COMP 250

Lecture 9

queue ADT

Sept. 27/28, 2017

ADT (abstract data type)

- List

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

- Stack

  push, pop(), ..

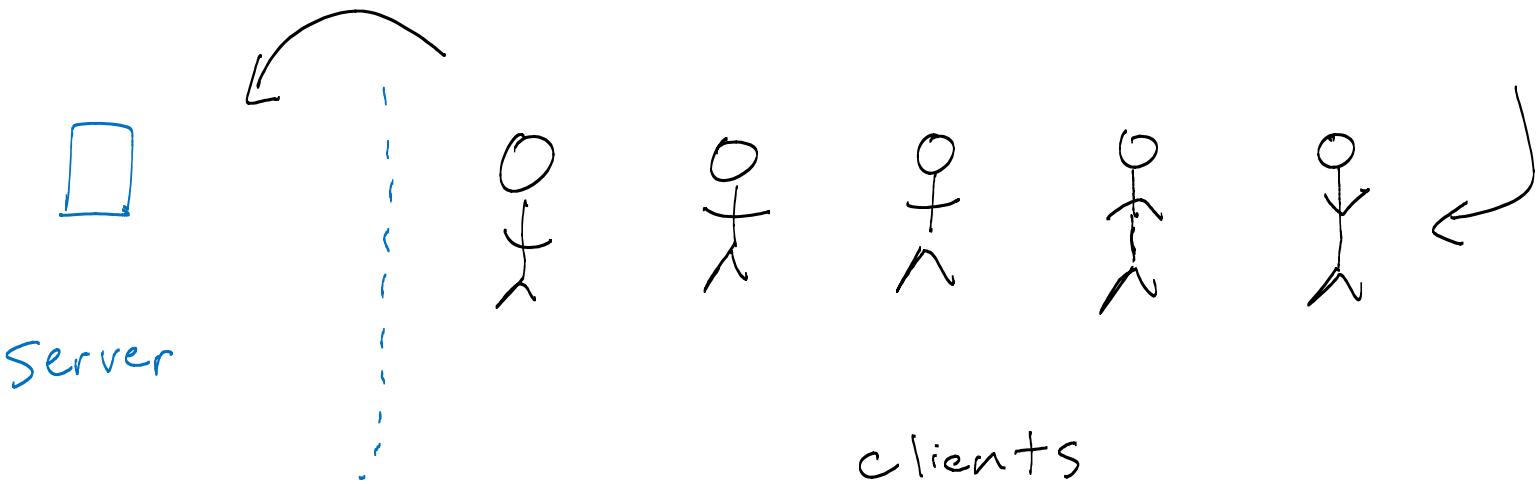
- Queue

  enqueue(e), dequeue()

Queue

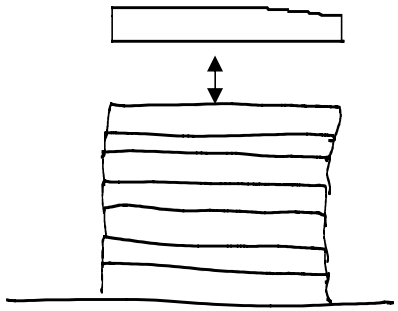
dequeue
(remove from
front)

enqueue
(add at back)



Examples

- keyboard buffer
- printer jobs
- CPU processes (applications do not run in parallel)
- web server
- …….

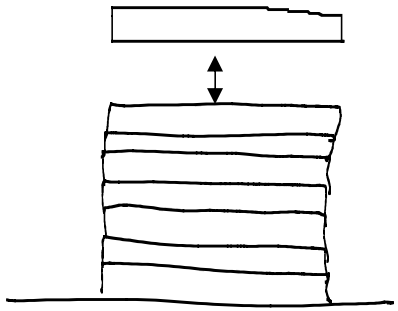


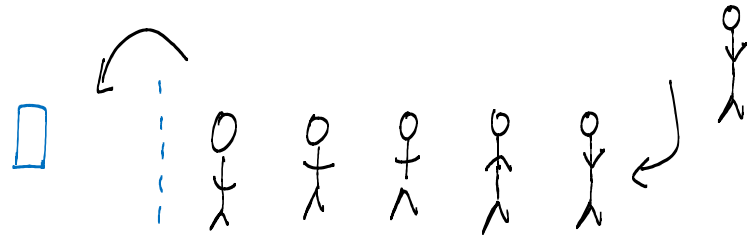
Stack

push(e)

pop()

LIFO
(last in, first out)





Stack

push(e)

pop()

LIFO
(last in, first out)

Queue

enqueue(e)

dequeue()

FIFO
(first in, first out)
"first come, first serve"

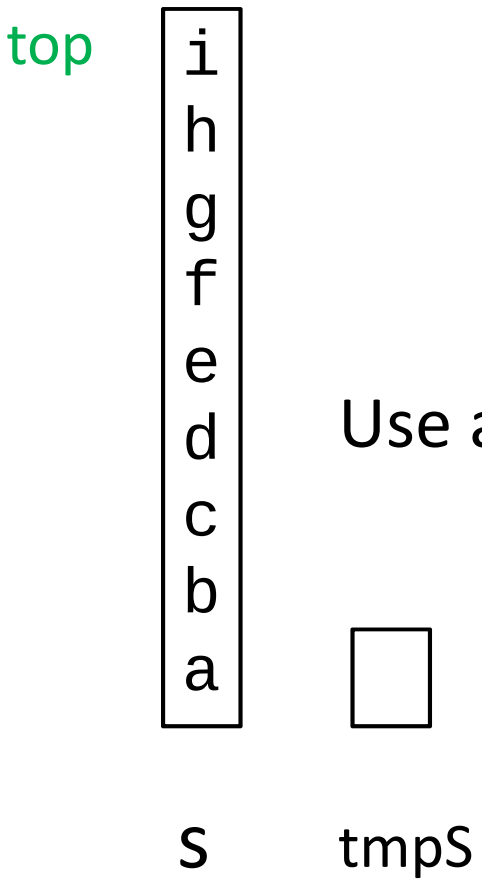
<u>Exercise</u>: Use stack(s) to implement a queue.

```
 enqueue(  e ){      // add element
              :
}


 dequeue( ) {        // remove 'oldest' element
              :
}
```

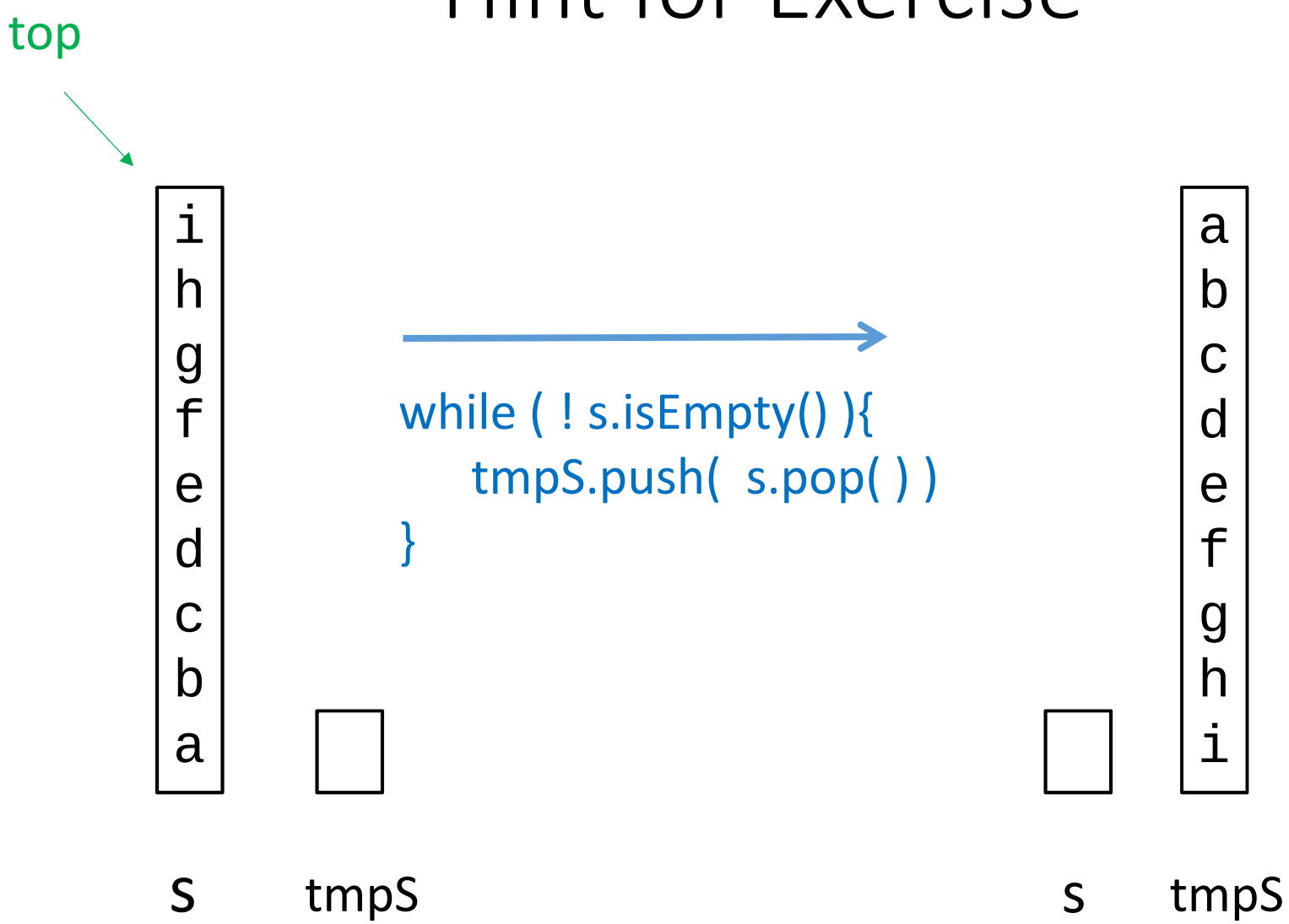
Write pseudocode for these two methods that uses a stack, namely use the operations push(e) , pop(), isEmpty() .

Hint for Exercise



Use a second stack.

Hint for Exercise



Queue Example

enqueue(a)	a
enqueue(b)	ab
dequeue( )	b

Queue Example

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

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	(unnecessary)	
array list		

How to implement a queue?

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

SLOW

Implementing a queue with an array list. (BAD)

length = 4

0123 indices

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

Requires shift

Implementing a queue with an array list. (BAD)

length = 4

	0123 indices	
enqueue(a)	a---	
enqueue(b)	ab--	
dequeue( )	b---	Requires shift
enqueue(c)	bc--	
enqueue(d)	bcd-	
enqueue(e)	bcde	
dequeue( )	cde-	Requires shift

Implementing a queue with an array list. (BAD)

length = 4

	0123 indices
enqueue(a)	a---
enqueue(b)	ab--
dequeue( )	b---
enqueue(c)	bc--
enqueue(d)	bcd-
enqueue(e)	bcde
dequeue( )	cde-
enqueue(f)	cdef
enqueue(g)	cdefg---

requires expansion

Implementing a queue with an **expanding array.**
(also BAD)

Use head and tail indices
(tail = head + size – 1)

enqueue(a)	a---	(0,0)
enqueue(b)	ab--	(0,1)
dequeue( )	-b--	(1,1)
enqueue(c)	-bc-	(1,2)
enqueue(d)	-bcd	(1,3)
enqueue(e)	?	

Implementing a queue with an **expanding array**.

(also BAD)

Use head and tail indices
(tail = head + size – 1)

enqueue(a)	a---	(0,0)
enqueue(b)	ab--	(0,1)
dequeue( )	-b--	(1,1)
enqueue(c)	-bc-	(1,2)
enqueue(d)	-bcd	(1,3)
enqueue(e)	-bcde---	(1,4)
dequeue( )	--cde---	(2,4)
enqueue(f)	--cdef--	(2,5)
enqueue(g)	--cdefg-	(2,6)

Make bigger array and copy to it.

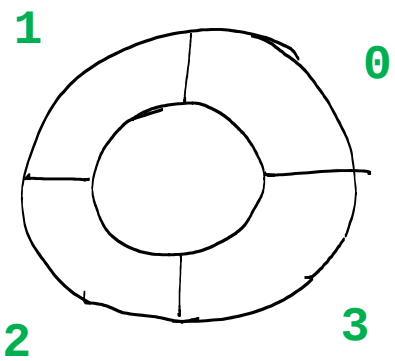
An expanding array is an inefficient usage of space.

A better idea is….

Circular array

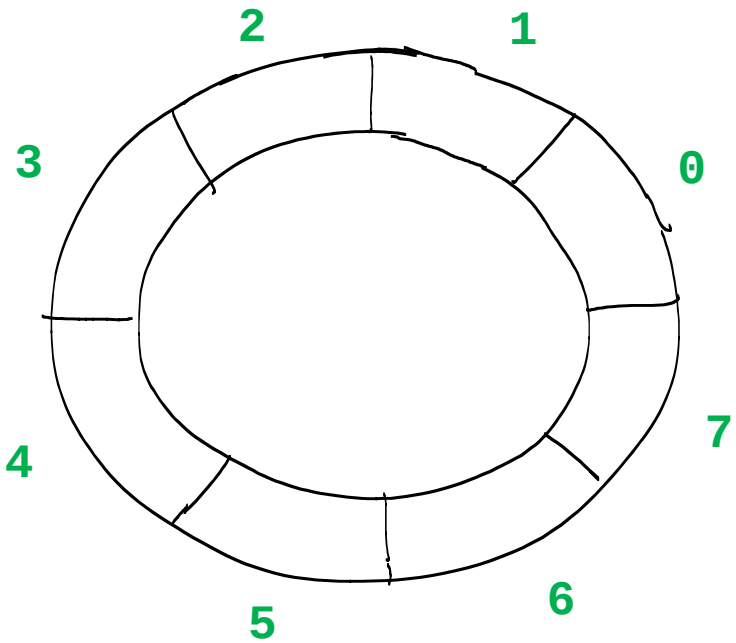
length = 4

0123



length = 8

01234567



Circular array

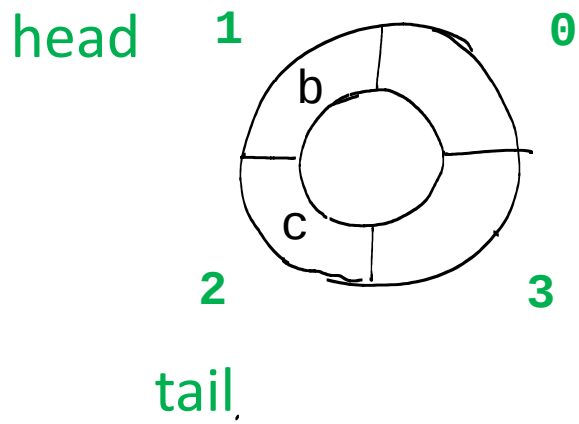
tail = (head + size – 1) % length

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

0123

- bc -

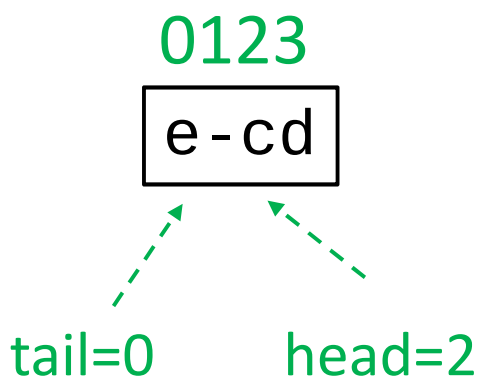
head=1 tail=2

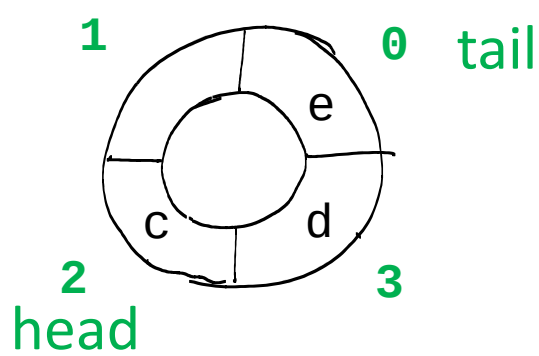


Circular array

tail = (head + size – 1) % length

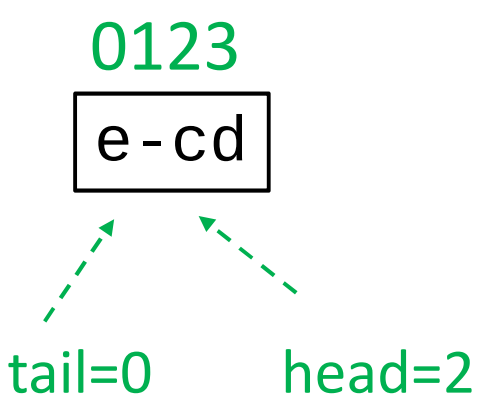
```
enqueue( a )
enqueue( b )
dequeue( )
enqueue( c )
enqueue( d )
enqueue( e )
dequeue()
```





Circular array

tail = (head + size – 1) % length




```
enqueue( element ){
    if  (size < length)
        queue[ (tail + 1) % length] = element
    else ....   // coming up
    size = size+1
}


dequeue(){   //  check if empty omitted
    element = queue[head]
    head = (head + 1) % length
    size = size-1
    return element
}
```

Implementing a queue with a circular array (GOOD)

tail = (head + size – 1) % length

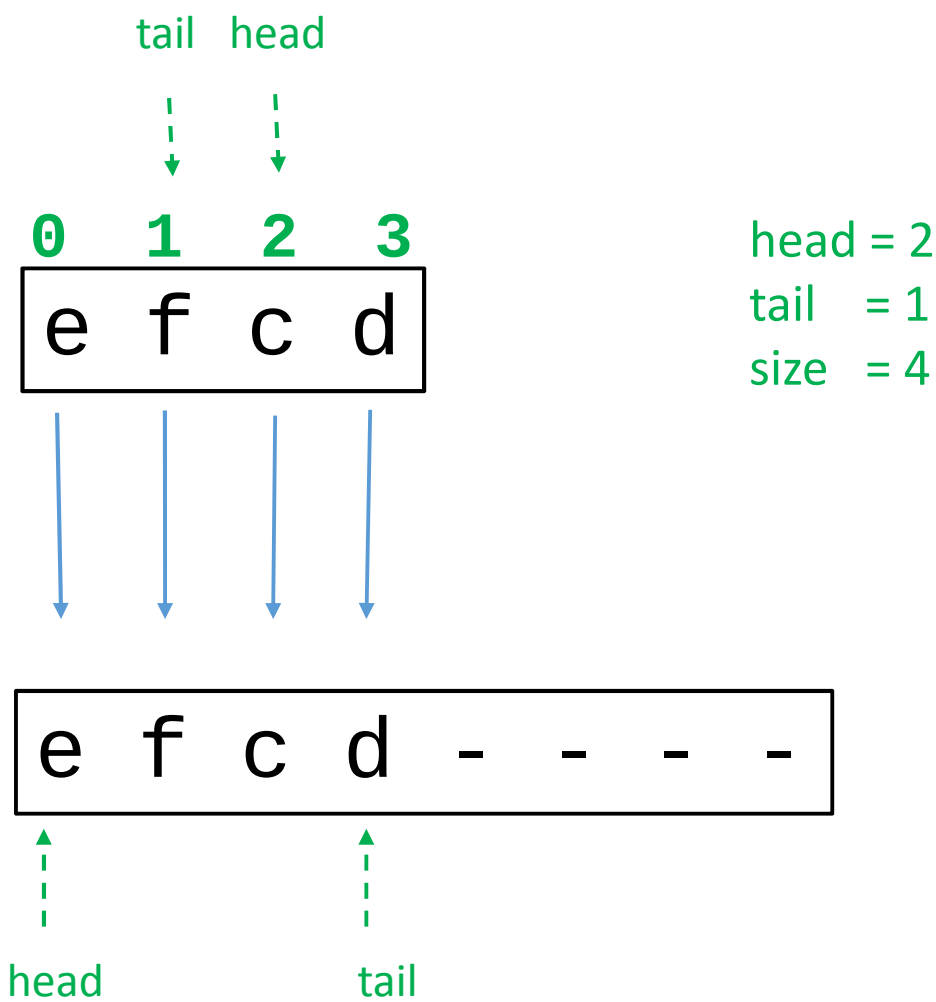
	array	(head, tail, size)
`enqueue( a )`	`a---`	`(0, 0, 1)`
`enqueue( b )`	`ab--`	`(0, 1, 2)`
`dequeue()`	`-b--`	`(1, 1, 1)`
`enqueue( c )`	`-bc-`	`(1, 2, 2)`
`enqueue( d )`	`-bcd`	`(1, 3, 3)`
`enqueue( e )`	`ebcd`	`(1, 0, 4)`
`dequeue()`	`e-cd`	`(2, 0, 3)`
`enqueue( f )`	`efcd`	`(2, 1, 4)`

Implementing a queue with a circular array

tail = (head + size – 1) % length

	array	(head, tail, size)
`enqueue( a )`	`a---`	`(0, 0, 1)`
`enqueue( b )`	`ab--`	`(0, 1, 2)`
`dequeue()`	`-b--`	`(1, 1, 1)`
`enqueue( c )`	`-bc-`	`(1, 2, 2)`
`enqueue( d )`	`-bcd`	`(1, 3, 3)`
`enqueue( e )`	`ebcd`	`(1, 0, 4)`
`dequeue()`	`e-cd`	`(2, 0, 3)`
`enqueue(f)`	`efcd`	`(2, 1, 4)`
`enqueue(g)`	**?**	

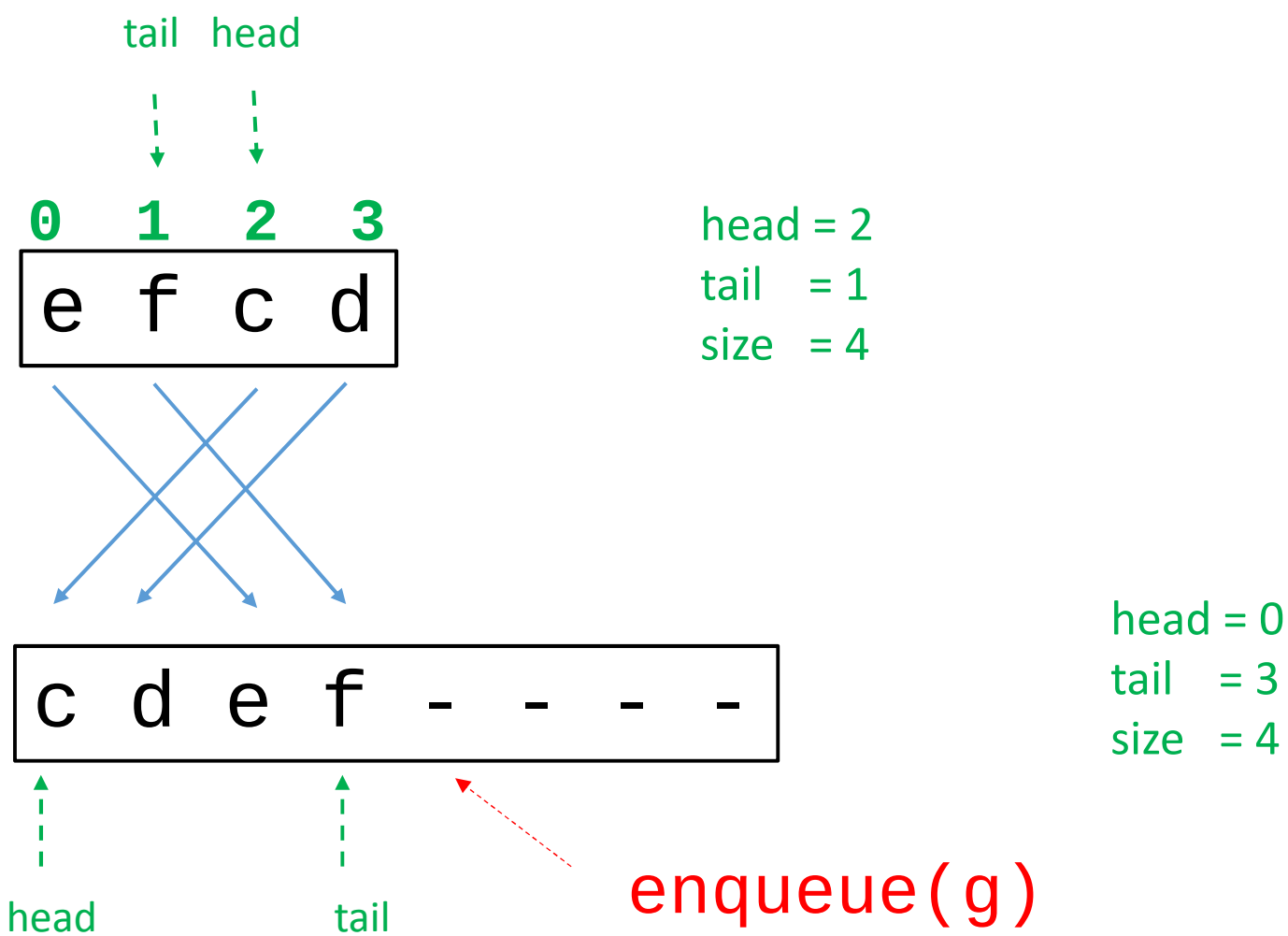
Increase length of array and copy? **BAD**



enqueue(g) ?

Increase length of array. Copy so that head moves to front.

(GOOD)



```
enqueue( element ){
    if ( queue.size == queue.length)  {
        //  increase length of array

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

What happens when size == 0 ?

tail = (head + size – 1) % 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

ADT's, API's & Java

The following are related, but quite different:

- ADT (abstract data type)
- Java API (application program interface)
- Java keyword `interface`

To be discussed much more at end of the course.

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
- …

Java API

API = application program *interface*

Gives class methods and some fields, and comments on what the methods do. e.g.

https://docs.oracle.com/javase/7/docs/api/java/util/LinkedList.html

Java `interface`

- reserved word (nothing to do with “I” in “API”)

- like a class, but only the method signatures are defined

Example: List interface

```
interface  List<T>  {
    void        add(T)
    void        add(int,  T)
    T           remove(int)
    boolean     isEmpty()
    T           get( int )
    int         size()
            :
}
```

https://docs.oracle.com/javase/7/docs/api/java/util/List.html

```
class  ArrayList<T>   implements  List<T> {

     void          add(T)          { .... }
     void          add(int,  T)   {  ....  }
     T             remove(int) {  ....  }
     boolean    isEmpty()     {  ....  }
     T             get( int )     {  ....  }
     int           size()          {  ....  }
              :
}
```

Each of the List methods are implemented.
(In addition, other methods may be defined and implemented.)

```
class  LinkedList<T>   implements  List<T> {

    void        add(T)        { .... }
    void        add(int,  T)  {  ....  }
    T           remove(int)   {  ....  }
    boolean     isEmpty()     {  ....  }
    T           get( int )    {  ....  }
    int         size()        {  ....  }
            :
}
```

Each of the List methods are implemented.
(In addition, other methods may be defined and implemented.)

More examples

- `interface` List

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

- `class` Stack

  push, pop(), ..

- `interface` Queue

  offer(e), poll (), ...