COMP 250

Lecture 14

queue ADT

Oct. 12, 2018

ADT (abstract data type)

- List

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

- Stack

  push, pop(), ..

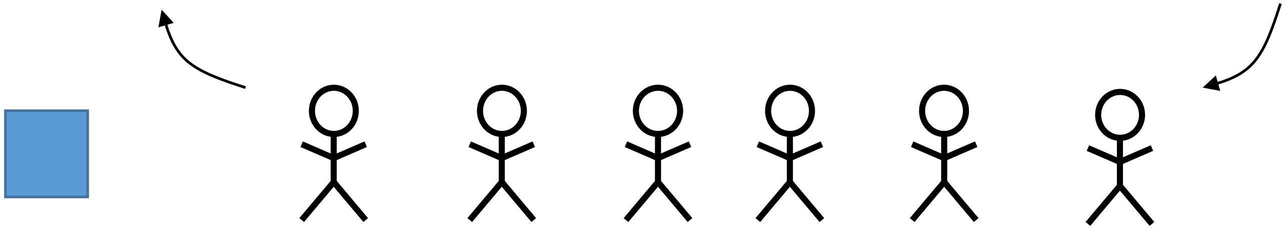
- Queue

  enqueue(e), dequeue()

Queue

dequeue
(remove from front)

enqueue
(add at back)

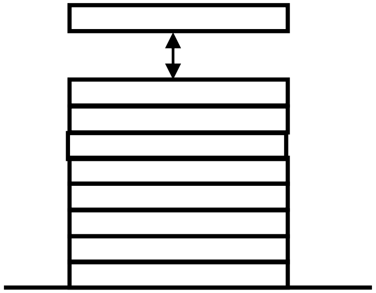


e.g. server

clients

Examples

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

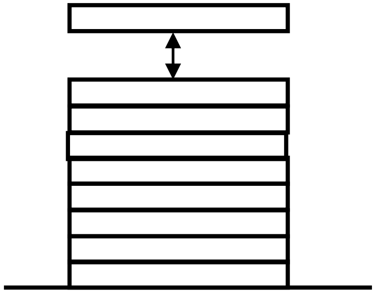


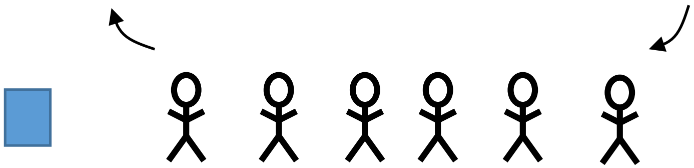
Stack

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”

Queue Example

enqueue(a)	a
enqueue(b)	ab
dequeue( )	b
enqueue(c)	
enqueue(d)	
enqueue(e)	
dequeue( )	
enqueue(f)	
enqueue(g)	

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---	shift
enqueue(c)	bc--	
enqueue(d)	bcd-	
enqueue(e)	bcde	
dequeue( )	cde-	shift

Implementing a queue with an array list. (BAD)

0123 indices

length = 4

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.**

Use (head,tail) indices.

	0123	
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)	?	

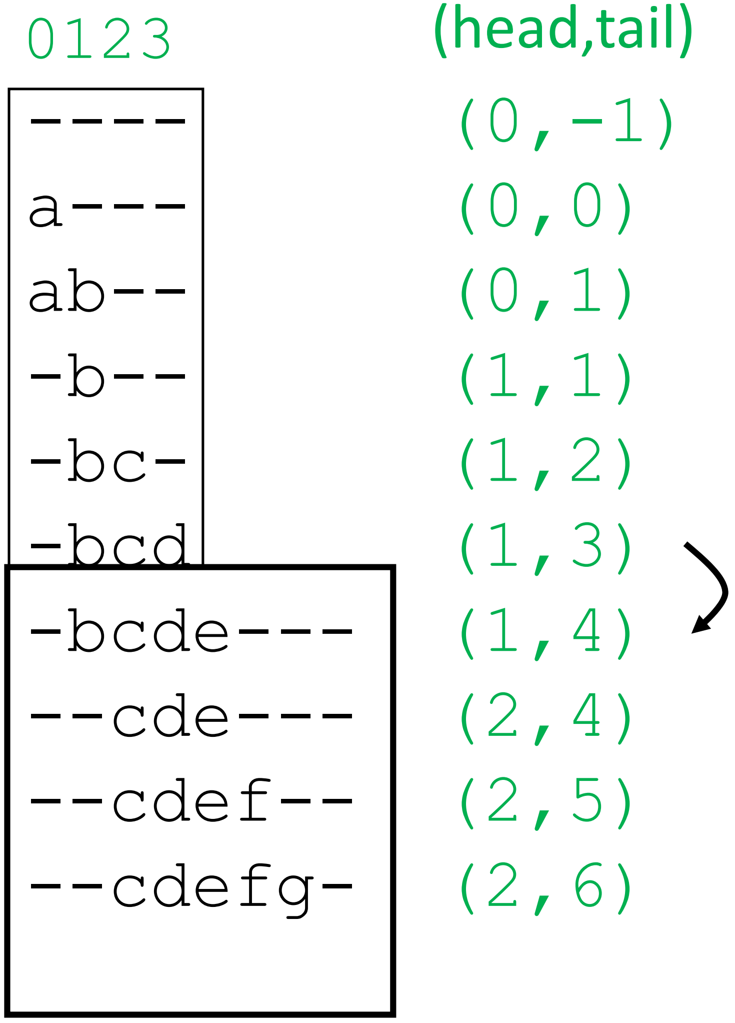
tail = head + size – 1

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

	0123	Use (head,tail) indices.
	----	(0,-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)	?	

tail = head + size – 1

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



	0123	(head,tail)
	`----`	(0,-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.

dequeue from head? tail? ?????

enqueue to head? tail? ?????

dequeue from head

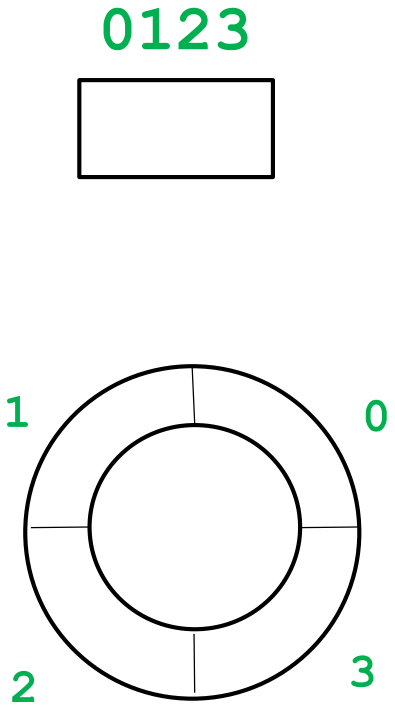
enqueue to tail + 1

An expanding array is an inefficient usage of space.

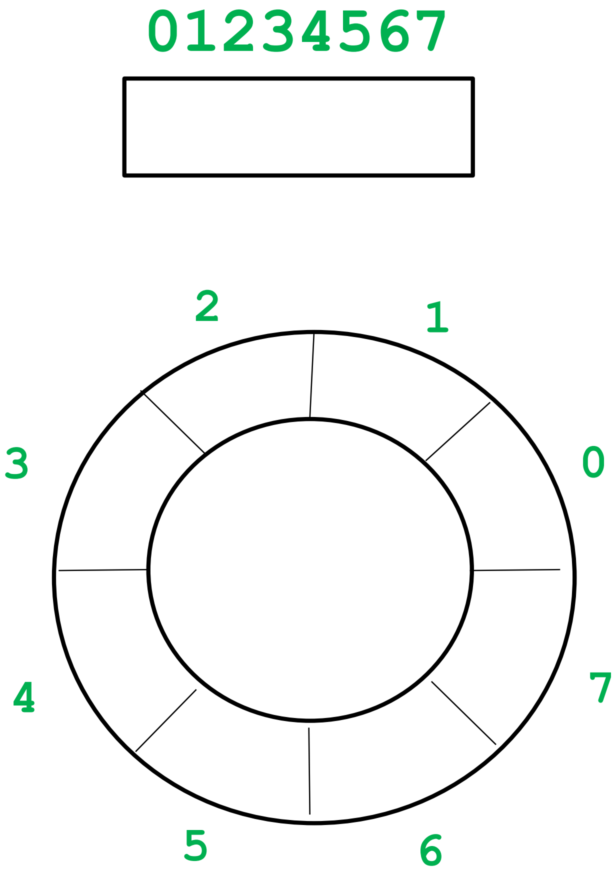
A better idea is….

Circular array

length = 4



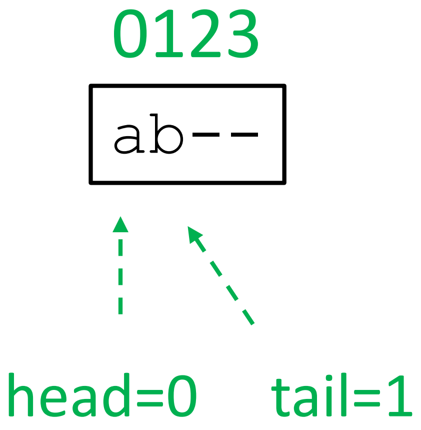
length = 8



Circular array

tail = (head + size – 1) % length

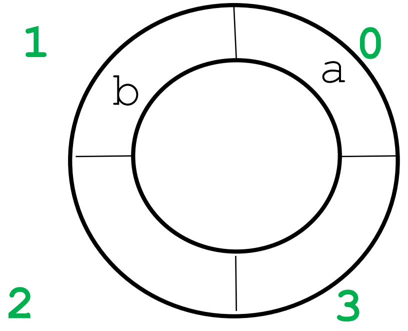
```
enqueue( a )
enqueue( b )
dequeue()
enqueue( c )
enqueue( d )
enqueue( e )
dequeue()
```



0123

ab--

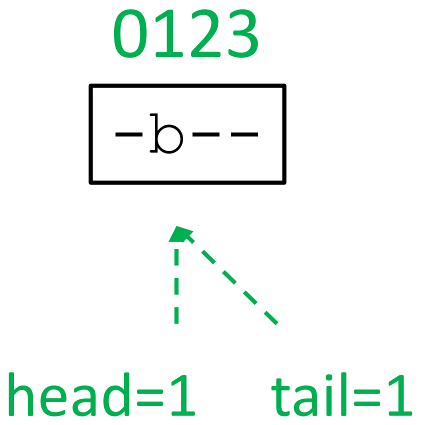
head=0 tail=1

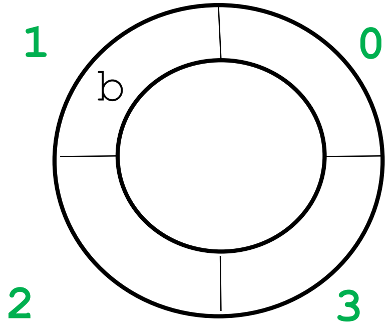


Circular array

tail = (head + size – 1) % length

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

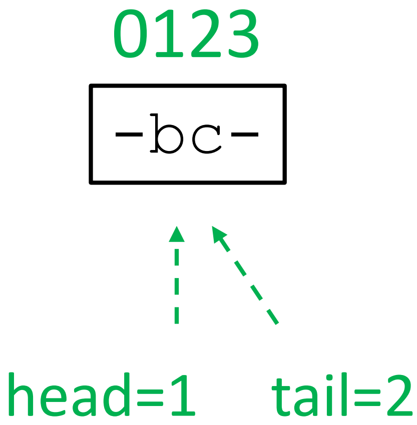




Circular array

tail = (head + size – 1) % length

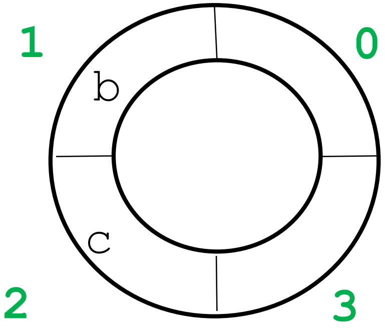
```
enqueue( a )
enqueue( b )
dequeue()
enqueue( c )
enqueue( d )
enqueue( e )
dequeue()
```



0123

-bc-

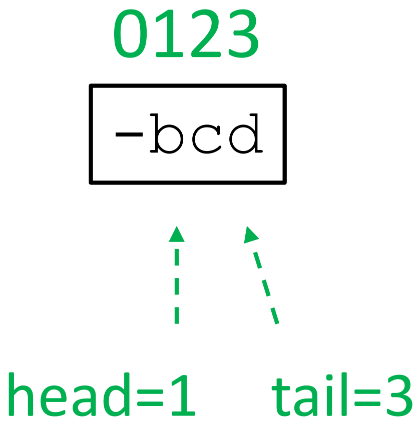
head=1 tail=2

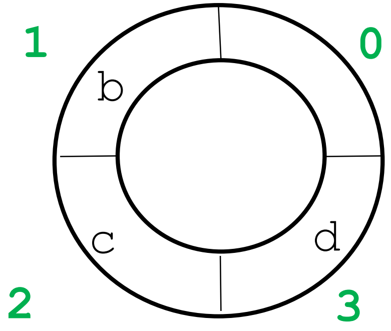


Circular array

tail = (head + size – 1) % length

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

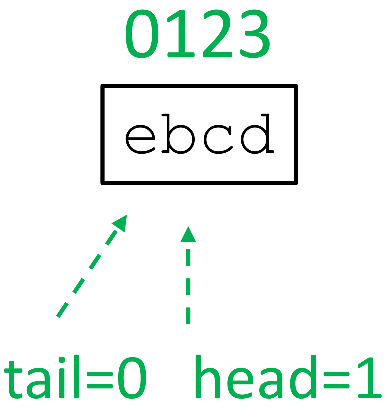




Circular array

tail = (head + size – 1) % length

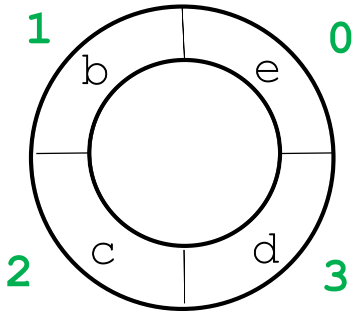
```
enqueue( a )
enqueue( b )
dequeue( )
enqueue( c )
enqueue( d )
enqueue( e )
dequeue()
```



0123

ebcd

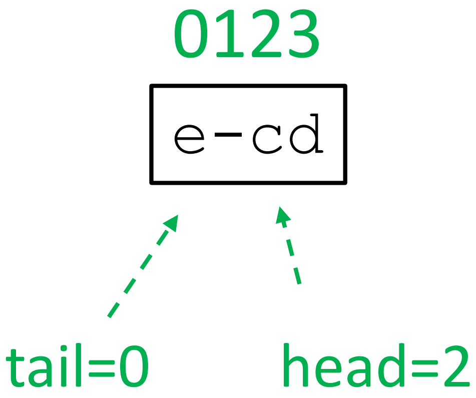
tail=0 head=1

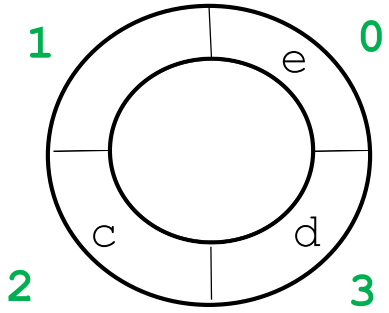


Circular array

tail = (head + size – 1) % length

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

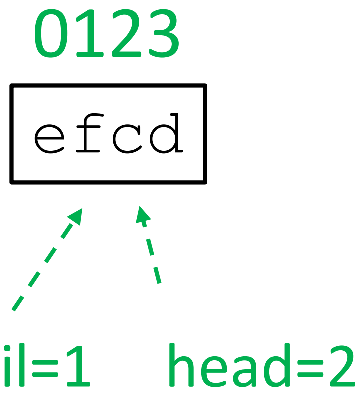


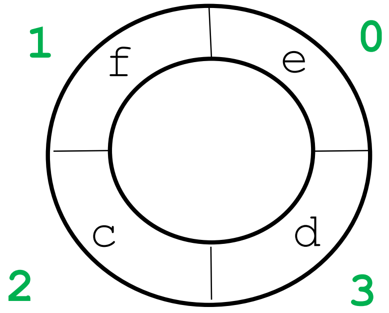


Circular array

tail = (head + size – 1) % length

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

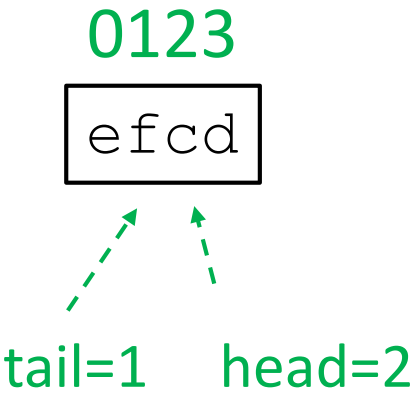


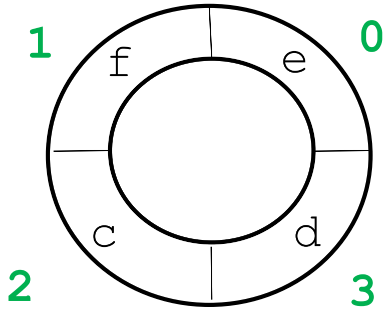


Circular array

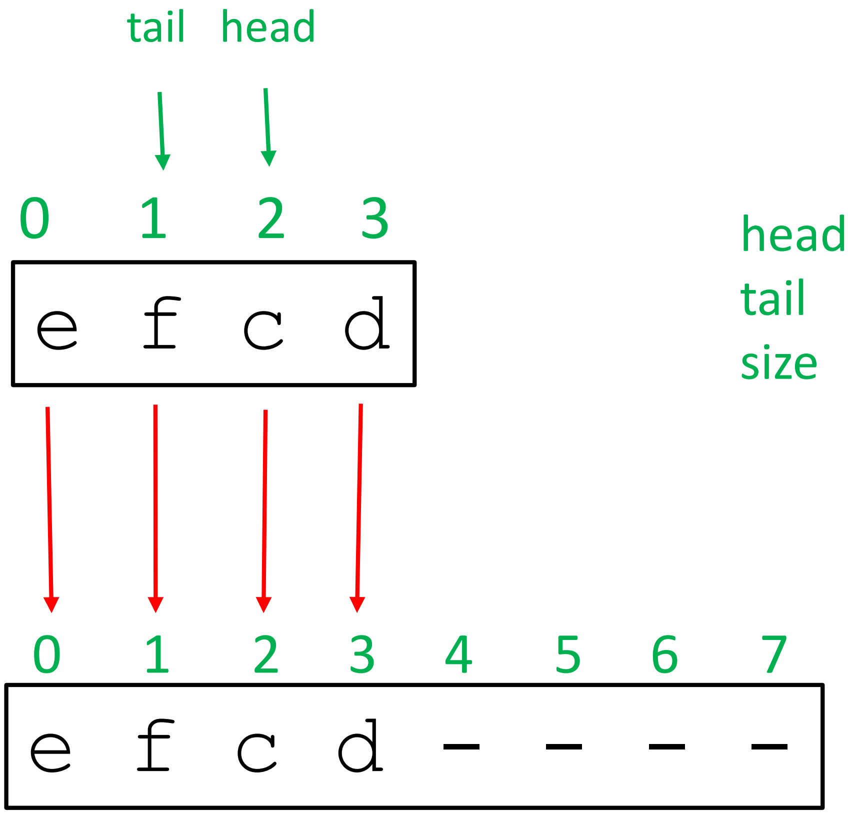
tail = (head + size – 1) % length

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





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



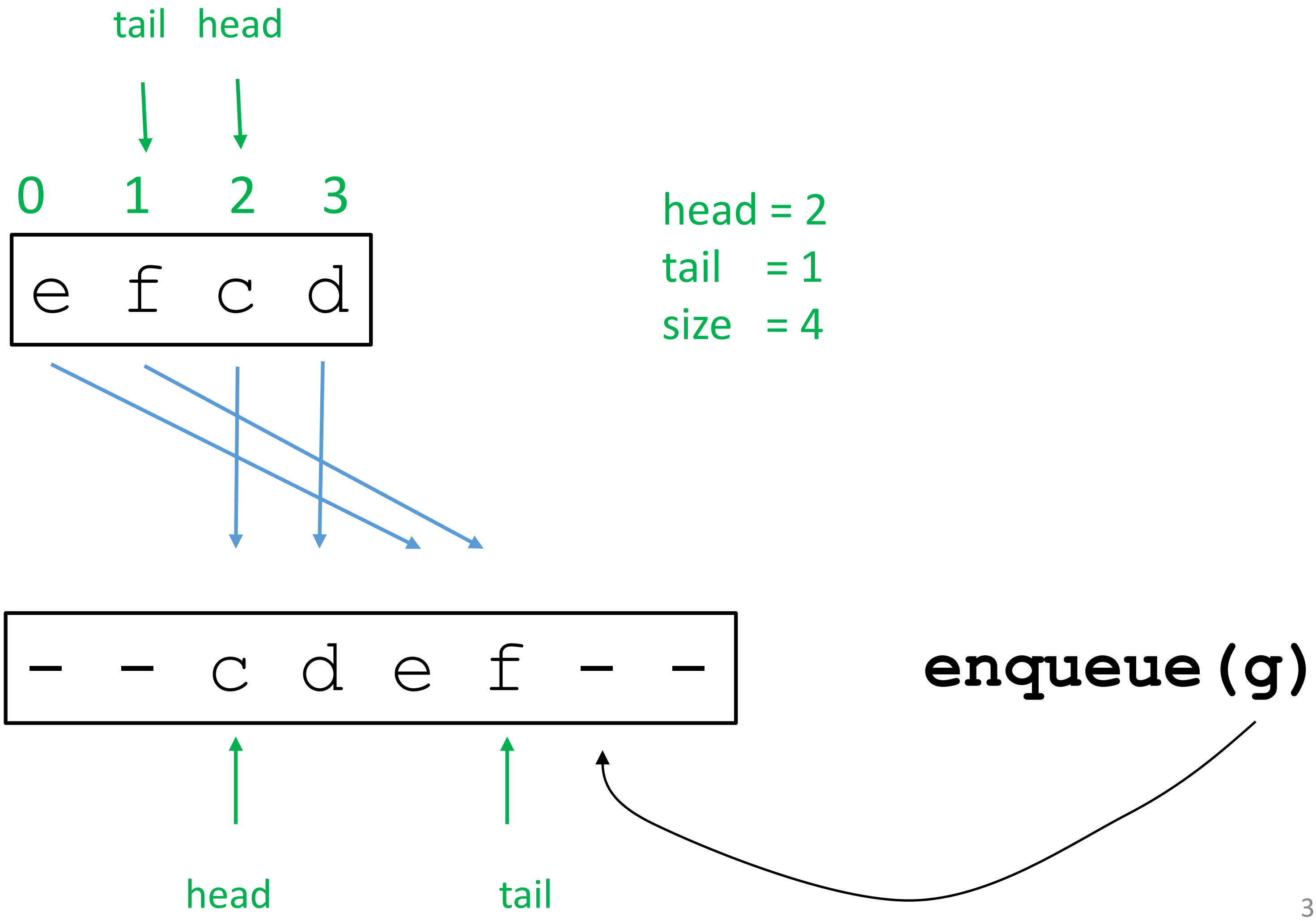
0	1	2	3
e	f	c	d

0	1	2	3	4	5	6	7
e	f	c	d	–	–	–	–

head = 2
tail = 1
size = 4

How to `enqueue(g)` ?

Increase length of array. Copy such that head stays. (GOOD, but we'll do it slightly differently, next slide)



tail head

0	1	2	3
e	f	c	d

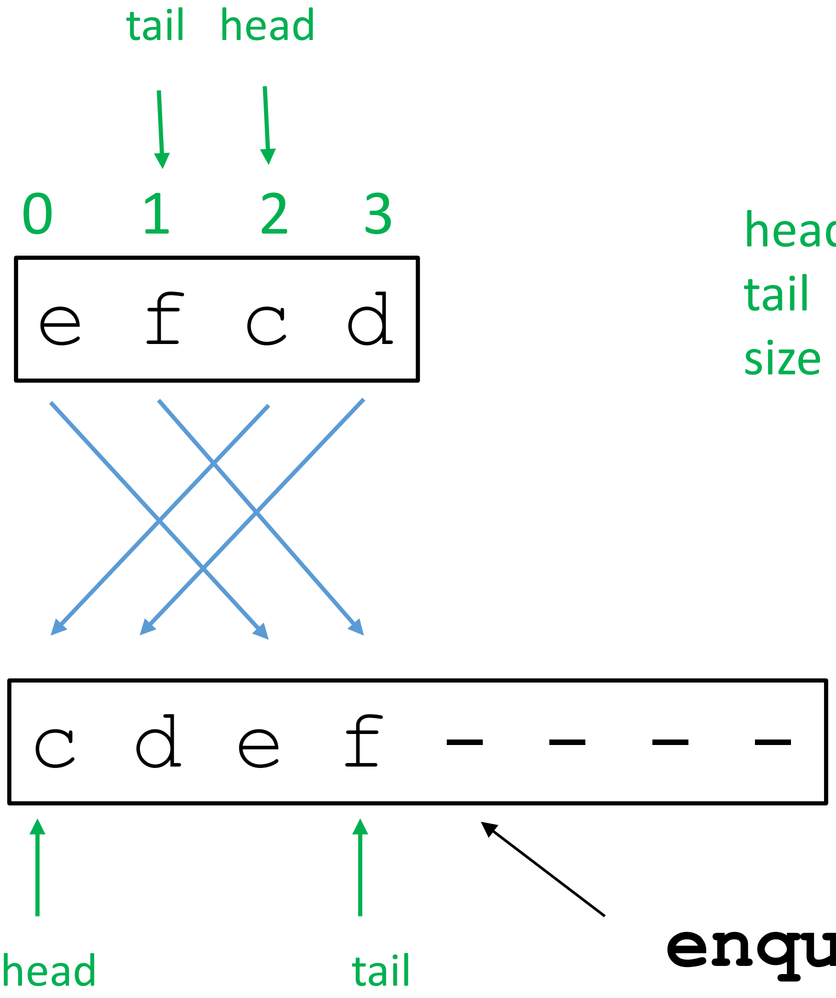
head = 2
tail = 1
size = 4

-	-	c	d	e	f	-	-

`enqueue(g)`

head tail

Increase length of array. Copy so that head moves to slot 0. (also GOOD)



tail head

0 1 2 3

e f c d

c d e f - - - -

head

tail

head = 2
tail = 1
size = 4

head = 0
tail = 3
size = 4

`enqueue(g)`

```
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
}
```

Note that we don’t have a tail variable here.   Instead, it can be computed anytime with: tail = (head + size – 1) % length

```
dequeue( ){    //  check that queue.size > 0
    element = queue[head]
    queue.size = queue.size – 1
    head = (head+1) % length
    return  element
}
```

What is the relation between head and tail when size == 0 ?

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

What is the relation between head and tail when size == 0 ?

tail = (head + ~~size~~ – 1) % length

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

What is the relation between head and tail 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 (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
- …

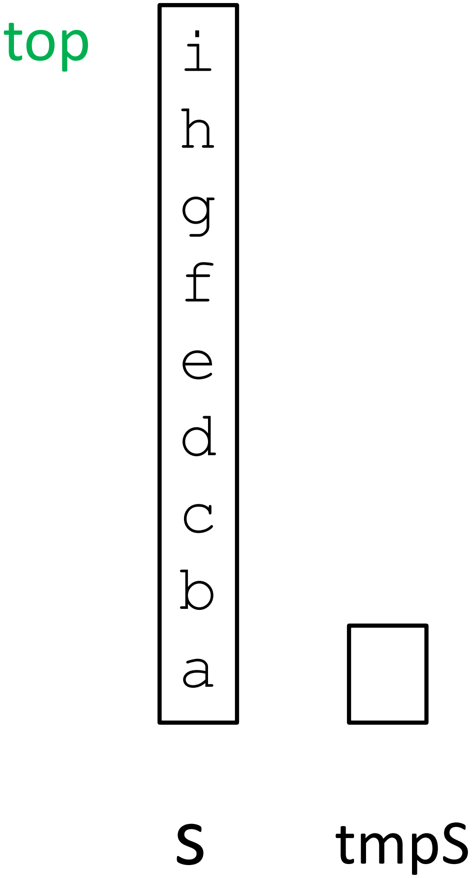
Exercise: Implement a queue using a stack(s).

```
  enqueue(  e ){
               :
}


  dequeue( ) {
               :
}
```

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

Hint: Use a second stack. What can we do?

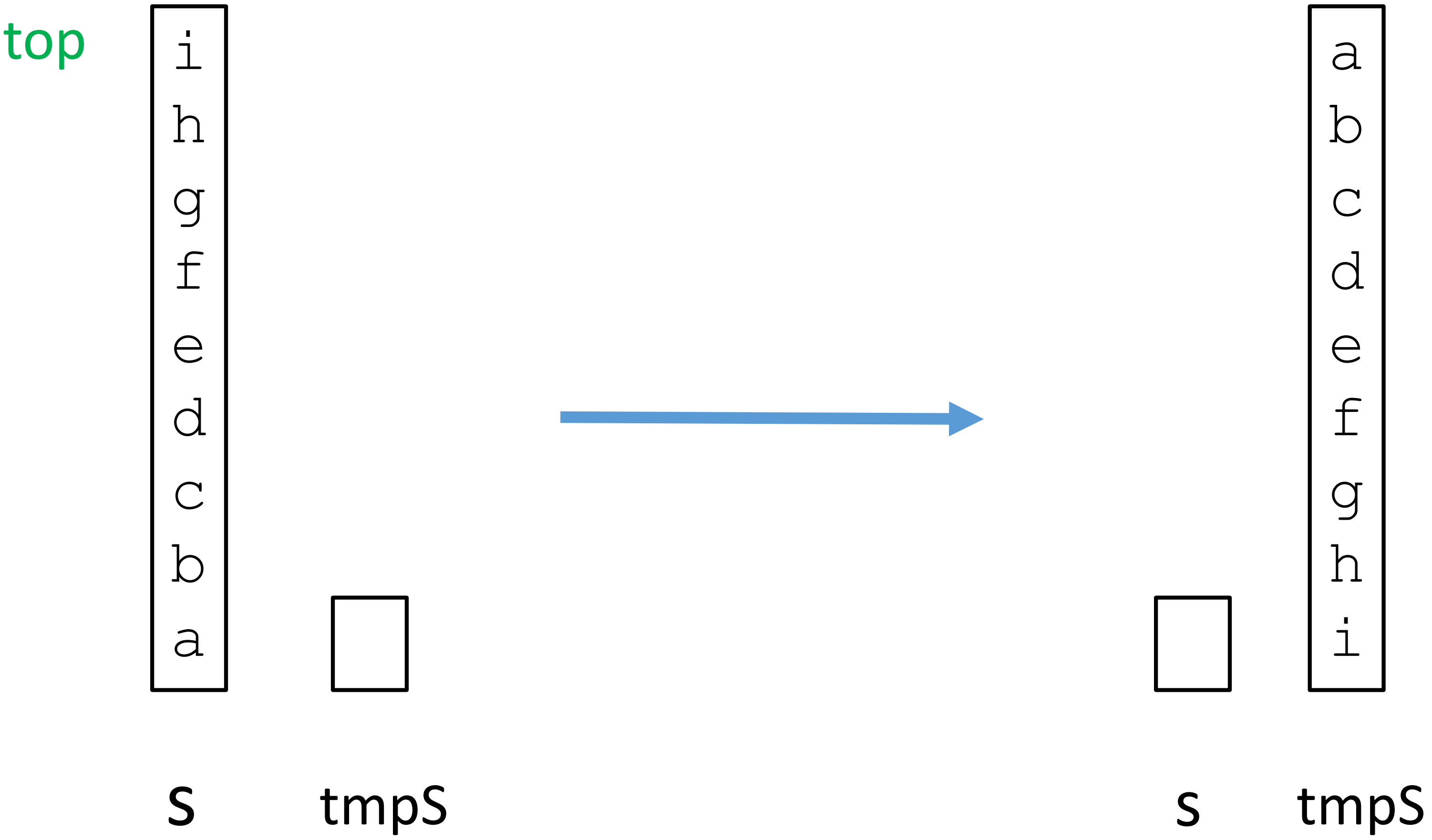


```
 enqueue( e ){
                :
}


 dequeue( ) {
                :
}
```

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

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



top

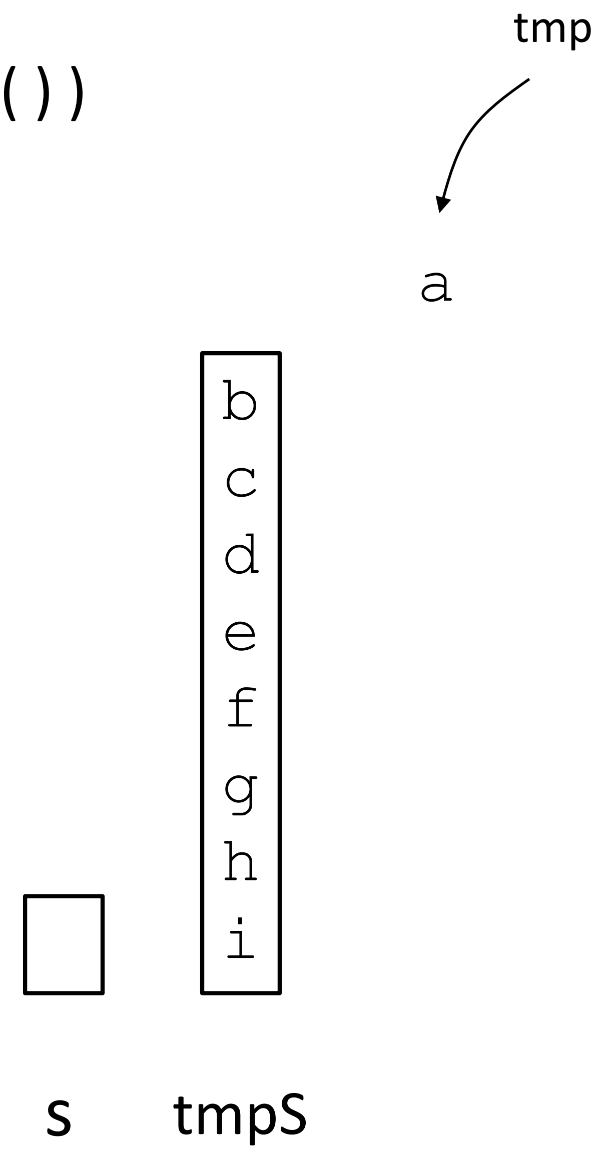
i
h
g
f
e
d
c
b
a

s tmpS

a
b
c
d
e
f
g
h
i

s tmpS

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



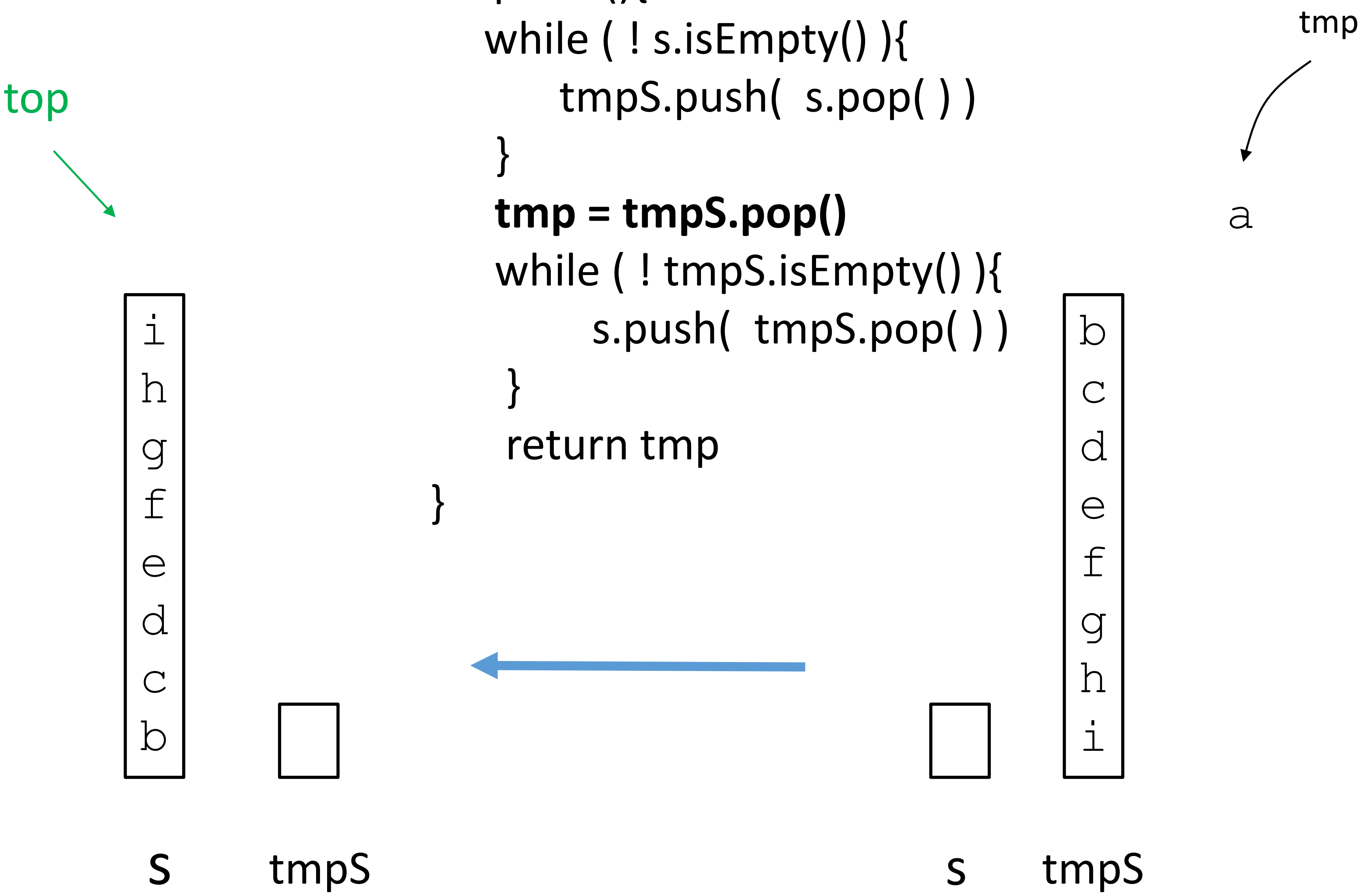
tmp

a

b
c
d
e
f
g
h
i

s   tmpS

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



Time permitting …..

Otherwise Giulia will cover the following next week.

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 next week)

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 (), ….