CSC316

Data Structures & Algorithms

Stack Abstract Data Type

Dr. Jason King

DEFINITION

Stack

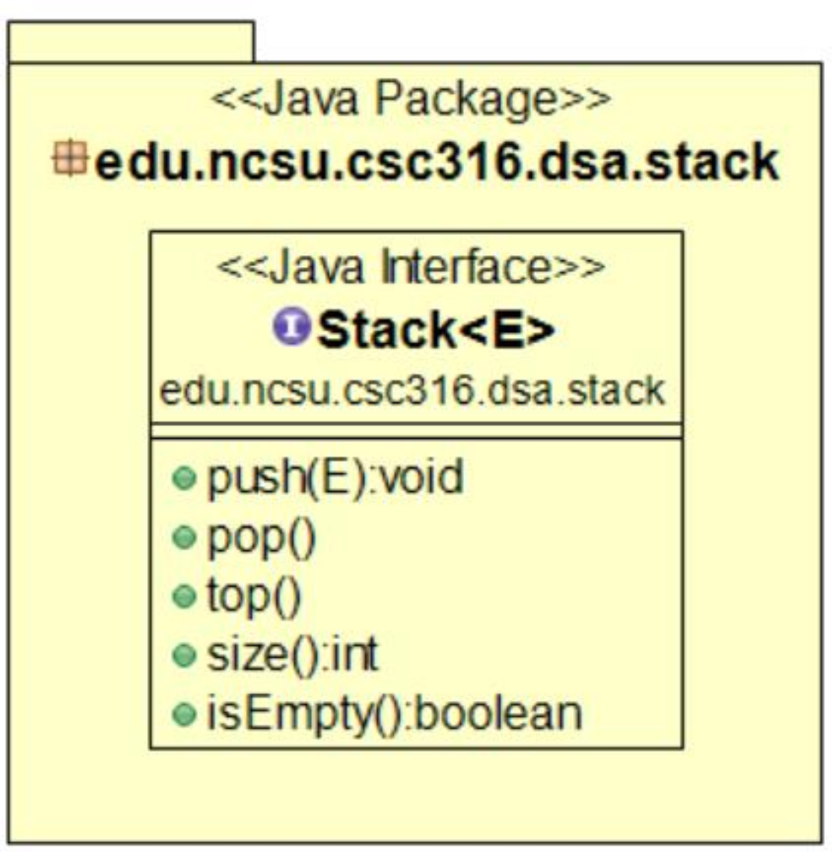
- Collection of data, where the last value inserted is the first value removed
- LIFO (last-in-first-out)
- Values can only be inserted or removed from the top of the stack

Stack

ABSTRACT DATA TYPE

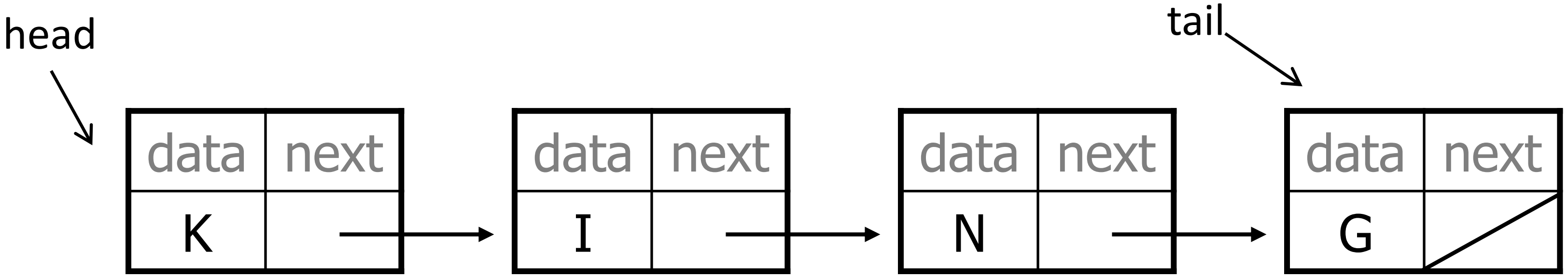
Operation	Description
push(e)	Adds the element to the top of the stack
pop()	Removes and returns the element at the top of the stack
top()	Returns, but does not remove, the element at the top of the stack
size()	Returns the number of elements in the stack
isEmpty()	Returns true if the stack is empty; otherwise, returns false

ABSTRACT DATA TYPE



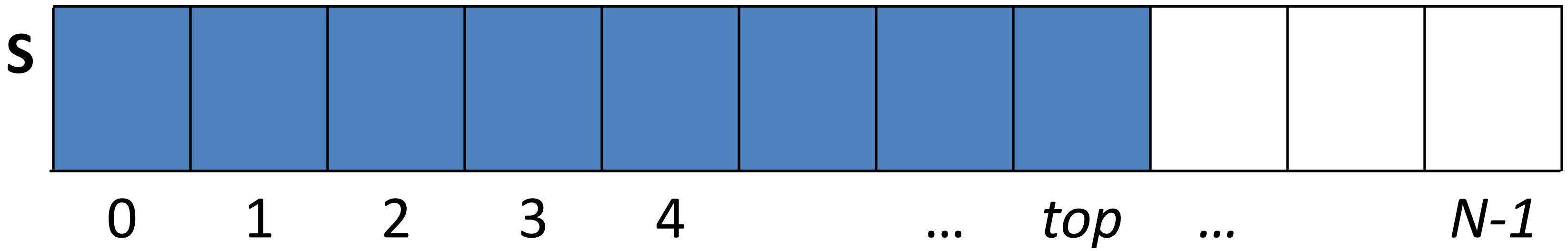
Stack with a Singly Linked List

- Top element stored at the head of the list
- **push(o)**: insert at the head →
- **pop()**: remove at the head →



Array-based Stacks

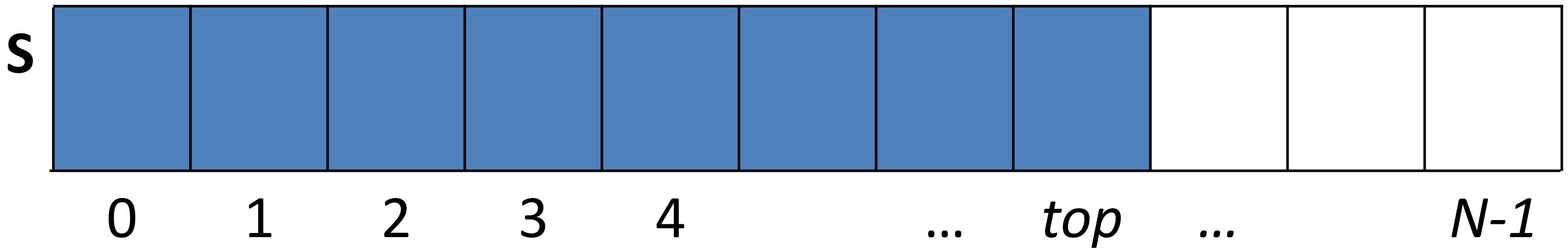
- Use an array of size *N*
- Add elements from left to right
  - Why?
- Variable *top* keeps track of the index of the top element



Array-based Stack Operations

Given:

- an array *S* of capacity *N*
- *top* is the index of the top element



Write an algorithm for each:

- popping an element from the stack
- pushing an element onto the stack

Array-based Stack Pop & Push Algorithms

```
Algorithm pop(S, top)
    Input an array S of capacity N
          the index top of the top element in the stack
    Output the element at the top of the stack
    if top = -1 then
        return “Empty Stack”
    else
        top ← top - 1
        return S[top+1]

Algorithm push(S, top, o)
    Input an element o to be added to the stack
          an array S of capacity N
          the index top of the top element in the stack
    if top + 1 = N
        return “Full Stack“
    else
        top ← top + 1
        S[top] ← o
```

Performance

Given *n* is the number of elements in the stack

- How much space is used?
  - O(n)
- What is the runtime complexity of the `push()` and `pop()` algorithms?
  - O(1)

QUESTION

What are the limitations of using an array-based stack?

Limitations

- The maximum size of the stack is defined beforehand and cannot change
  - How can you mitigate this limitation?

- Trying to push to a full stack generates an implementation-specific exception
  - How can you mitigate this limitation?

REVIEW

Operation	Array-based Stack	Singly Linked Stack	Circularly Linked Stack	Doubly Linked Stack	Positional Linked Stack
push(e)					
pop()					
top()					
size()					
isEmpty()					

** assuming size is maintained as a field*

REVIEW

Operation	Array-based Stack	Singly Linked Stack	Circularly Linked Stack	Doubly Linked Stack	Positional Linked Stack
push(e)	O(1)*				
pop()	O(1)*				
top()	O(1)*				
size()	O(1)*				
isEmpty()	O(1)				

Assuming we push to the end of the array

** assuming size is maintained as a field*

REVIEW

Operation	Array-based Stack	Singly Linked Stack	Circularly Linked Stack	Doubly Linked Stack	Positional Linked Stack
push(e)	O(1)*	O(1)			
pop()	O(1)*	O(1)			
top()	O(1)*	O(1)			
size()	O(1)*	O(1)*			
isEmpty()	O(1)	O(1)			

Assuming we push to the front of the list

** assuming size is maintained as a field*

REVIEW

Operation	Array-based Stack	Singly Linked Stack	Circularly Linked Stack	Doubly Linked Stack	Positional Linked Stack
push(e)	O(1)*	O(1)	O(1)		
pop()	O(1)*	O(1)	O(1)		
top()	O(1)*	O(1)	O(1)		
size()	O(1)*	O(1)*	O(1)*		
isEmpty()	O(1)	O(1)	O(1)		

** assuming size is maintained as a field*

REVIEW

Operation	Array-based Stack	Singly Linked Stack	Circularly Linked Stack	Doubly Linked Stack	Positional Linked Stack
push(e)	O(1)*	O(1)	O(1)	O(1)	
pop()	O(1)*	O(1)	O(1)	O(1)	
top()	O(1)*	O(1)	O(1)	O(1)	
size()	O(1)*	O(1)*	O(1)*	O(1)*	
isEmpty()	O(1)	O(1)	O(1)	O(1)	

** assuming size is maintained as a field*

REVIEW

Operation	Array-based Stack	Singly Linked Stack	Circularly Linked Stack	Doubly Linked Stack	Positional Linked Stack
push(e)	O(1)*	O(1)	O(1)	O(1)	O(1)
pop()	O(1)*	O(1)	O(1)	O(1)	O(1)
top()	O(1)*	O(1)	O(1)	O(1)	O(1)
size()	O(1)*	O(1)*	O(1)*	O(1)*	O(1)*
isEmpty()	O(1)	O(1)	O(1)	O(1)	O(1)

** assuming size is maintained as a field*

REVIEW

Operation	Array-based Stack	Singly Linked Stack	Circularly Linked Stack	Doubly Linked Stack	Positional Linked Stack
push(e)	O(1)*	O(1)	O(1)	O(1)	O(1)
pop()	O(1)*	O(1)	O(1)	O(1)	O(1)
top()	O(1)*	O(1)	O(1)	O(1)	O(1)
size()	O(1)*	O(1)*	O(1)*	O(1)*	O(1)*
isEmpty()	O(1)	O(1)	O(1)	O(1)	O(1)

** assuming size is maintained as a field*

Queue ADT

DEFINITION

Queue

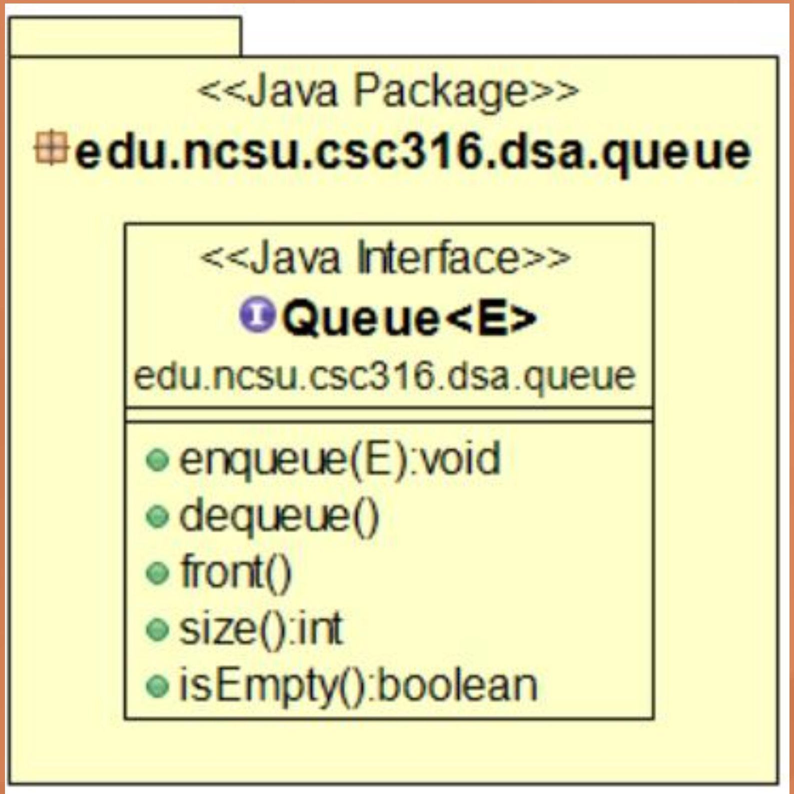
- Collection of data, where the first value inserted is the first value removed
- FIFO (first-in-first-out)
- Values can only be inserted at the end of the queue, or removed from the front of the queue

Queue

ABSTRACT DATA TYPE

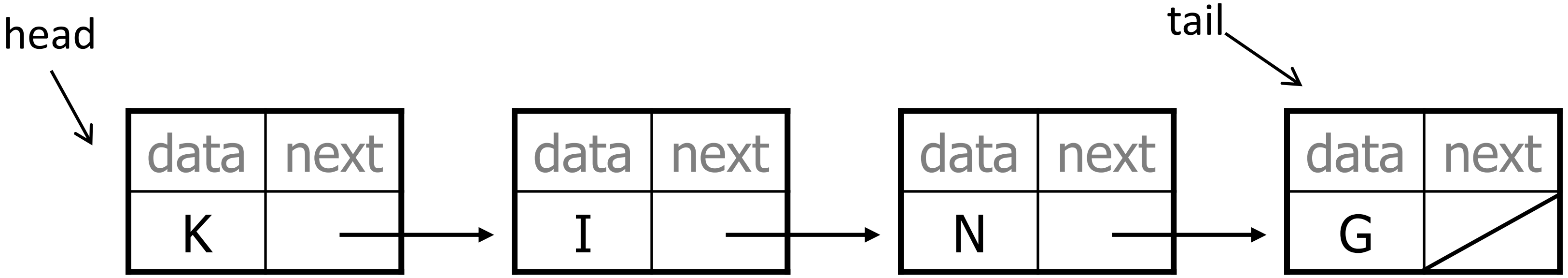
Operation	Description
enqueue(e)	Adds the element to the back of the queue
dequeue()	Removes and returns the element at the front of the queue
front()	Returns, but does not remove, the element at the front of the queue
size()	Returns the number of elements in the queue
isEmpty()	Returns true if the queue is empty; otherwise, returns false

ABSTRACT DATA TYPE



Queue with a Singly Linked List

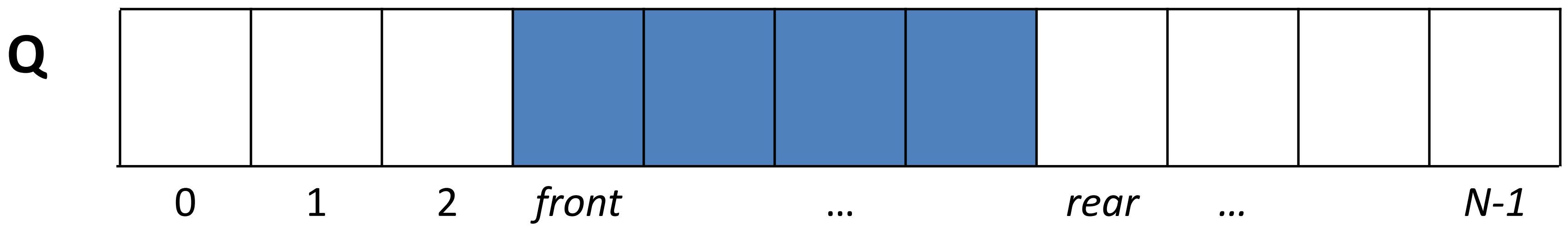
- Use head and tail pointers
- **enqueue(o)**: insert at the tail →
- **dequeue()**: remove at the front →



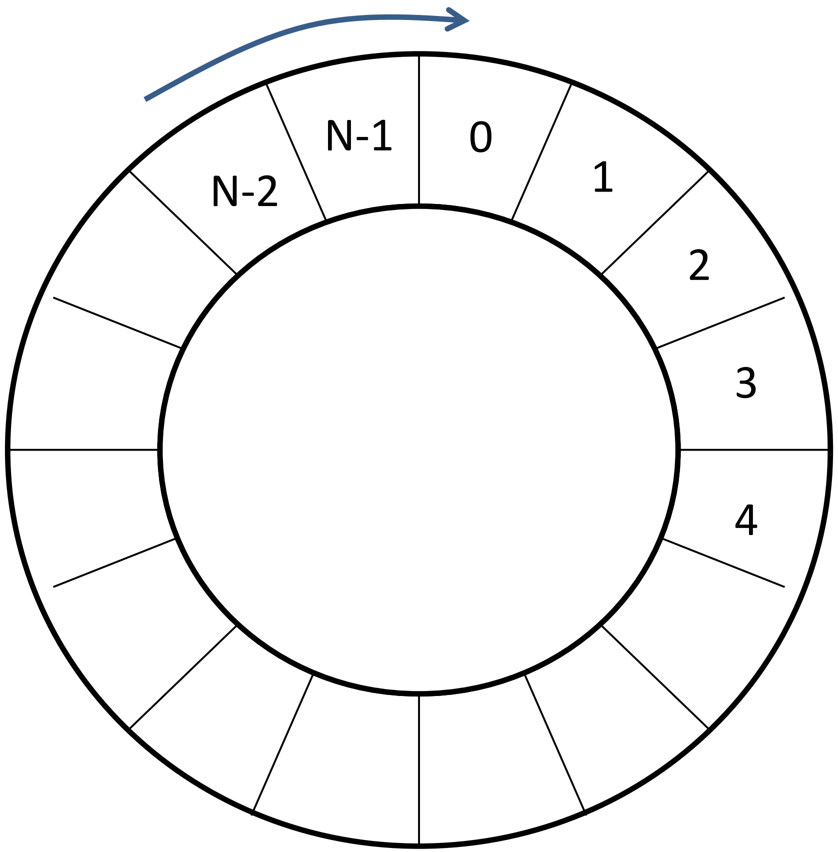
Array-based Queue Implementation

- Use an array of size *N*
  - Use the array in a **<u>circular</u>** fashion
- Keep track of the front and rear of the queue
  - *front*: index of the front element
  - *rear*: index <u>immediately past</u> the last element in the queue
- Keep *rear* empty →maximum number of values in the queue is N-1

Array-based Queue Implementation



Circular Buffer

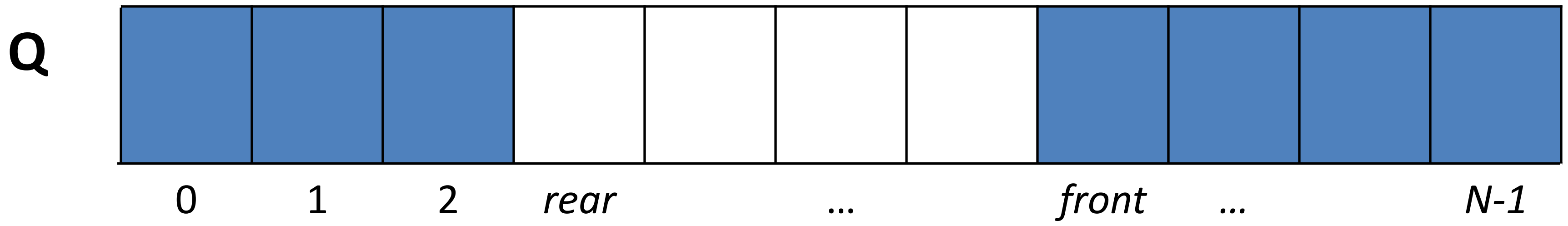


- Use **mod N** arithmetic to address
- Positions 0 and N-1 are adjacent

Array-based Queue Algorithms

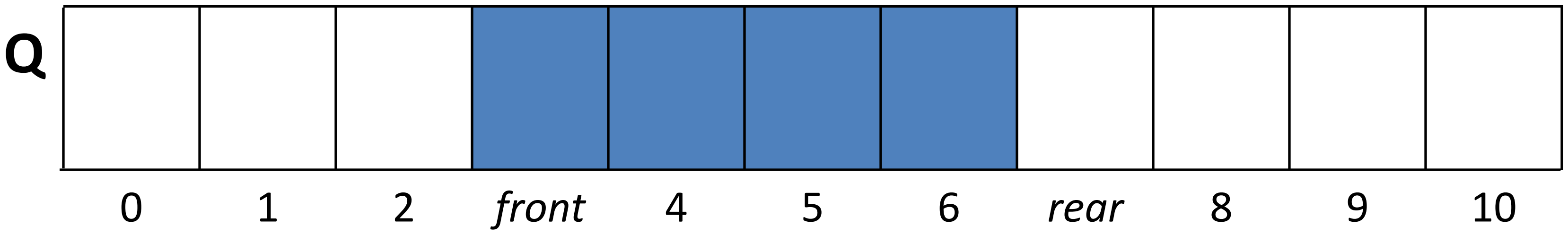
Given:

- an array *Q* of capacity *N*
- *front*, the index of the first element in the queue
- *rear*, the index immediately after the last element in the queue



Array-based Queue Dequeue

```
Algorithm dequeue(Q, front)
   Input an array Q of capacity N
         the index of the front element
   Output the element at the front of the queue
   if isEmpty() then
      return "Empty Queue"
   else
      o ← Q[front]
      front ← (front + 1) mod N
      return o
```



QUESTION

Can you resize an array-based circular buffer?

IMPORTANT

You **cannot** resize an array-based circular buffer like you resize an array-based list.

→ What if the elements “wrap around” the end of the array?

Instead, to resize an array-based circular buffer:

→ Create a larger array

→ Copy elements into the larger array

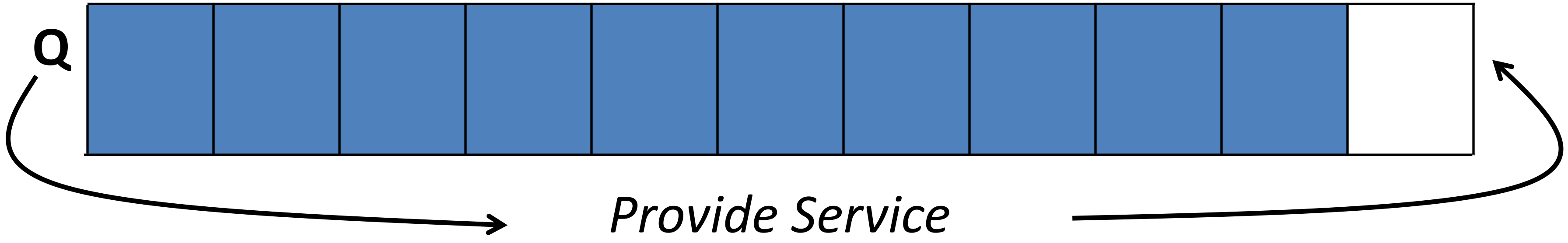
  → The element at “front” in the old array should go into the new array at index 0

→ Update “front” and “rear” to reflect the new array

Application: Round-Robin Scheduler

Use a queue Q and repeatedly perform these steps:

1. *p = Q.dequeue()*
2. Service process *p*
3. If process is not complete, *Q.enqueue(p)*



REVIEW

Operation	Array-based Queue	Singly Linked Queue	Circularly Linked Queue	Doubly Linked Queue	Positional Linked Queue
enqueue(e)					
dequeue()					
front()					
size()					
isEmpty()					

** assuming size is maintained as a field*

REVIEW

Operation	Array-based Queue	Singly Linked Queue	Circularly Linked Queue	Doubly Linked Queue	Positional Linked Queue
enqueue(e)	O(1)				
dequeue()	O(1)				
front()	O(1)				
size()	O(1)*				
isEmpty()	O(1)				

Assuming we implement as a circular buffer

** assuming size is maintained as a field*

REVIEW

Operation	Array-based Queue	Singly Linked Queue	Circularly Linked Queue	Doubly Linked Queue	Positional Linked Queue
enqueue(e)	O(1)	O(1)			
dequeue()	O(1)	O(1)			
front()	O(1)	O(1)			
size()	O(1)*	O(1)*			
isEmpty()	O(1)	O(1)			

Assuming we enqueue at the end of the list & the list has a tail pointer

** assuming size is maintained as a field*

REVIEW

Operation	Array-based Queue	Singly Linked Queue	Circularly Linked Queue	Doubly Linked Queue	Positional Linked Queue
enqueue(e)	O(1)	O(1)	O(1)		
dequeue()	O(1)	O(1)	O(1)		
front()	O(1)	O(1)	O(1)		
size()	O(1)*	O(1)*	O(1)*		
isEmpty()	O(1)	O(1)	O(1)		

** assuming size is maintained as a field*

REVIEW

Operation	Array-based Queue	Singly Linked Queue	Circularly Linked Queue	Doubly Linked Queue	Positional Linked Queue
enqueue(e)	O(1)	O(1)	O(1)	O(1)	
dequeue()	O(1)	O(1)	O(1)	O(1)	
front()	O(1)	O(1)	O(1)	O(1)	
size()	O(1)*	O(1)*	O(1)*	O(1)*	
isEmpty()	O(1)	O(1)	O(1)	O(1)	

** assuming size is maintained as a field*

REVIEW

Operation	Array-based Queue	Singly Linked Queue	Circularly Linked Queue	Doubly Linked Queue	Positional Linked Queue
enqueue(e)	O(1)	O(1)	O(1)	O(1)	O(1)
dequeue()	O(1)	O(1)	O(1)	O(1)	O(1)
front()	O(1)	O(1)	O(1)	O(1)	O(1)
size()	O(1)*	O(1)*	O(1)*	O(1)*	O(1)*
isEmpty()	O(1)	O(1)	O(1)	O(1)	O(1)

** assuming size is maintained as a field*

REVIEW

Operation	Array-based Queue	Singly Linked Queue	Circularly Linked Queue	Doubly Linked Queue	Positional Linked Queue
enqueue(e)	O(1)	O(1)	O(1)	O(1)	O(1)
dequeue()	O(1)	O(1)	O(1)	O(1)	O(1)
front()	O(1)	O(1)	O(1)	O(1)	O(1)
size()	O(1)*	O(1)*	O(1)*	O(1)*	O(1)*
isEmpty()	O(1)	O(1)	O(1)	O(1)	O(1)

** assuming size is maintained as a field*

Documenting & Analyzing
Algorithms that use Data Structures

Document Algorithms

- If your algorithm needs to interact with a data structure
  - Your algorithm should **use the ADT operations**!
- When you analyze your algorithm, you can examine tradeoffs in performance of different data structures

Example: Document Algorithm

Write an algorithm `reverse` that takes an input list and reverses the elements in the list.

```
Algorithm reverse(L)
    Input a list L of n elements
    Output the list of elements in reverse order
    S ← empty stack
    while NOT L.isEmpty() do
        x ← L.remove(0)
        S.push(x)
    while NOT S.isEmpty() do
        L.addLast( S.pop() )
    return L
```

All of these are operations defined by the abstract data types!

Example: Analyze Algorithm

What is the asymptotic running time if:

- input list is an array-based list
- stack is an array-based stack with top at the end

Operation	Running Time
L.isEmpty()	O(1)
L.remove(0)	O(n)
S.push(x)	O(1)
S.isEmpty()	O(1)
S.pop()	O(1)
L.addLast(v)	O(1)

```
Algorithm reverse(L)
    Input a list L of n elements
    Output the list of elements in reverse order
    S ← empty stack
    while NOT L.isEmpty() do
        x ← L.remove(0)
        S.push(x)
    while NOT S.isEmpty() do
        L.addLast( S.pop() )
    return L
```

O(1) isEmpty + O(n) remove + O(1) push repeated O(n) times

O(1) isEmpty + O(1) addLast + O(1) pop repeated O(n) times

O(n^2)

Example: Analyze Algorithm

What is the asymptotic running time if:

- input list is linked list with pointer at head (no tail)
- stack is a linked stack with top of stack at tail

Operation	Running Time
L.isEmpty()	O(1)
L.remove(0)	O(1)
S.push(x)	O(n)
S.isEmpty()	O(1)
S.pop()	O(n)
L.addLast(v)	O(n)

```
Algorithm reverse(L)
    Input a list L of n elements
    Output the list of elements in reverse order
    S ← empty stack
    while NOT L.isEmpty() do
        x ← L.remove(0)
        S.push(x)
    while NOT S.isEmpty() do
        L.addLast( S.pop() )
    return L
```

O(1) isEmpty + O(1) remove + O(n) push repeated O(n) times

O(1) isEmpty + O(n) addLast + O(n) pop repeated O(n) times

$O(n^2)$

Example: Analyze Algorithm

What is the asymptotic running time if:

- input list is circularly linked list with tail pointer
- stack is a linked stack with top of stack at front

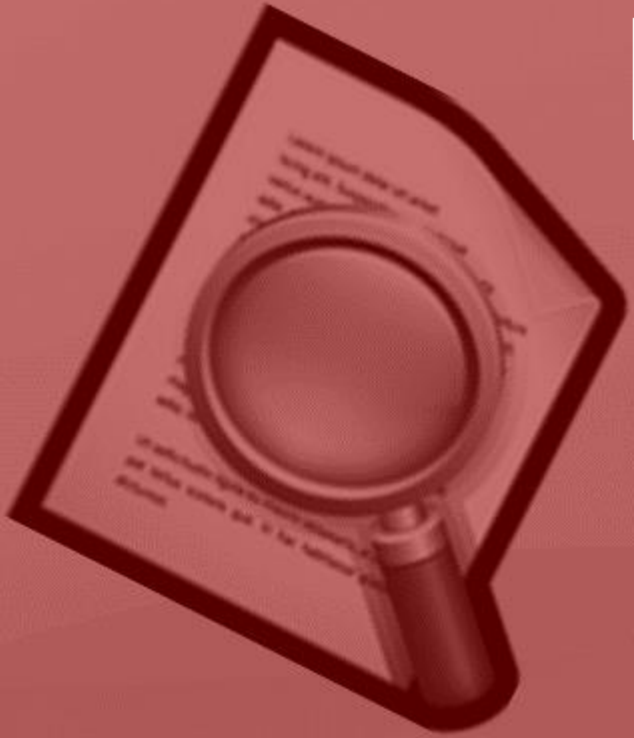
Operation	Running Time
L.isEmpty()	O(1)
L.remove(0)	O(1)
S.push(x)	O(1)
S.isEmpty()	O(1)
S.pop()	O(1)
L.addLast(v)	O(1)

```
Algorithm reverse(L)
    Input a list L of n elements
    Output the list of elements in reverse order
    S ← empty stack
    while NOT L.isEmpty() do
        x ← L.remove(0)
        S.push(x)
    while NOT S.isEmpty() do
        L.addLast( S.pop() )
    return L
```

O(1) isEmpty + O(1) remove + O(1) push repeated O(n) times

O(1) isEmpty + O(1) addLast + O(1) pop repeated O(n) times

O(n)



REVIEW

- Design and document your algorithms using abstract data types and their associated behaviors
- Select your data structure(s) to optimize algorithm runtime performance
- You may need to consider alternative data structure(s)