a collection of objects that are inserted and removed according to the *last-in, first-out* (*LIFO*) principle
  - Insert an object into a stack at any time
  - Only access or remove the most recently inserted object (top)
- The Stack ADT stores arbitrary objects
- Think of a spring-loaded plate dispenser
  - Example:



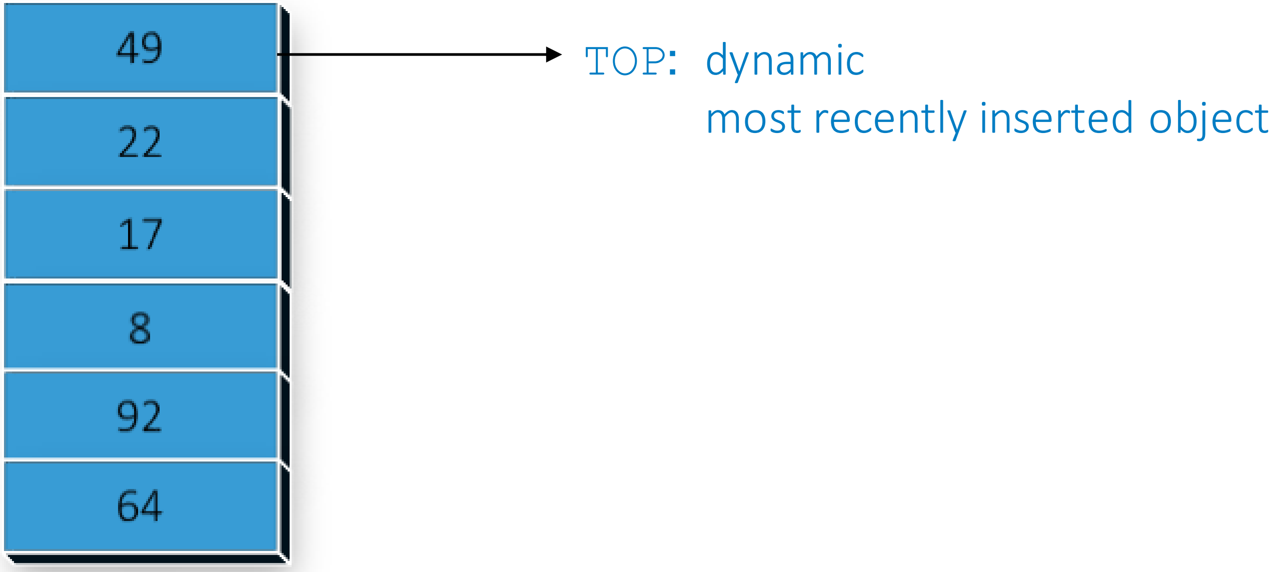
# Stack Methods

- `push()`: adds element *e* to the top of the stack
  - Step 1: increment `TOP` so it points to the space just above the previous `TOP`



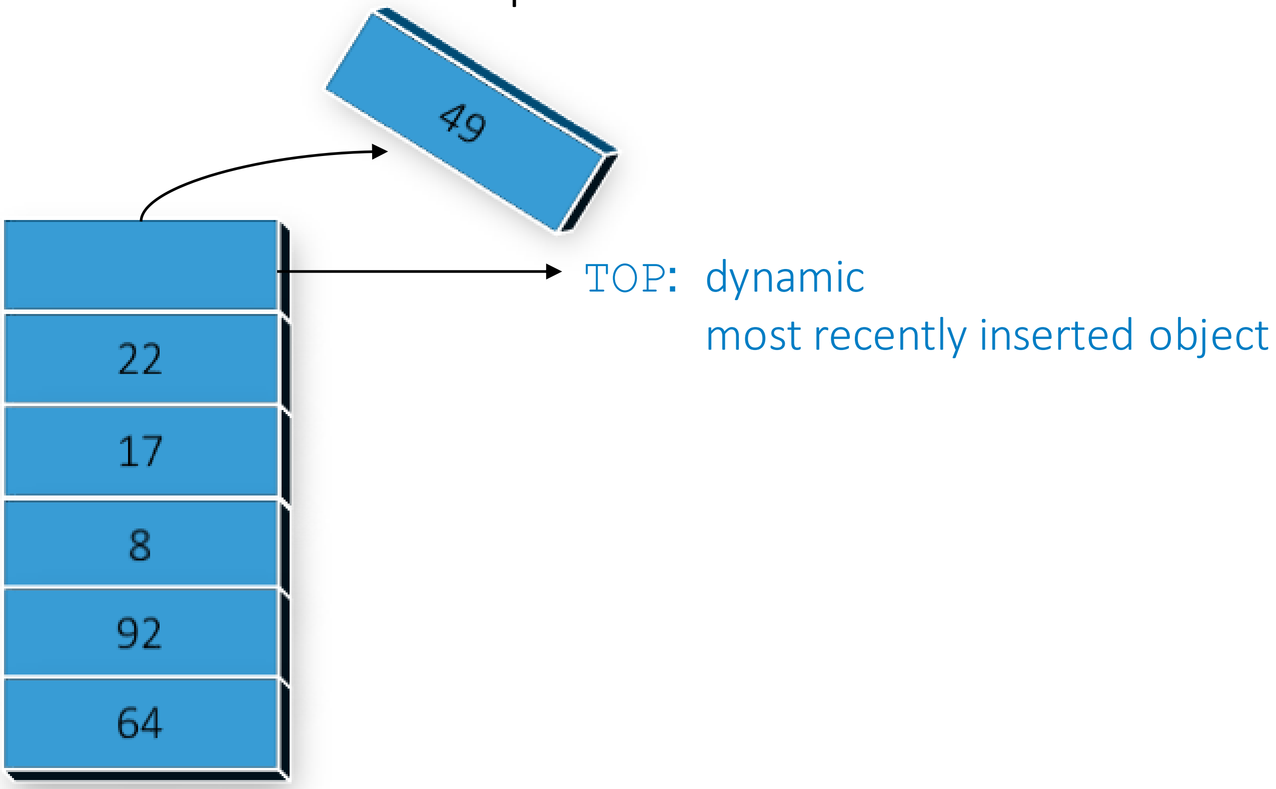
# Stack Methods (cont'd.)

- `push()`: adds element $e$ to the top of the stack
  - Step 2: insert new object



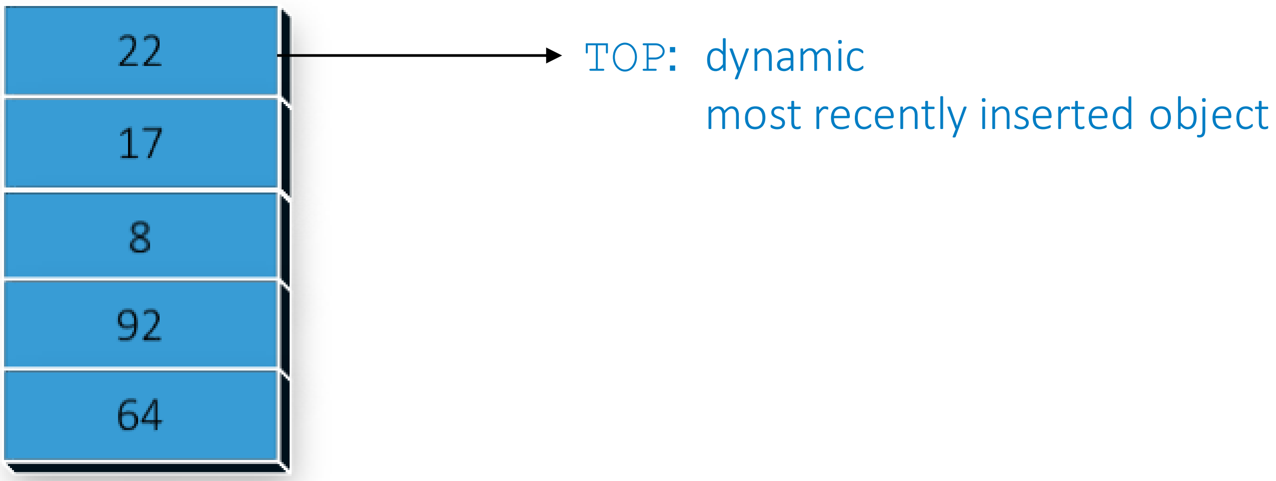
# Stack Methods (cont'd.)

- `pop()`: removes and returns the `TOP` element from the stack or `null` if the stack is empty
  - Step 1: remove the value at top



# Stack Methods (cont'd.)

- `pop()`: removes and returns the `TOP` element from the stack or `null` if the stack is empty
  - Step 2: decrement `TOP`

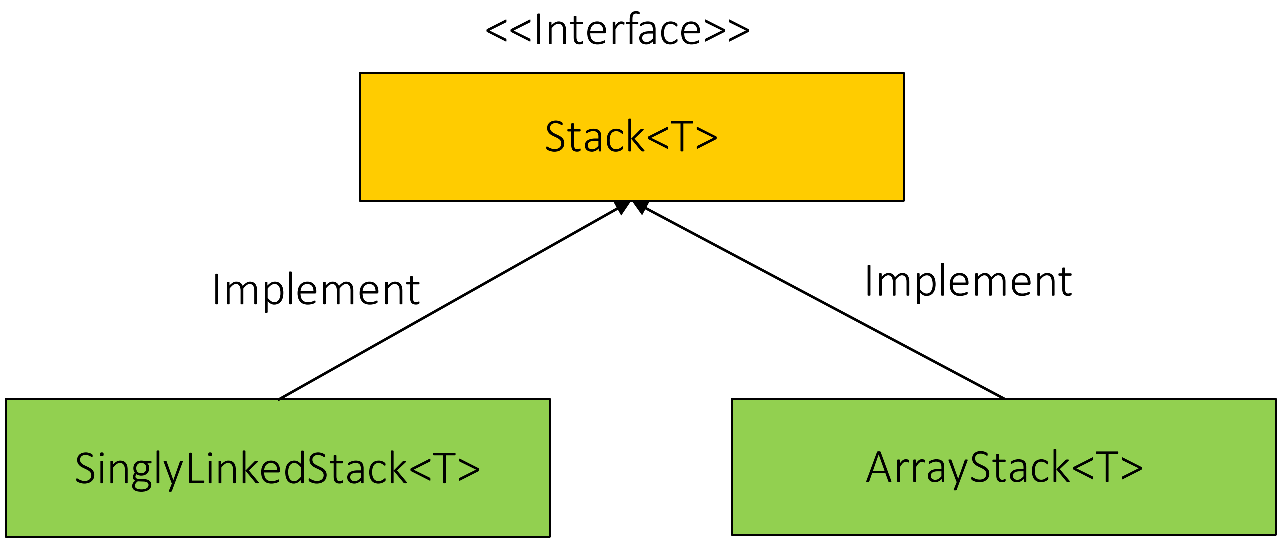


# Stack Methods (cont'd.)

| Method | Description |
| --- | --- |
| `size()` | Returns the number of elements in the stack |
| `isEmpty()` | Returns a boolean indicating whether the stack is empty |
| `top()` | Returns the `top` element of the stack, without removing it or `null` if the stack is empty |
| `push(e)` | Adds element `e` to the top of the stack |
| `pop()` | Removes and returns the `top` element from the stack or `null` if the stack is empty |

# Stack Implementation

- Singly Linked list-based implementation
- Array-based implementation



# Singly Linked List-based Implementation

- Create a singly linked list `sll`

```
SinglyLinkedList<T> sll = new SinglyLinkedList<>();
```

| Stack Method | Singly Linked List Method |
|---|---|
| `size()` | `sll.size()` |
| `isEmpty()` | `sll.isEmpty()` |
| `top()` | `sll.first()` |
| `push(e)` | `sll.addFirst(e)` |
| `pop()` | `sll.removeFirst()` |

**SinglyLinkedStack.java**

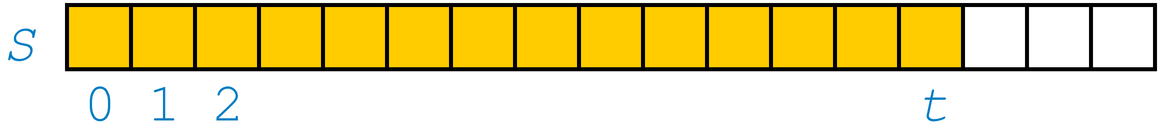
# Array-based Implementation

- A simple way of implementing the Stack ADT uses an array
- A variable keeps track of the index of the top element
- We add elements from left to right

```
Algorithm size()
    return t + 1
```

```
Algorithm isEmpty()
    return t = -1
```

```
Algorithm top()
    if isEmpty() then
        return null
    return S[t]
```

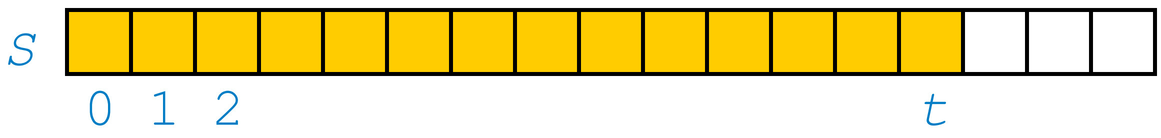



**ArrayStack.java**

# Array-based Implementation (cont'd.)

- Operation push throws an exception if the array is full
- This exception is implementation-dependent

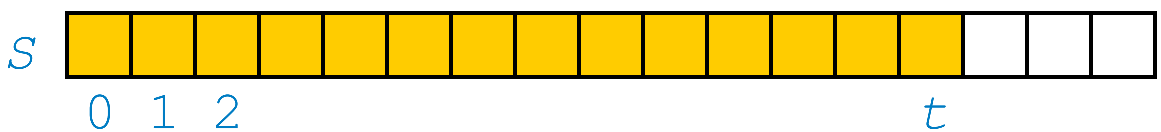
```
Algorithm push(o)
    if size() = S.length then
        throw IllegalStateException
    t ← t + 1
    S[t] ← o
```




ArrayStack.java

# Array-based Implementation (cont'd.)

**Algorithm** *pop*()
  **if** *isEmpty*() **then**
    **return** null
  $answer \leftarrow S[t]$
  $S[t] \leftarrow$ null
  $t \leftarrow t - 1$
  **return** *answer*




**ArrayStack.java**

# Performance and Limitations

- Performance
  - Let $n$ be the number of elements in the stack
  - The space used is $O(n)$
  - Each operation runs in time $O(1)$
- Limitations
  - The maximum size of the stack must be defined a priori and cannot be changed
  - Trying to push a new element into a full stack causes an implementation-specific exception

# Performance and Limitations (cont'd.)

| Stack Method | Time Complexity |
| --- | --- |
| `size()` | $O(1)$ |
| `isEmpty()` | $O(1)$ |
| `top()` | $O(1)$ |
| `push(e)` | $O(1)$ |
| `pop()` | $O(1)$ |

# Summary: Stack

- A stack allows access to the last item inserted
- The important stack operations are pushing (inserting) an item onto the top of the stack and popping (removing) the item that's on the top
- These data structures can be implemented with arrays or with other mechanisms such as linked list

| | Access/Update | Search | Insertion **`push()`** | Deletion **`pop()`** |
|---|---|---|---|---|
| Stack | $O(n)$ | $O(n)$ | $O(1)$ | $O(1)$ |

# Example I

- Given an array of integers, write a program that can reverse it with time complexity $O(n)$
  - Example:

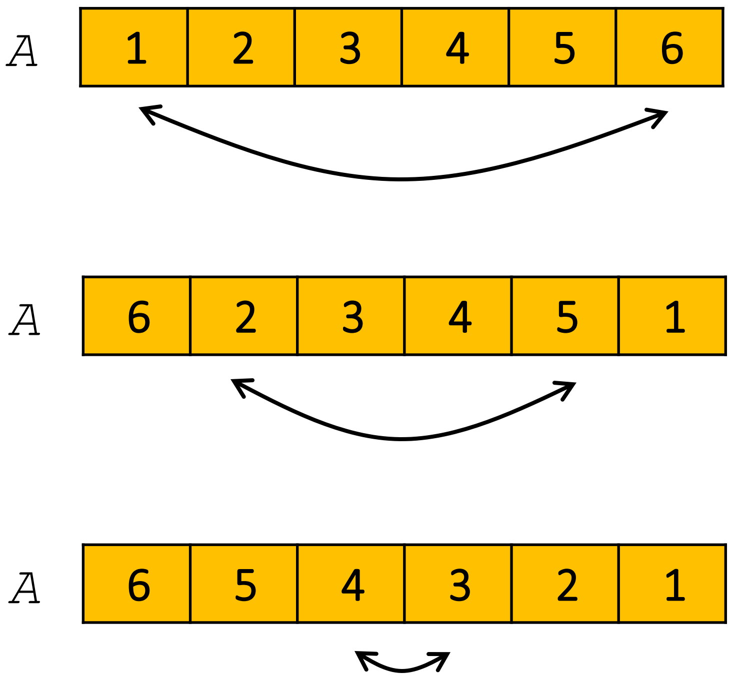
Before: $A$

| | | | | | |
|---|---|---|---|---|---|
| 1 | 2 | 3 | 4 | 5 | 6 |

After: $A$

| | | | | | |
|---|---|---|---|---|---|
| 6 | 5 | 4 | 3 | 2 | 1 |

# Iterative Algorithm

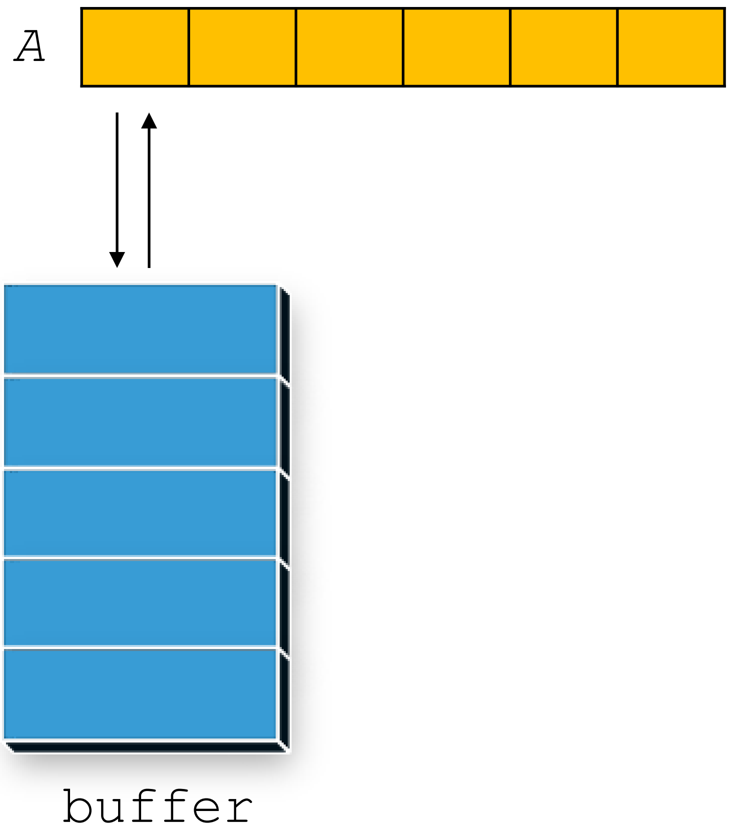



**Algorithm** reverseArray1($A$)
  **Input** array $A$ of $n$ integers
  **Output** array $A$ in reversed order
  $start \leftarrow 0$
  $end \leftarrow n - 1$
  **while** $start <= end$ **do**
    $temp \leftarrow A[start]$
    $A[start] \leftarrow A[end]$
    $A[end] \leftarrow temp$
    $start \leftarrow start + 1$
    $end \leftarrow end - 1$
  **return** $A$

Algorithm **reverseArray1** runs in $O(n)$ time

`ReverseArray1.java`

# Stack Algorithm



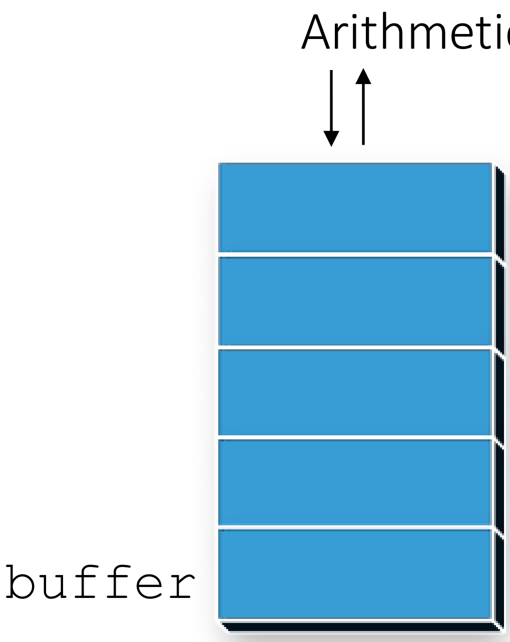

Algorithm reverseArray2($A$)
  Input array $A$ of $n$ integers
  Output array $A$ in reversed order
    $stack \leftarrow$ empty
    for $i \leftarrow 0$ to $n - 1$ do
      $stack$.push($A[i]$)
    for $i \leftarrow 0$ to $n - 1$ do
      $A[i] \leftarrow stack$.pop()
    return $A$

Algorithm **reverseArray2** runs in $O(n)$ time

`ReverseArray2.java`

# Example II/Exercise 5.1

- Design a calculator that can evaluate an arithmetic expression
  - Accept single () and +/– operators only
  - Read the input as a string literal
  - For example: "1+3−(21+7)+12−6+(9+3)−11"




**SimpleStackCalculator.java**

# Evaluating Arithmetic Expressions

- Precedence
  - () has precedence over +/-
- Associativity
  - Operators of the same precedence group
  - Evaluated from left to right
  - Example: $(x - y) + z$ rather than $x - (y + z)$
- Idea: push each operator on the stack, but first pop and perform higher and equal precedence operations

# Evaluating Arithmetic Expressions (cont'd.)

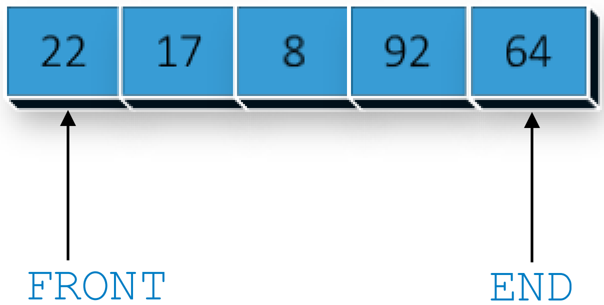
- Ordinary arithmetic expressions are written in infix notation, so-called because the operator is written between the two operands
- In postfix notation, the operator follows the two operands
- Arithmetic expressions are typically evaluated by translating them to postfix notation and then evaluating the postfix expression
- A stack is a useful tool both for translating an infix to a postfix expression and for evaluating a postfix expression

# Applications of Stacks

- Direct applications
  - Page-visited history in a Web browser
  - Undo sequence in a text editor
  - Chain of method calls in the Java Virtual Machine
- Indirect applications
  - Auxiliary data structure for algorithms
  - Component of other data structures

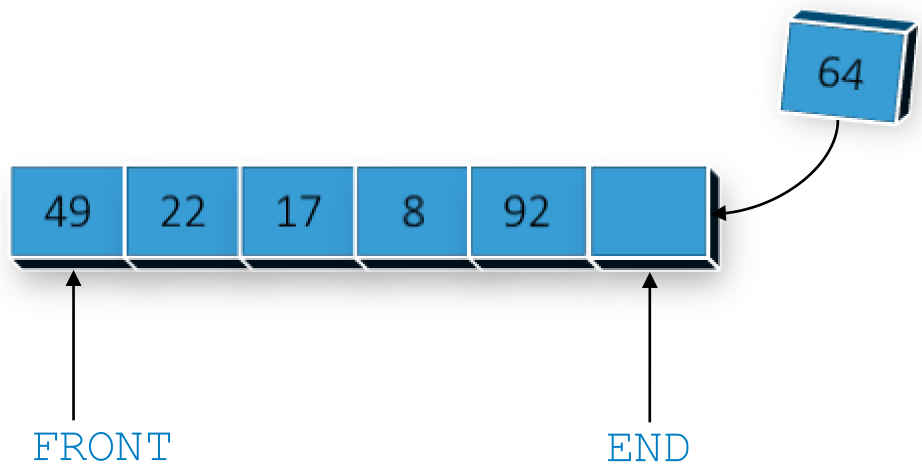
# The Queue ADT

- A queue is a collection of objects that are inserted and removed according to the *first-in*, *first-out* (*FIFO*) principle
  - Insertions are at the rear of the queue
  - Removals are at the front of the queue
- The Queue ADT stores arbitrary objects
  - Example:



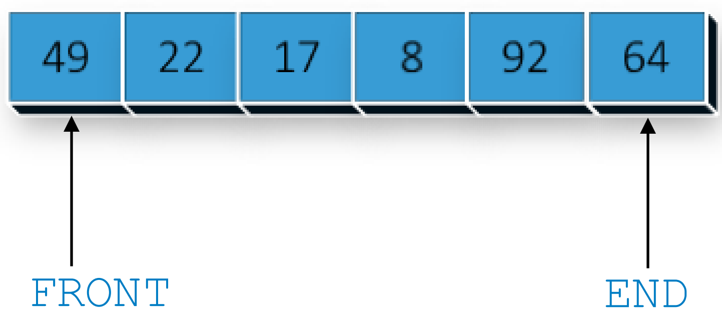
# Queue Methods

- `enqueue(e)`: adds element *e* to the back of queue
  - Step 1: increment `END` so it points to the space just after the previous `END`



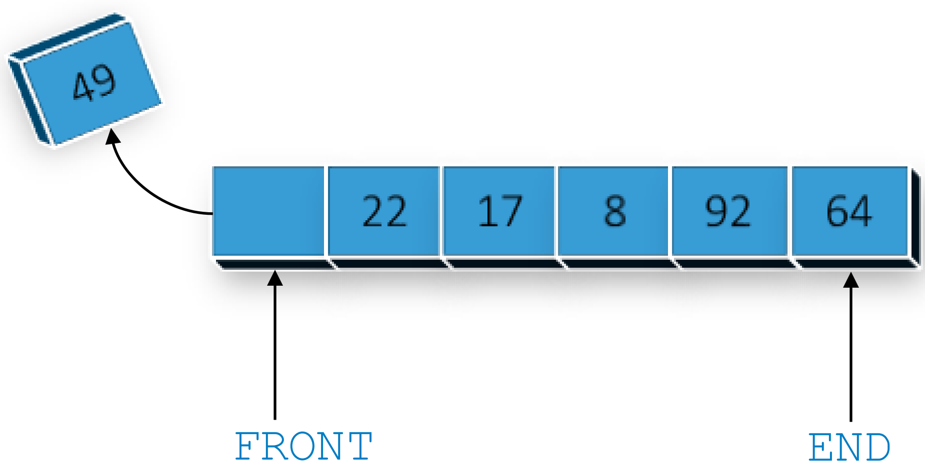
# Queue Methods (cont'd.)

- `enqueue(e)`: adds element *e* to the back of queue
  - Step 2: insert new object



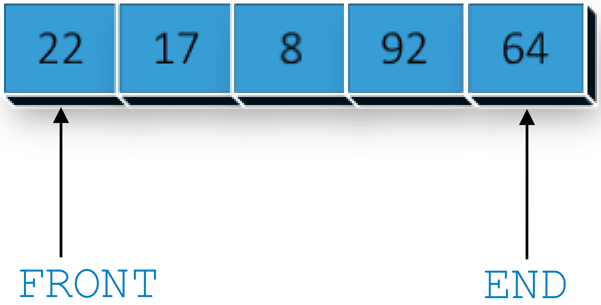
# Queue Methods (cont'd.)

- `dequeue()`: removes and returns the first element from the queue or `null` if the queue is empty
  - Step 1: remove the value at front



# Queue Methods (cont'd.)

- `dequeue()`: removes and returns the first element from the queue or `null` if the queue is empty
  - Step 2: decrement `FRONT`

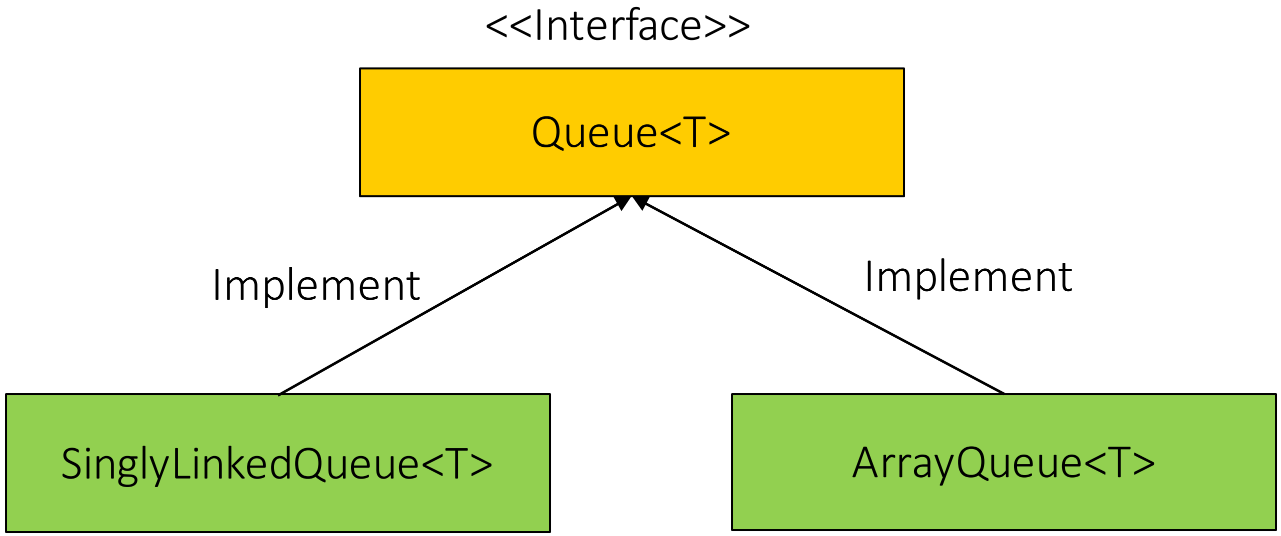


# Queue Methods (cont'd.)

| Method | Description |
|---|---|
| `size()` | Returns the number of elements in the queue |
| `isEmpty()` | Returns a boolean indicating whether the queue is empty |
| `first()` | Returns the first element of the queue, without removing it or `null` if the queue is empty |
| `enqueue(e)` | Adds element *e* to the back of queue |
| `dequeue()` | Removes and returns the first element from the queue or `null` if the queue is empty |

# Queue Implementation

- Linked list-based implementation
- Circular array-based implementation



# Singly Linked List-based Implementation

- Create a singly linked list `sll`

```
SinglyLinkedList<T> sll = new SinglyLinkedList<>();
```

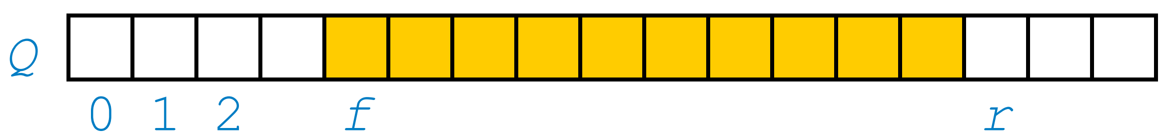
| Queue Method | Singly Linked List Method |
|---|---|
| `size()` | `sll.size()` |
| `isEmpty()` | `sll.isEmpty()` |
| `first()` | `sll.first()` |
| `enqueue(e)` | `sll.addLast(e)` |
| `dequeue()` | `sll.removeFirst()` |

**SinglyLinkedQueue.java**

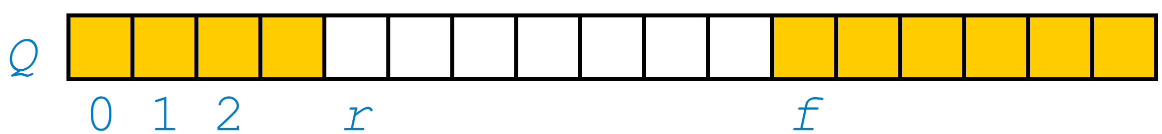
# Array-based Implementation

- Use an array of size $N$ in a circular fashion
- Two variables keep track of the front and size
  - $f$ : index of the front element
  - $sz$ : number of stored elements
- When the queue has fewer than $N$ elements, array location $r = (f + sz) \bmod N$ is the first empty slot past the rear of the queue

Normal configuration




Wrapped-around configuration



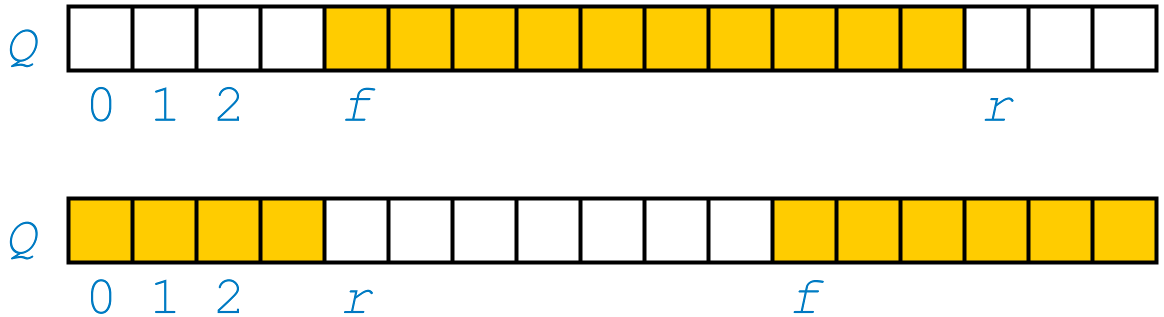
# Array-based Implementation (cont'd.)

- We use the modulo operator (remainder of division)

```
Algorithm size()
    return sz
```

```
Algorithm isEmpty()
    return sz = 0
```

```
Algorithm first()
    if isEmpty() then
        return null
    return Q[f]
```

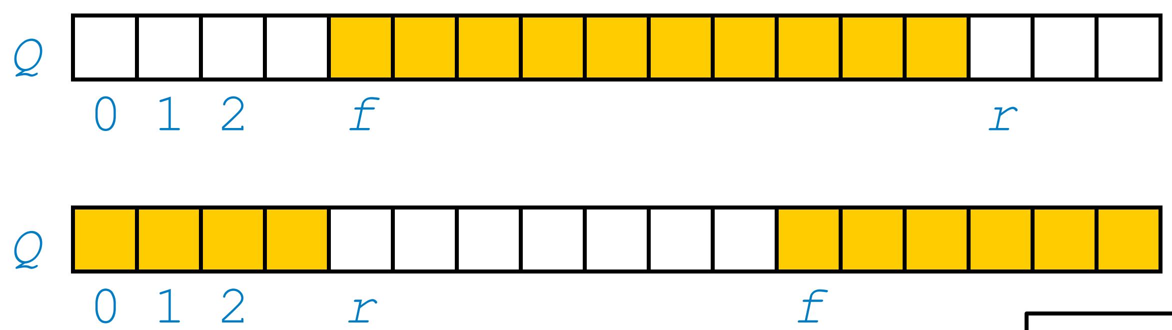



`ArrayQueue.java`

# Array-based Implementation (cont'd.)

- Note that operation dequeue returns null if the queue is empty

**Algorithm** dequeue()
    **if** isEmpty() **then**
        **return** null
    $answer \leftarrow Q[f]$
    $f \leftarrow (f + 1) \bmod N$
    $sz \leftarrow sz - 1$
    **return** $answer$

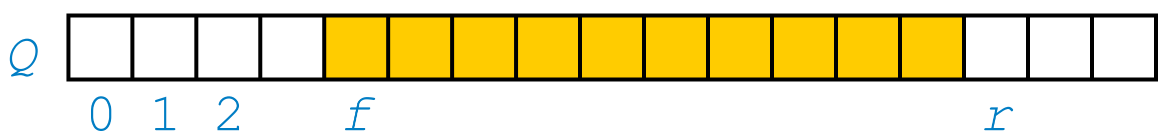


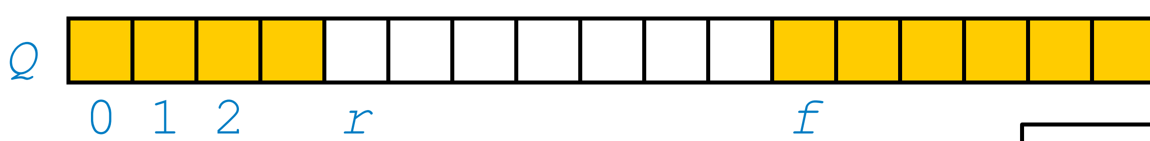

`ArrayQueue.java`

# Array-based Implementation (cont'd.)

- Operation enqueue throws an exception if the array is full
- This exception is implementation-dependent

**Algorithm** enqueue($o$)
    if $sz = Q.length$ then
        throw IllegalStateException
    $r \leftarrow (f + sz) \bmod N$
    $Q[r] \leftarrow o$
    $sz \leftarrow sz + 1$




`ArrayQueue.java`

# Performance

| Queue Method | Time Complexity |
|---|---|
| `size()` | $O(1)$ |
| `isEmpty()` | $O(1)$ |
| `first()` | $O(1)$ |
| `enqueue(e)` | $O(1)$ |
| `dequeue()` | $O(1)$ |

# Summary: Queue

- A queue allows access to the first item that was inserted
- The important queue operations are inserting an item at the rear of the queue and removing the item from the front of the queue
- A queue can be implemented as a circular queue, which is based on an array in which the indices wrap around from the end of the array to the beginning

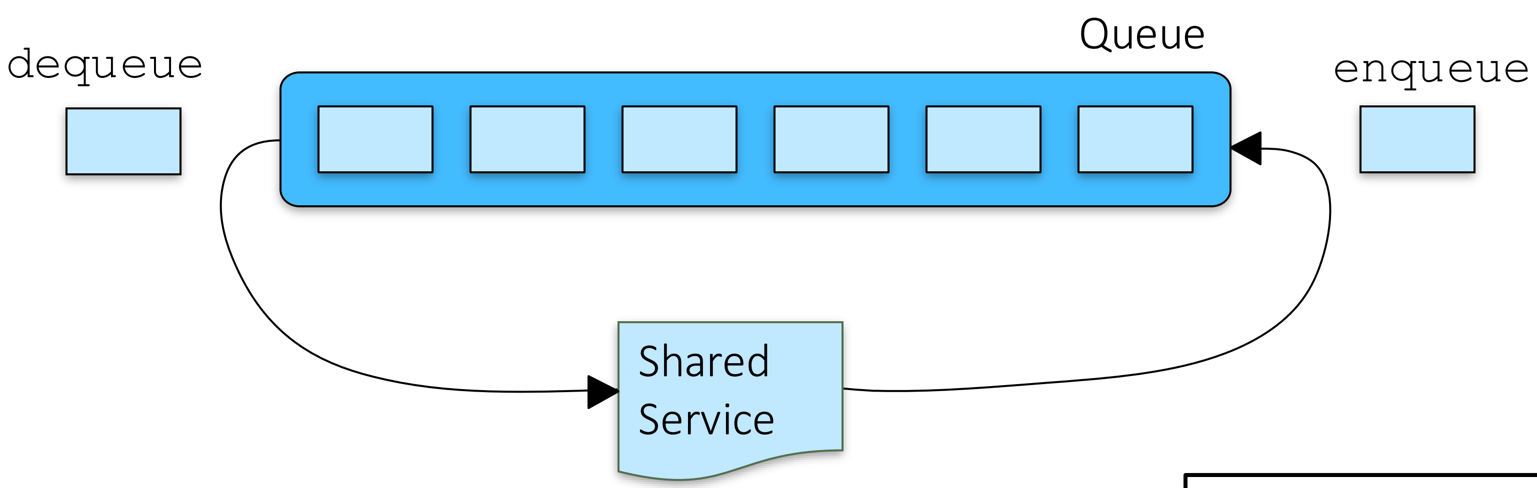
| | Access/Update | Search | Insertion `enqueue()` | Deletion `dequeue()` |
|---|---|---|---|---|
| Queue | $O(n)$ | $O(n)$ | $O(1)$ | $O(1)$ |

# Example: Round Robin Schedulers

- **Round-robin** (RR) is one of the algorithms employed by process and network schedulers in computing
  - To schedule processes fairly, a round-robin scheduler generally employs time-sharing, giving each job a time slot or *quantum* (its allowance of CPU time), and interrupting the job if it is not completed by then
  - The job is resumed next time a time slot is assigned to that process. If the process terminates or changes its state to waiting during its attributed time quantum, the scheduler selects the first process in the ready queue to execute. In the absence of time-sharing, or if the quanta were large relative to the sizes of the jobs, a process that produced large jobs would be favored over other processes

# Example: Round Robin Schedulers (cont'd.)

- We can implement a round robin scheduler using a queue *Q* by repeatedly performing the following steps:
  - `e = Q.dequeue()`
  - Service element *e*
  - `Q.enqueue(e)`




**RoundRobin.java**

# Exercise 5.2: Website Hit Counter

- Assume you are designing a website, your boss asks you to design a program named `HitCounter` that can count how many hits received in the past minute. You may assume that each hit comes with a timestamp, hits are received in a chronological order, i.e., the timestamp is monotonically increasing, the earliest timestamp starts at 1, and no hits arrive at the same time

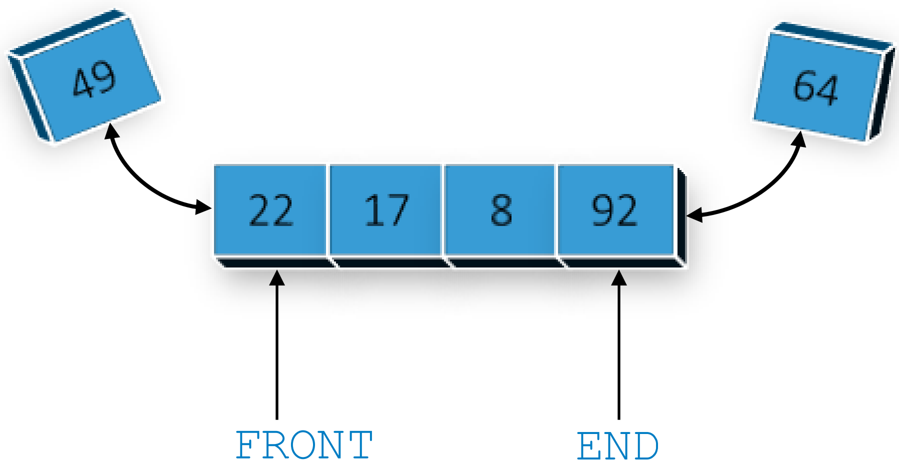
**`HitCounter.java`**

# Applications of Queues

- Direct applications
  - Waiting lists, bureaucracy
  - Access to shared resources (e.g., printer)
  - Multiprogramming
- Indirect applications
  - Auxiliary data structure for algorithms
  - Component of other data structures

# Deque: Double-ended Queue

- A deque is a collection of objects that can be inserted or deleted at both the front and the end of the queue



# Deque Methods

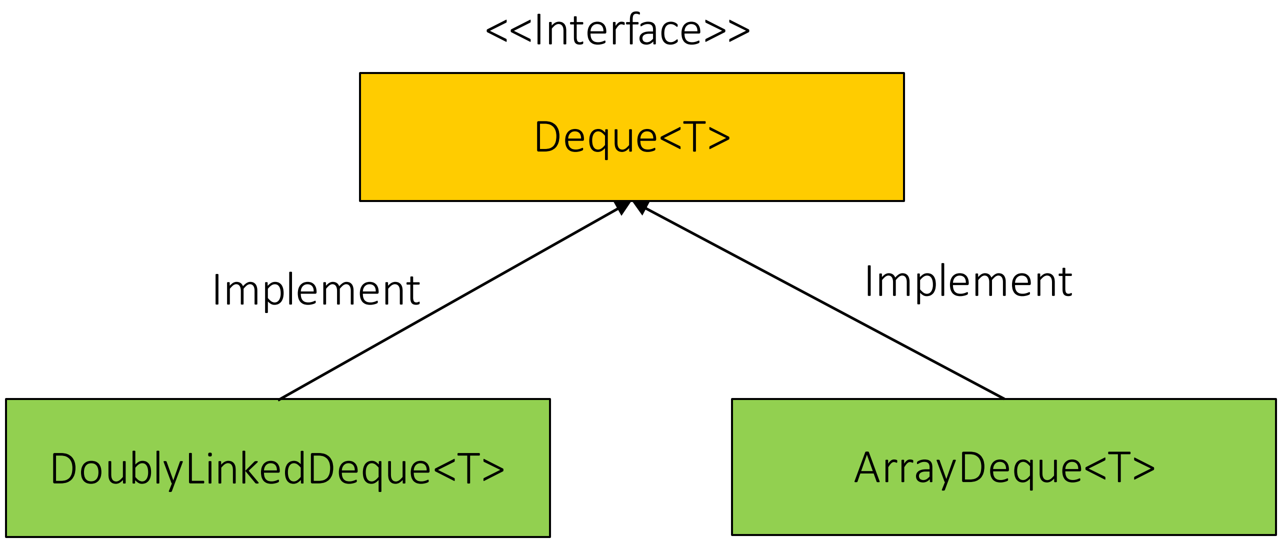
| Method | Description |
|---|---|
| `size()` | Returns the number of elements in the deque |
| `isEmpty()` | Returns a boolean indicating whether the deque is empty |
| `first()` | Returns the first element of the deque, without removing it or `null` if the deque is empty |
| `last()` | Returns the last element of the deque, without removing it or `null` if the deque is empty |

# Deque Methods (cont'd.)

| Method | Description |
|---|---|
| `addFirst(e)` | Inserts a new element *e* at the front of the deque |
| `addLast(e)` | Inserts a new element *e* at the back of the deque |
| `removeFirst()` | Removes and returns the first element of the deque, or `null` if the deque is empty |
| `removeLast()` | Removes and returns the last element of the deque, or `null` if the deque is empty |

# Deque Implementation

- Doubly linked list-based implementation
- Circular array-based implementation

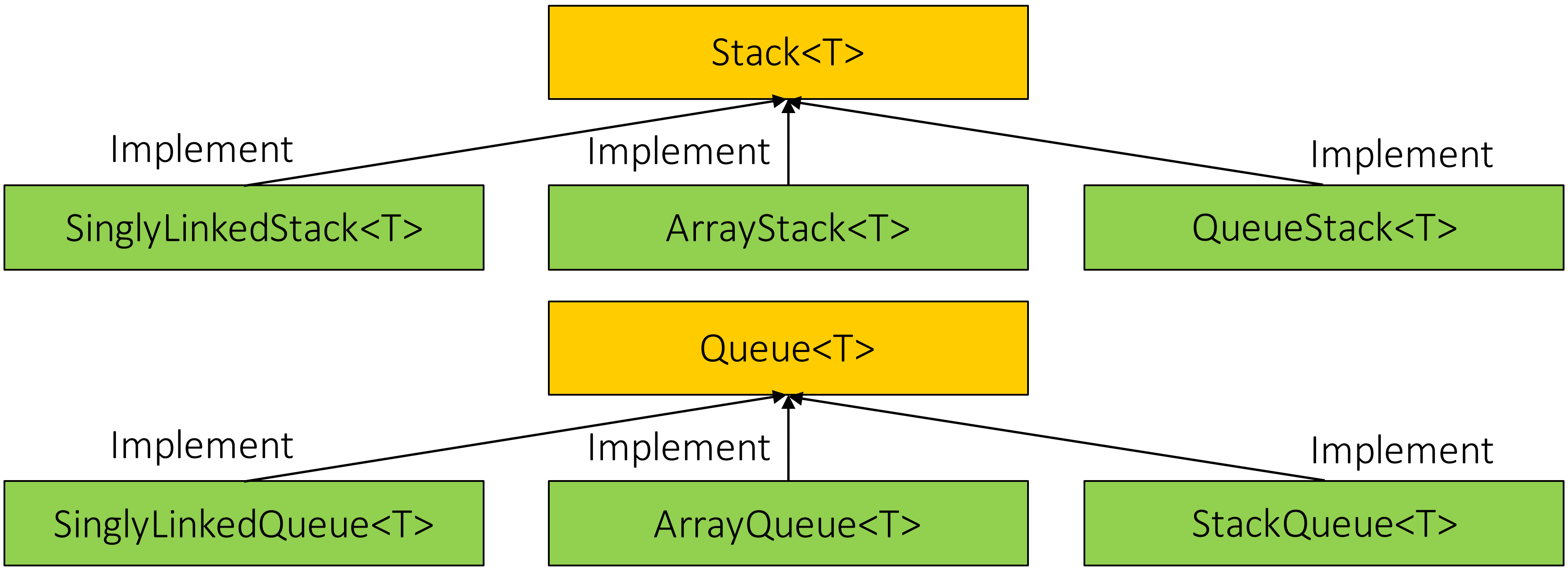



**`DoublyLinkedDeque.java`**

**`ArrayDeque.java`**

# Exercise 5.3

- How to implement a queue with stacks ?
- How to implement a stack with queues ?




**QueueStack.java**

**StackQueue.java**

# Summary

- The Stack ADT
  - ADT
  - Implementation
    - Singly linked list-based implementation
    - Array-based implementation
    - Queue-based implementation
  - Applications
    - Sequence order reversing
    - Simple calculator design

# Summary (cont'd.)

- The Queue ADT
  - ADT
  - Implementation
    - Singly linked list-based implementation
    - Circular array-based implementation
    - Stack-based implementation
  - Application
    - Round robin scheduler
    - Website hit counter
- The Deque ADT