2. We take as starting point the implementation of a bounded queue using an array of size \(n\) as discussed on the slides.

\(\text{size}()\) can be implemented by

\[
\text{return } r - h
\]

\(\text{isEmpty}()\) can be implemented by

\[
\begin{align*}
\text{if } h = r & \text{ then} \\
\text{return } true \\
\text{else} & \\
\text{return } false
\end{align*}
\]

\(\text{head}()\) can be implemented by

\[
\begin{align*}
\text{if } h < r & \text{ then} \\
\text{return } A[h \mod n] \\
\text{else} & \\
\text{throw } EmptyQueueException
\end{align*}
\]

3. Suppose we have two unbounded stacks \(S_1\) and \(S_2\), with operations \textit{push} and \textit{pop} in \(O(1)\). The queue (from head to tail) consists of the elements on \(S_2\) (from top to bottom) and then in reverse order the elements on \(S_1\) (from bottom to top).

The operation \textit{enqueue}(e) simply pushes \(e\) onto \(S_1\):

\[
\text{Algorithm } \textit{enqueue}(e) \\
S_1.\text{push}(e)
\]

The worst-case time complexity of this operation is in \(O(1)\).

The idea for \textit{dequeue}() is as follows.

- If \(S_2\) is empty, then one by one each element is popped from \(S_1\) and pushed onto \(S_2\), until \(S_1\) is empty.
- The top element of \(S_2\) is popped and returned as head of the queue.
Algorithm `dequeue()`
if $S_1$.isEmpty() and $S_2$.isEmpty() then
    throw EmptyQueueException
if $S_2$.isEmpty() then
    while not $S_1$.isEmpty() do
        $S_2$.push($S_1$.pop())
return $S_2$.pop()

The time complexity of this is operation is worst-case in $O(n)$, but on average in $O(1)$. Namely, each element is pushed onto and popped from $S_1$ exactly once, and also pushed onto and popped from $S_2$ exactly once.

4. Suppose we have two unbounded queues $Q_1$ and $Q_2$, with operations `enqueue` and `dequeue` in $O(1)$. The nonempty stack (from top to bottom) consists of the elements on $Q_1$ in reverse order (from tail to head) followed by the elements on $Q_2$ in reverse order (from tail to head).

The operation `push(e)` simply enqueues $e$ onto $Q_1$:

Algorithm `push(e)`
$Q_1$.enqueue($e$)

The worst-case time complexity of this operation is in $O(1)$.

The idea for `pop()` is as follows.

- If $Q_1$ is nonempty, then one by one each element except the tail is dequeued from $Q_1$ and enqueued onto $Q_2$. Finally, the tail of $Q_1$ is dequeued and returned as top of the stack.

- If $Q_1$ is empty, then one by one each element except the tail is dequeued from $Q_2$ and enqueued onto $Q_1$. Finally, the tail of $Q_2$ is dequeued and returned as top of the stack.

Algorithm `pop()`
if $Q_1$.isEmpty() and $Q_2$.isEmpty() then
    throw EmptyStackException
if not $Q_1$.isEmpty() then
    while $Q_1$.size() > 1 do
        $Q_2$.enqueue($Q_1$.dequeue())
    return $Q_1$.dequeue()
while $Q_2$.size() > 1 do
    $Q_1$.enqueue($Q_2$.dequeue())
return $Q_2$.dequeue()

The worst-case time complexity of this operation is in $O(n)$.

**Remark:** An alternative solution is to always place the top element of the stack at the head of $Q_1$, and keep $Q_2$ empty. Then a `pop()` is simply a dequeue on $Q_1$. (If $Q_1$
is empty, an empty exception is thrown.) This takes \(O(1)\). A \textit{push}(d) first one by one dequeues each element from \(Q_1\) and enqueues it on \(Q_2\), until \(Q_1\) is empty. Then it enqueues \(d\) on \(Q_1\). Finally, it one by one dequeues each element from \(Q_2\) and enqueues it on \(Q_1\), until \(Q_2\) is empty. This takes \(O(n)\) on a stack of \(n\) elements.

5. \textit{inList}(v) can be performed on a singly-linked list as follows.

```java
var w : node
w ← first
while w ≠ null do
  if w = v then
    return true
  w ← w.next
return false
```

\textit{inList}(v) can be performed on a doubly-linked list in a similar fashion (using \textit{header.next} instead of \textit{first}).

To perform \textit{insertBefore}(v, d) on a singly-linked list, we introduce a help function \textit{predecessor}(v) which returns the predecessor node of \(v\) in a list (if there is no predecessor it returns \textit{null}).

```java
var w1, w2 : node
w1 ← null
w2 ← first
while w2 ≠ null do
  if w2 = v then
    return w1
  w1 ← w2
  w2 ← w2.next
return null
```

\textit{insertBefore}(v, d) can now be performed on a singly-linked list as follows.

```java
var x : node
if inList(v) returns false then
  throw NodeAbsentException
new w : node
w.element ← d
w.next ← v
x ← predecessor(v)
if x ≠ null then
  x.next ← w
else
  first ← w
```

\textit{insertBefore}(v, d) can be performed on a doubly-linked list as follows (without the help function \textit{predecessor}(v)).
if inList(v) returns false then
    throw NodeAbsentException
new w : node
w.element ← d
w.next ← v
if v.pred ≠ null then
    v.pred.next ← w
    w.pred ← v.pred
else
    header.next ← w
    w.pred ← null
v.pred ← w

var pointer1, pointer2, temp : node
pointer1 ← null
pointer2 ← first
while pointer2 ≠ null do
    temp ← pointer2.next
    pointer2.next ← pointer1
    pointer1 ← pointer2
    pointer2 ← temp
    temp ← first
    first ← last
    last ← temp

Remark: In case of a doubly-linked list, one wouldn’t need to use the auxiliary temp variable within the while loop to remember the original value of pointer2.next.