COMP 250

Lecture 18

queue ADT

ADT (abstract data type)

- List

  add(i,e), remove(i), get(i), set(i), …..

- Stack

  push, pop(), ..

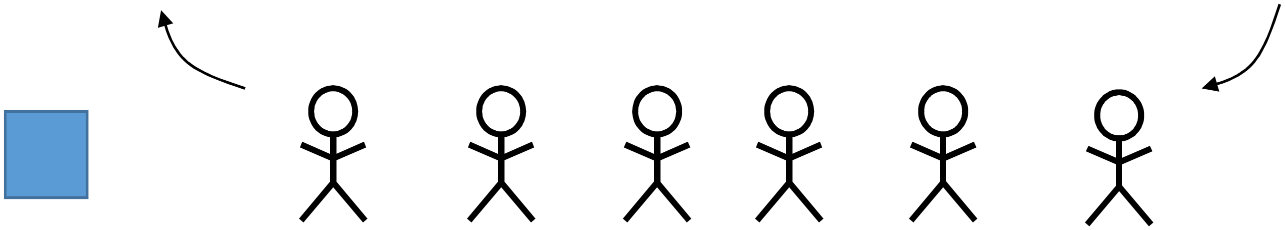
- Queue

  enqueue(e), dequeue()

Queue

dequeue
(remove from front)

enqueue
(add at back)



e.g. server

clients

Examples

- keyboard buffer

  (delay when you type)

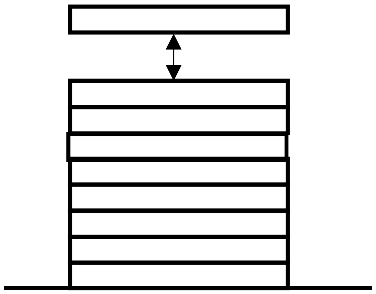
- CPU processes

  (different applications do not run in parallel; they line up and each gets a certain amount of time on the CPU and then they have to line up again)

- web server

  (many customers trying to access the same web site)

- cafeteria …

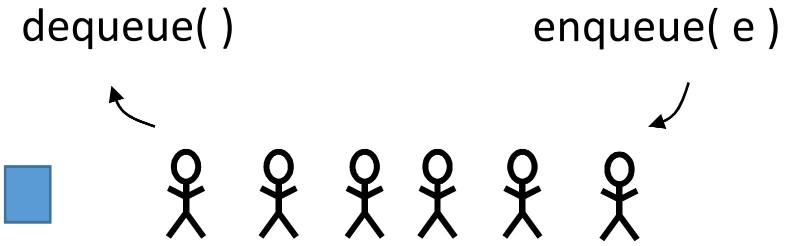


Stack

push(e)

pop()

LIFO
(last in, first out)



Queue

enqueue(e)

dequeue()

FIFO
(first in, first out)
“first come, first serve”

Queue Example

`enqueue( a )`	`a`	
`enqueue( b )`	`ab`	
`dequeue( )`	`b`	returns `a`

time

Queue Example

`enqueue( a )`	`a`	
`enqueue( b )`	`ab`	
`dequeue( )`	`b`	returns `a`
`enqueue( c )`	`bc`	
`enqueue( d )`	`bcd`	
`enqueue( e )`	`bcde`	

time

Queue Example

`enqueue( a )`	`a`
`enqueue( b )`	`ab`
`dequeue( )`	`b`
`enqueue( c )`	`bc`
`enqueue( d )`	`bcd`
`enqueue( e )`	`bcde`
`dequeue( )`	`?`
`enqueue( f )`	
`enqueue( g )`	

time

Queue Example

`enqueue( a )`	`a`	
`enqueue( b )`	`ab`	
`dequeue( )`	`b`	
`enqueue( c )`	`bc`	
`enqueue( d )`	`bcd`	
`enqueue( e )`	`bcde`	
`dequeue( )`	`cde`	**returns** `b`
`enqueue( f )`	`cdef`	
`enqueue( g )`	`cdefg`	

time

How to implement a queue?

	enqueue(e)	dequeue()
singly linked list	?	?
doubly linked list	?	?
array list	?	?

How to implement a queue?

	enqueue(e)	dequeue()
singly linked list	addLast(e)	removeFirst()
doubly linked list	?	?
array list	?	?

How to implement a queue?

	enqueue(e)	dequeue()
singly linked list	addLast(e)	removeFirst()
doubly linked list	same, or addFirst()	& removeLast ()
array list	?	?

How to implement a queue?

	enqueue(e)	dequeue()
singly linked list	addLast(e)	removeFirst()
doubly linked list	same, or addFirst()	& removeLast ()
array list*	addLast(e)	**removeFirst()**

(inefficient)

*arraylists generally don’t have addLast or removeFirst methods, but they have other methods that can do the same thing.

Implementing a queue with an array list. (inefficient)

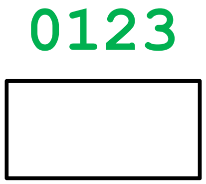
`0123` indices

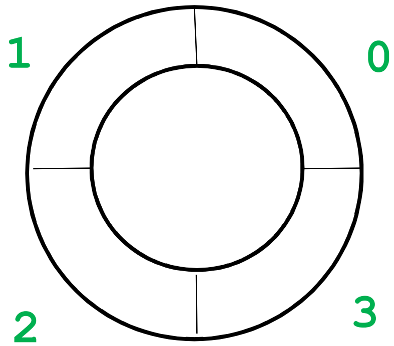
length = 4

`enqueue( a )`	`a---`	
`enqueue( b )`	`ab--`	
`dequeue( )`	`b---`	requires a shift
`enqueue( c )`	`bc--`	
`enqueue( d )`	`bcd-`	
`enqueue( e )`	`bcde`	
`dequeue( )`	`cde-`	requires a shift
`enqueue( f )`		
`enqueue( g )`		

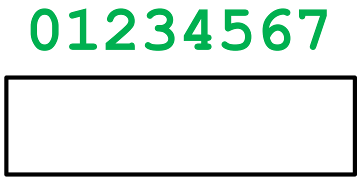
Circular array

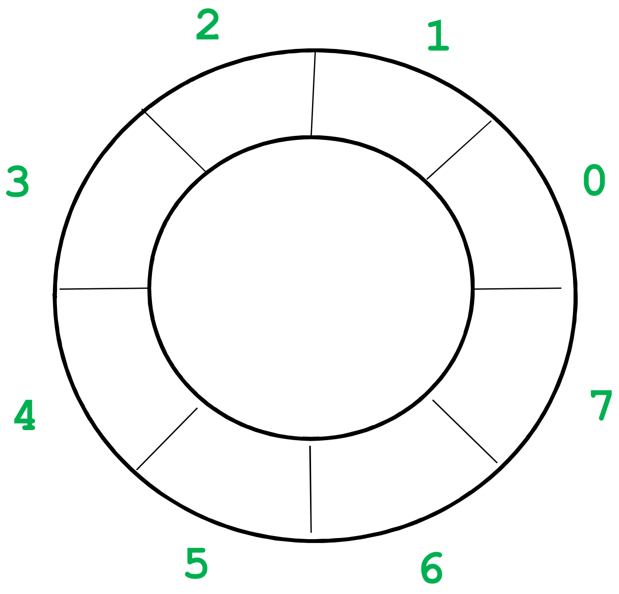
length = 4





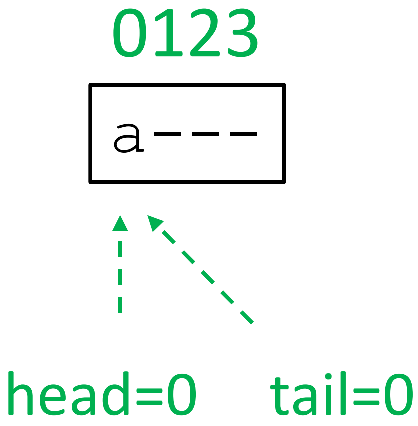
length = 8

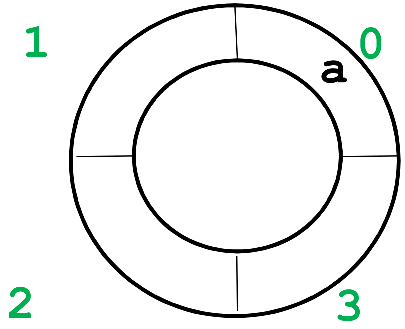




Circular array

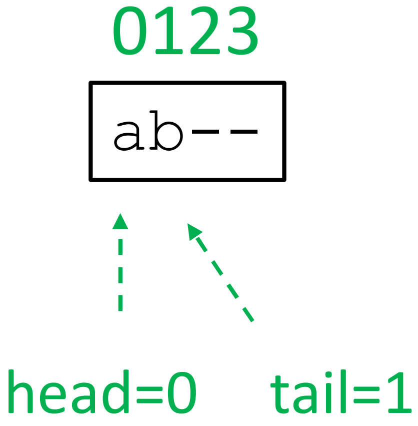
`enqueue( a )`





Circular array

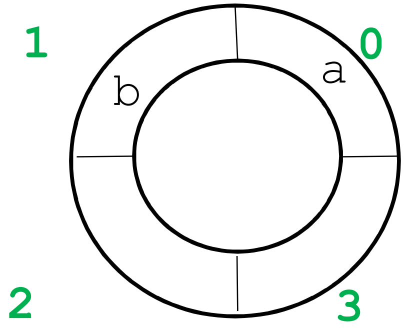
```
enqueue( a )
enqueue( b )
```



0123

`ab--`

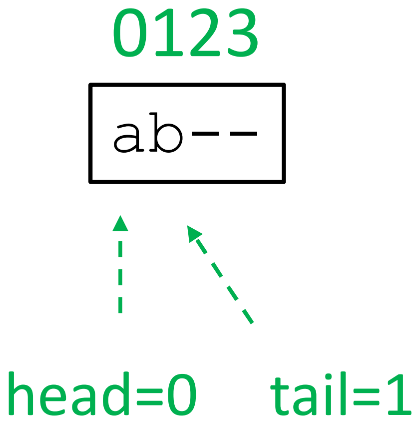
head=0 tail=1



Circular array

`enqueue( a )`
`enqueue( b )`

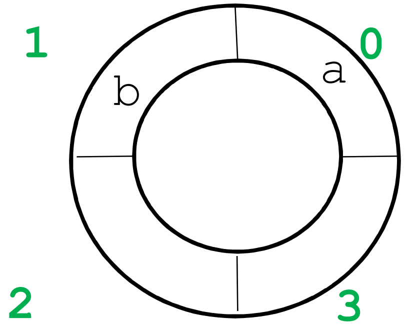
`dequeue()   ?`



0123

ab--

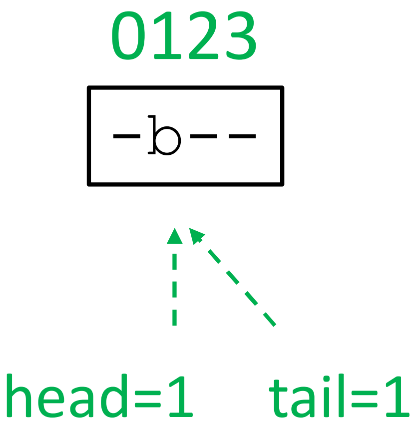
head=0 tail=1

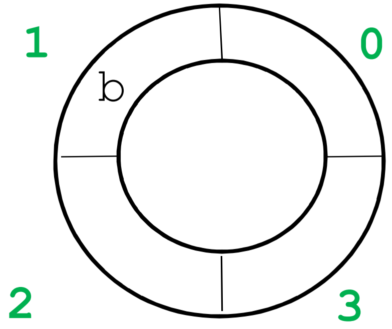


Circular array

```
enqueue( a )
enqueue( b )
dequeue()
```

`enqueue( c )`   **?**





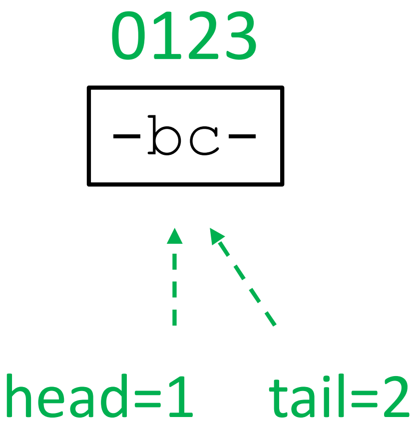
tail = (head + size – 1) mod length

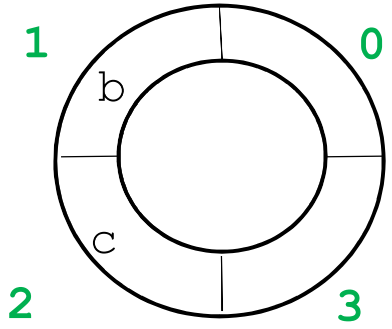
Note: if size = 1, then tail = head.

Circular array

```
enqueue( a )
enqueue( b )
dequeue()
enqueue( c )
```

`enqueue( d )` **?**





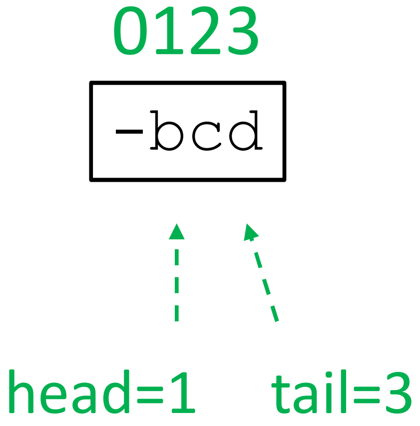
tail = (head + size – 1) mod length

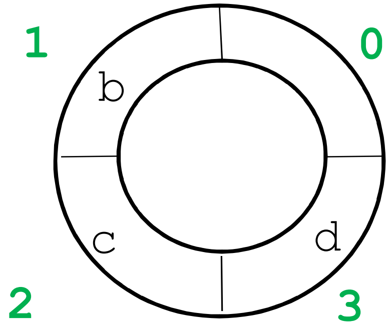
Note: if size = 1, then tail = head.

Circular array

```
enqueue( a )
enqueue( b )
dequeue()
enqueue( c )
enqueue( d )
```

enqueue(e)   ?

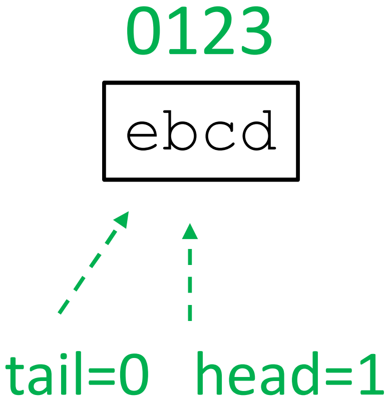


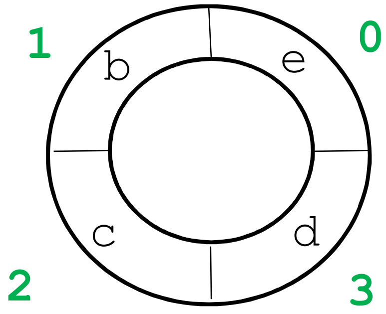


Circular array

```
enqueue( a )
enqueue( b )
dequeue( )
enqueue( c )
enqueue( d )
enqueue( e )
```

dequeue() ?



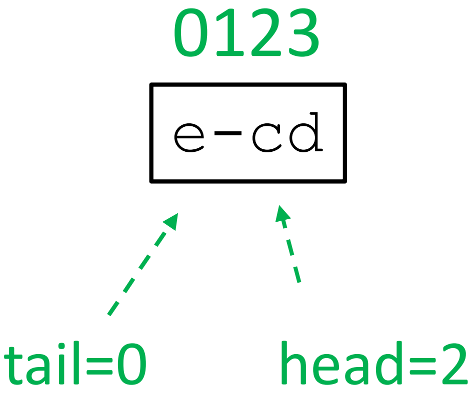


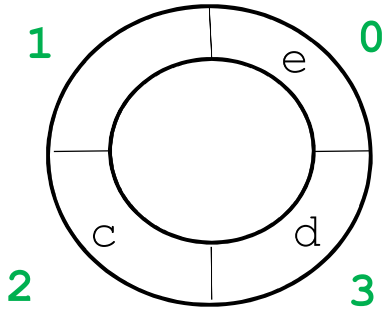
tail = (head + size – 1) **mod length**

Circular array

```
enqueue( a )
enqueue( b )
dequeue( )
enqueue( c )
enqueue( d )
enqueue( e )
dequeue()
```

`enqueue( f ) ?`

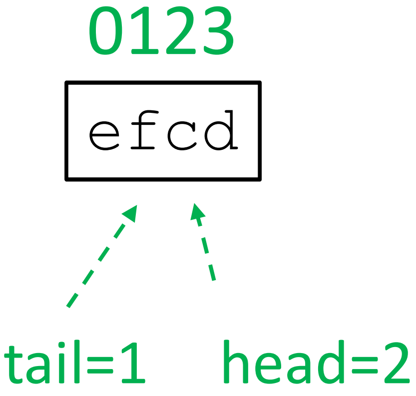


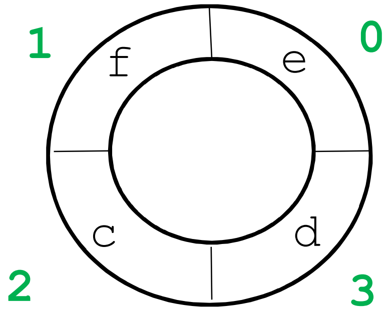


$$\text{tail} = (\text{head} + \text{size} - 1) \ \text{mod length}$$

Circular array

```
enqueue( a )
enqueue( b )
dequeue( )
enqueue( c )
enqueue( d )
enqueue( e )
dequeue()
enqueue( f )
```





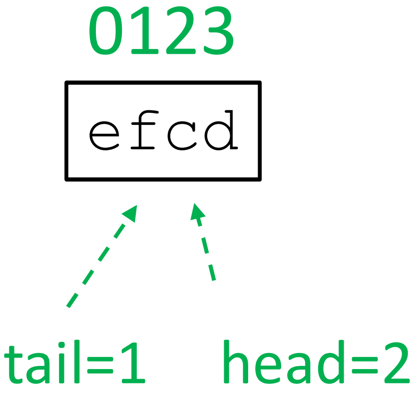
tail = (head + size – 1) mod length

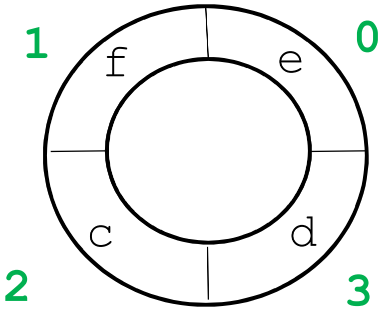
<u>NOTE:</u> When size = length, we have tail = (head – 1) mod length.

Circular array

```
enqueue( a )
enqueue( b )
dequeue( )
enqueue( c )
enqueue( d )
enqueue( e )
dequeue()
enqueue( f )
```

enqueue(g) ?

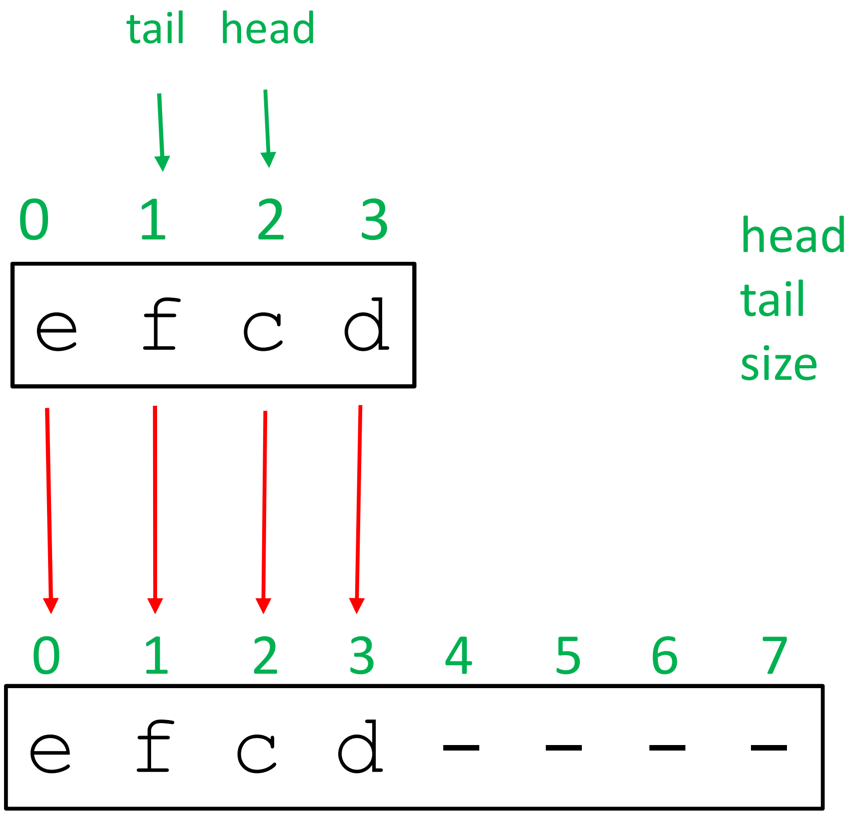




tail = (head + size – 1) mod length

How to `enqueue(g)` ?

Increase length of array and copy? **NO**



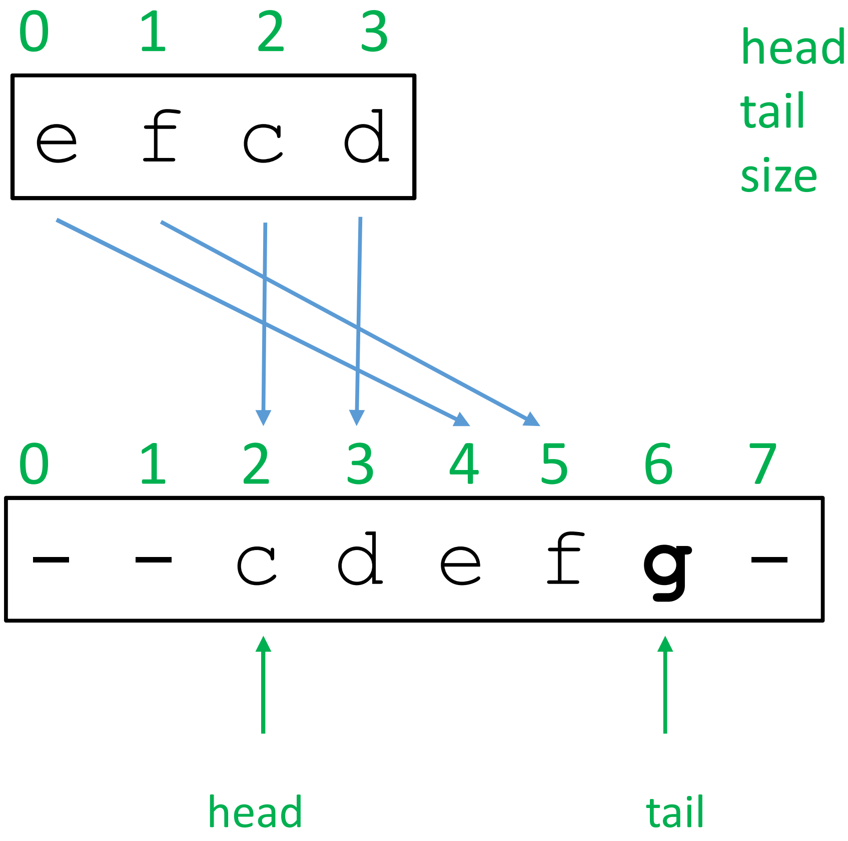
head = 2
tail = 1
size = 4

head = ?
tail = ?
size = 4

How to `enqueue(`**`g`**`)` ?

Increase length of array. Copy such that head stays as is.

Add the new element **g**.



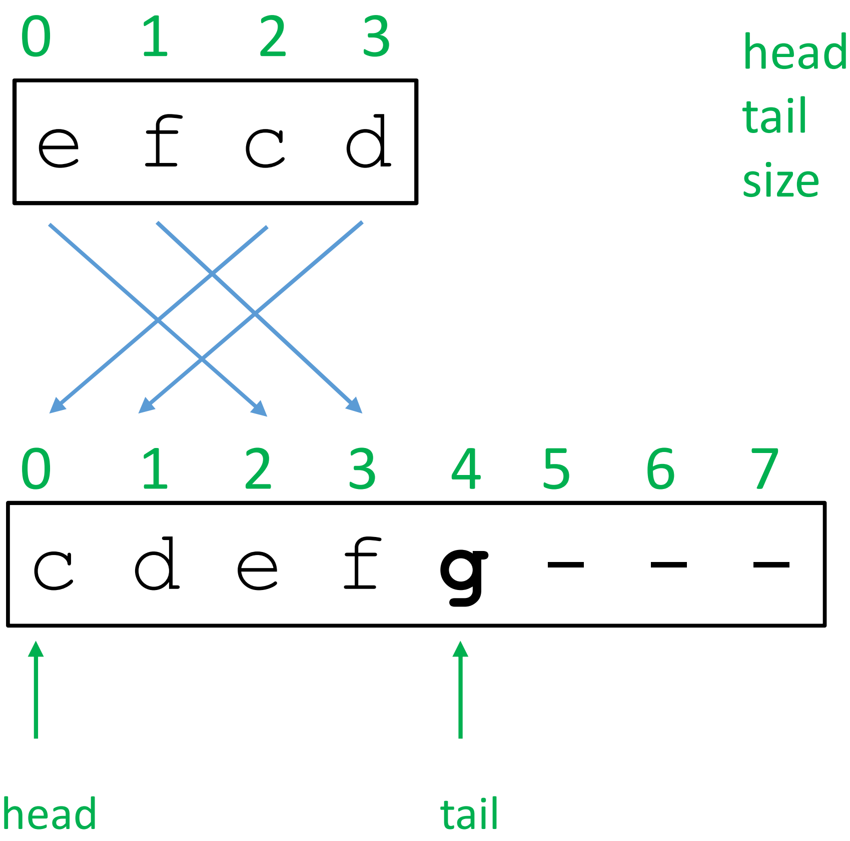
head = 2
tail = 1
size = 4

head = 2
tail = 6
size = 5

How to `enqueue(`**`g`**`)` ? (Alternative)

Increase length of array. Copy elements so that head is 0.

Add the new element **g**.



head = 2
tail = 1
size = 4

head = 0
tail = 4
size = 5

```
enqueue( element ){            //  using a circular array
    if ( size == length )  {
          Increase length of array.
          Copy elements so that head is 0.
    }
    tail = (tail + 1)  mod length
    queue[ tail ] = element
    size = size + 1
}
```

NOTE:
We don’t actually need the tail variable, since tail = (head + size – 1) mod length.

```
enqueue( element ){                          //  using a circular array
      if ( size == length )  {
            //  increase length of array

           create a bigger array tmp[ ]    //  e.g. 2*size
           for i = 0  to  size  - 1
                   tmp[i] = queue[ (head + i)  mod  size ]
           head = 0
           tail  =  size - 1
           queue = tmp
      }
      tail = (tail + 1)  mod length
      queue[ tail ] = element
      size = size + 1
}
```

<u>NOTE:</u>
We don't actually need the tail variable, since tail = (head + size – 1) mod length.

```
dequeue( ){                  //  using a circular array
```

Do you modify the head? the tail? both?

```
}
```

```
dequeue( ){                    //  using a circular array

    //  check that queue.size > 0   (omitted)

    element = queue[head]
    size =  size – 1
    head = (head+1) mod length
    return  element
}
```

Note: this does not affect the tail.

What is the relation between head and tail when size == 0 ?
Suppose length = 4.

$$\text{tail} = (\text{head} + \cancelto{0}{\text{size}} - 1) \bmod \text{length}$$

	array	(head, tail, size)
`Initial state`	`----`	`(0, 3, 0)`

What is the relation between head and tail when size == 0 ?
Suppose length = 4.

$$tail = (head + \overset{0}{\cancel{size}} - 1) \bmod length$$

	array	(head, tail, size)
`Initial state`	`----`	`(0, 3, 0)`
`enqueue( a )`	`a---`	`(0, 0, 1)`
`enqueue( b )`	`ab--`	`(0, 1, 2)`
`dequeue()`	`-b--`	`(1, 1, 1)`
`dequeue()`	`----`	`(2, 1, 0)`

tail head

Recall: ADT (abstract data type)

Defines a data type by the values and operations from the user's perspective only. It ignores the details of the implementation.

Examples:

- list
- stack
- queue
- …

Exercise: what can we do with just the operations of an ADT ?

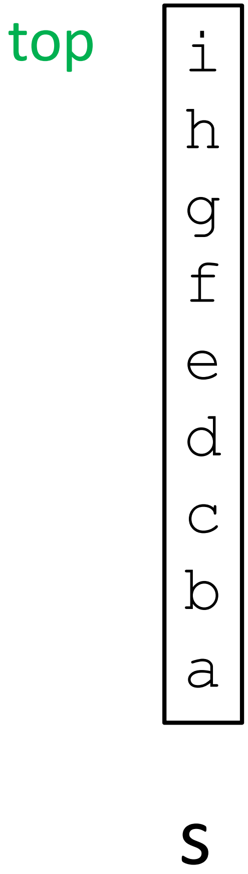
Exercise: Implement a queue using a stack(s).

Hint: you want FIFO (first in, first out) behavior.

```
enqueue(  e ){
             :                    // use only  push(e) , pop(),  isEmpty()
}


dequeue( ) {
             :                    // use only  push(e) , pop(),  isEmpty()
}
```

You are also allowed temporary variables, loops, etc.



top

i
h
g
f
e
d
c
b
a

S

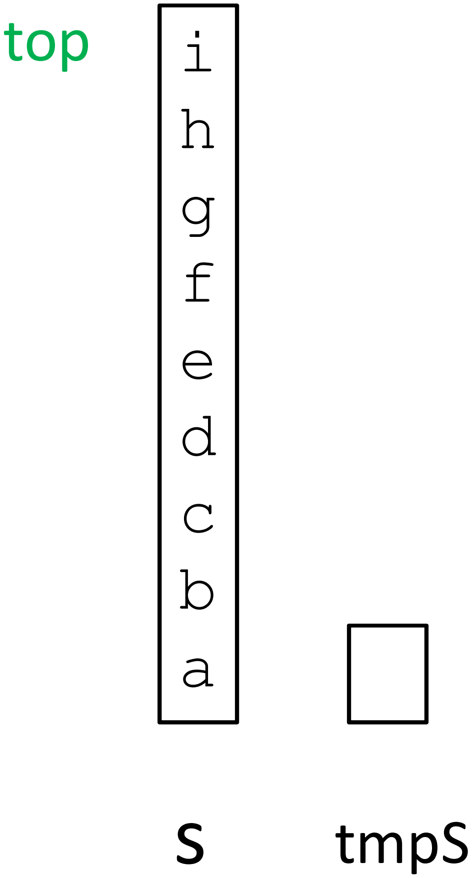
```
enqueue(  element ){
      s.push( element )
}
```

Suppose we implement enqueue as above.

The example on the left shows what happens when we “enqueue” elements `a` to `i` , namely we push them onto a stack.

But how do we implement dequeue() in this case?

Hint: Use a second stack.


top

i
h
g
f
e
d
c
b
a

S

tmpS

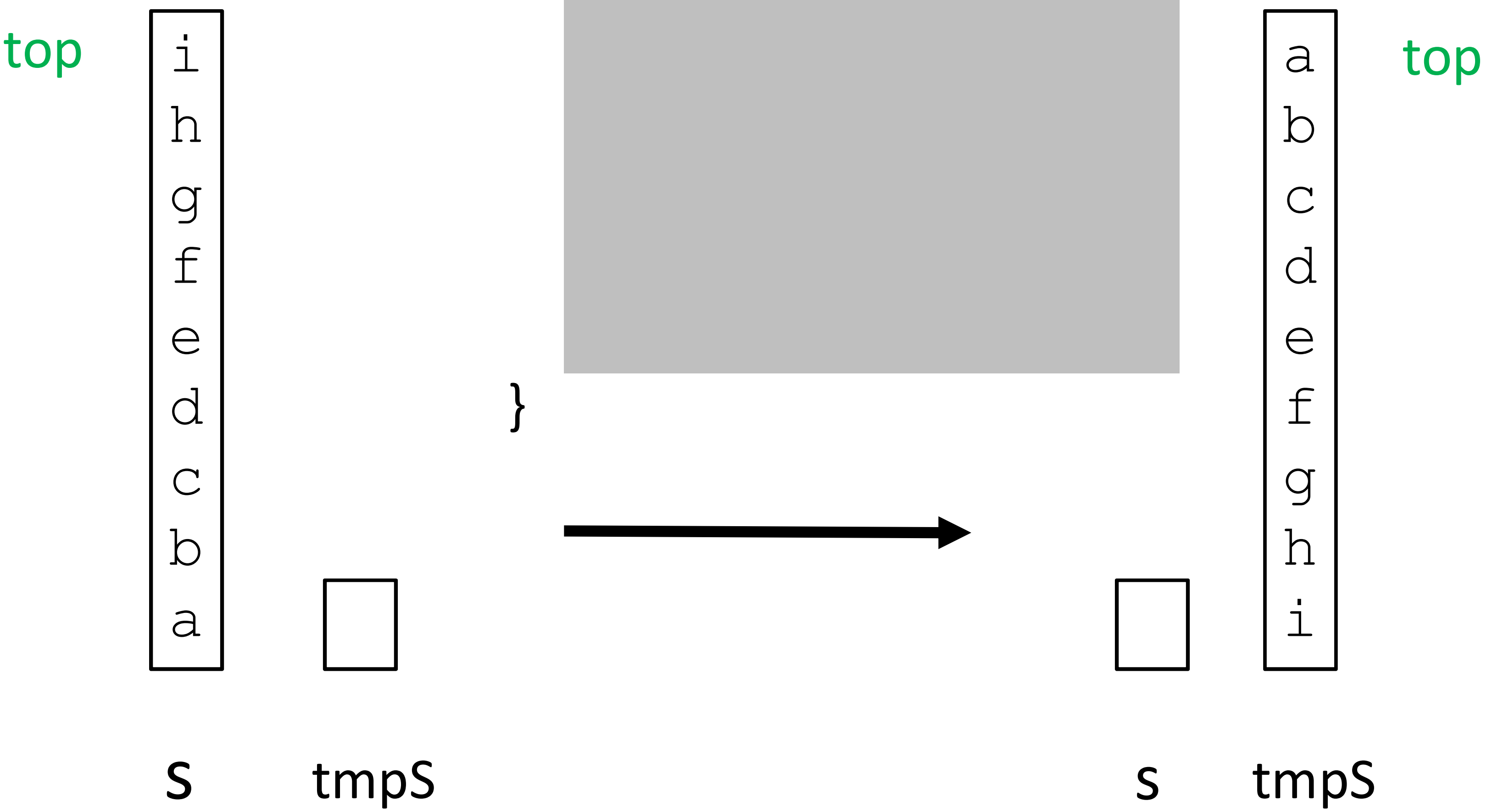
```
dequeue(){

}
```

Write pseudocode that uses only operations push(e) , pop(), isEmpty() .

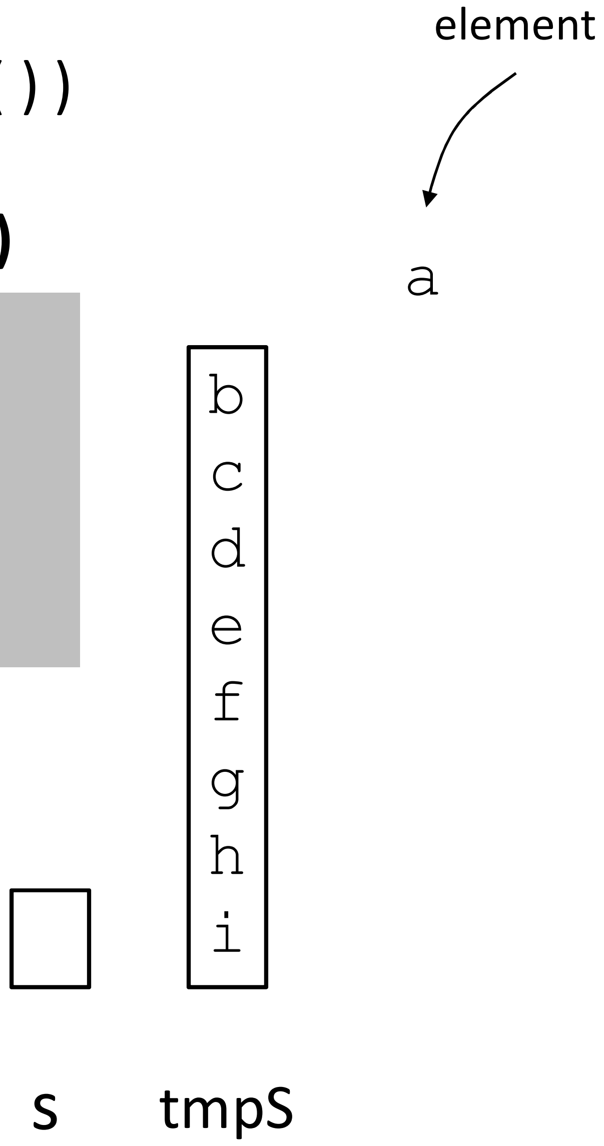
```
dequeue(){
    while ( ! s.isEmpty() ){
        tmpS.push(  s.pop( ) )
    }

}
```

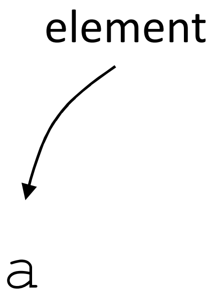


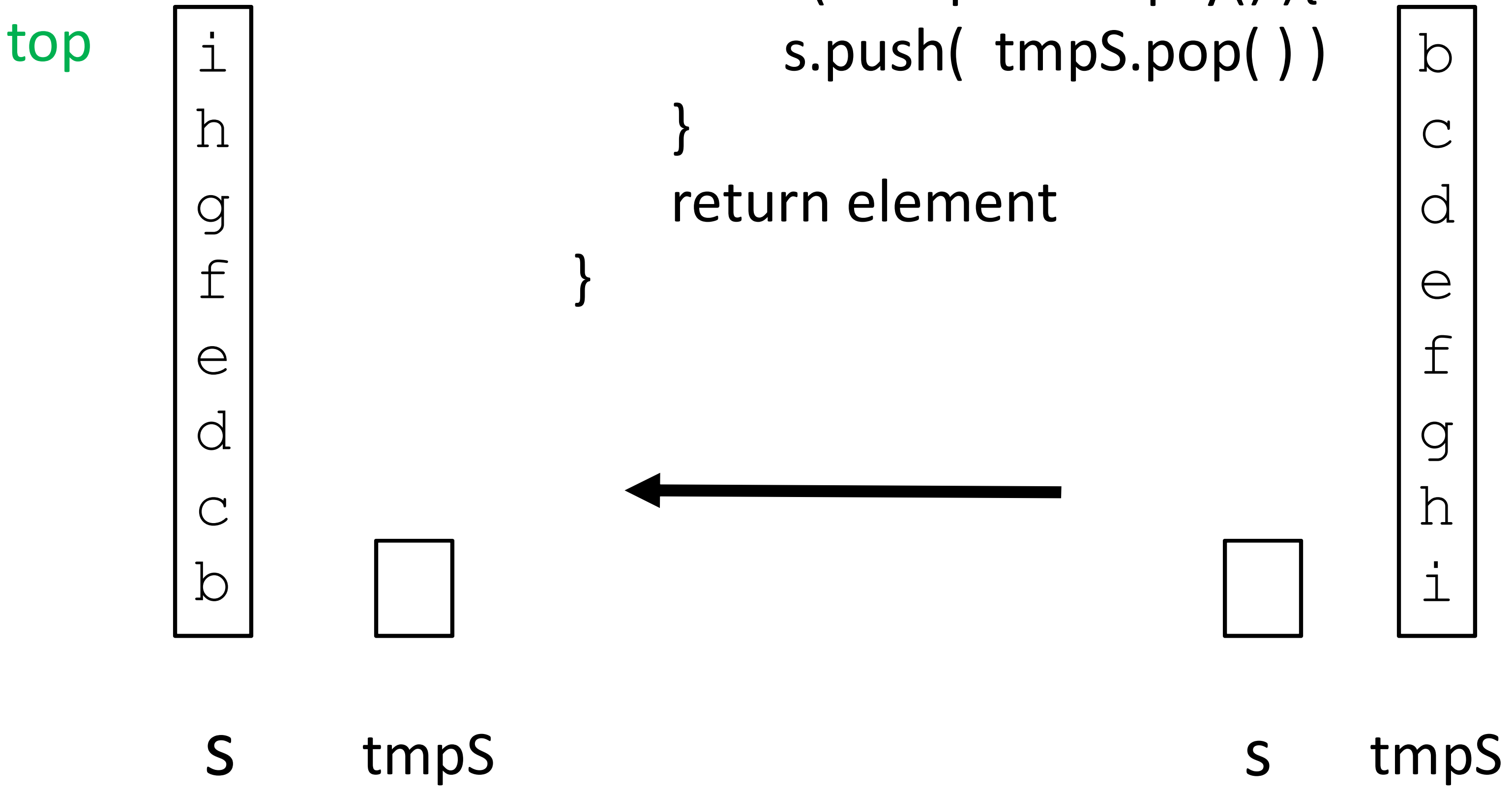
```
dequeue(){
    while ( ! s.isEmpty() ){
         tmpS.push(  s.pop( ) )
    }
    element = tmpS.pop()

}
```



```
dequeue(){
    while ( ! s.isEmpty() ){
        tmpS.push(  s.pop( ) )
    }
    element = tmpS.pop()
    while ( ! tmpS.isEmpty() ){
        s.push(  tmpS.pop( ) )
    }
    return element
}
```





Coming up...

Lectures

Fri. Feb. 18   Mathematical Induction

Next week... recursion

Assessments

Assignment 2 is due on Fri. Feb. 25.

Assignment 3 will be released shortly after that.   (READING WEEK)