COMP 250

Lecture 10

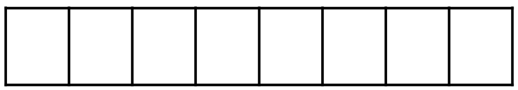
singly linked lists

Jan. 28, 2022

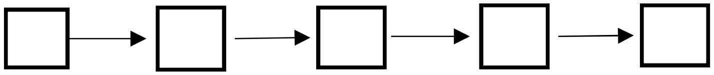
Lists

- array list (last lecture)
- singly linked list (today)
- doubly linked list (next lecture)

array

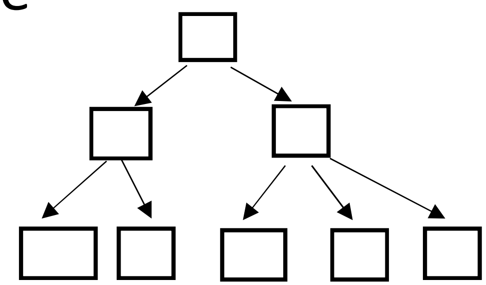


linked list

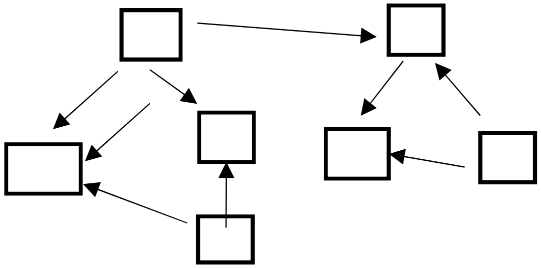


Linked lists are a special case of both of the following which are core topics to be covered later this course:

tree

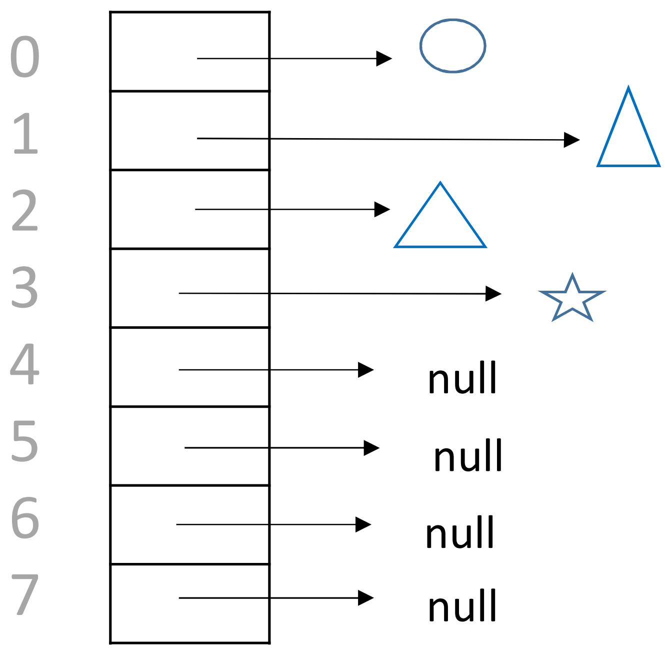


graph



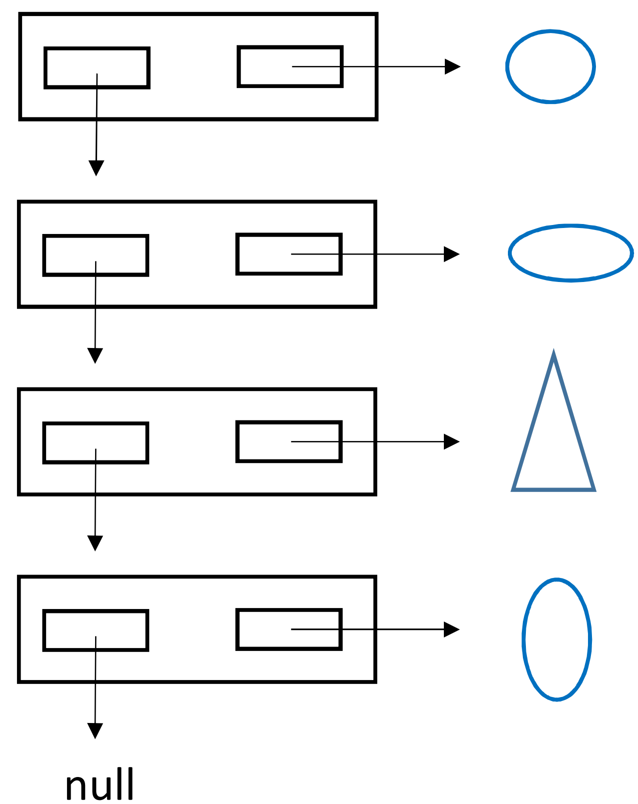
array list

array



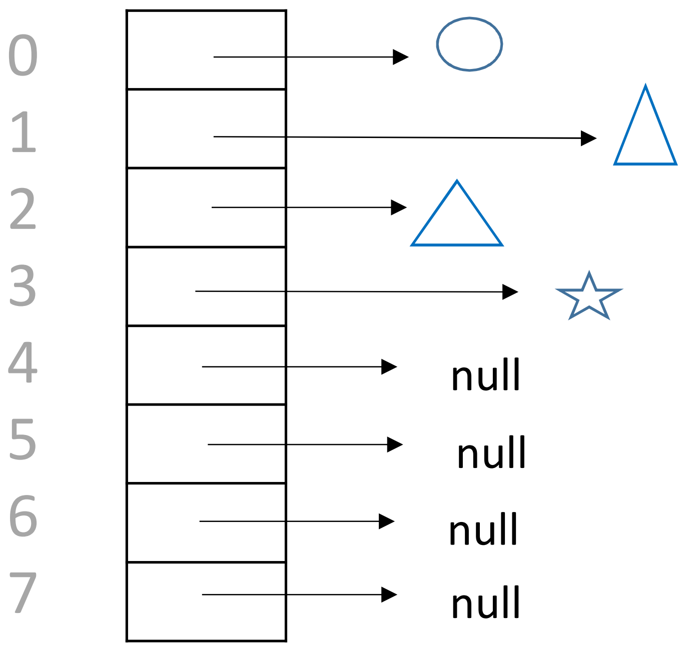
linked list

sequence of “node” objects



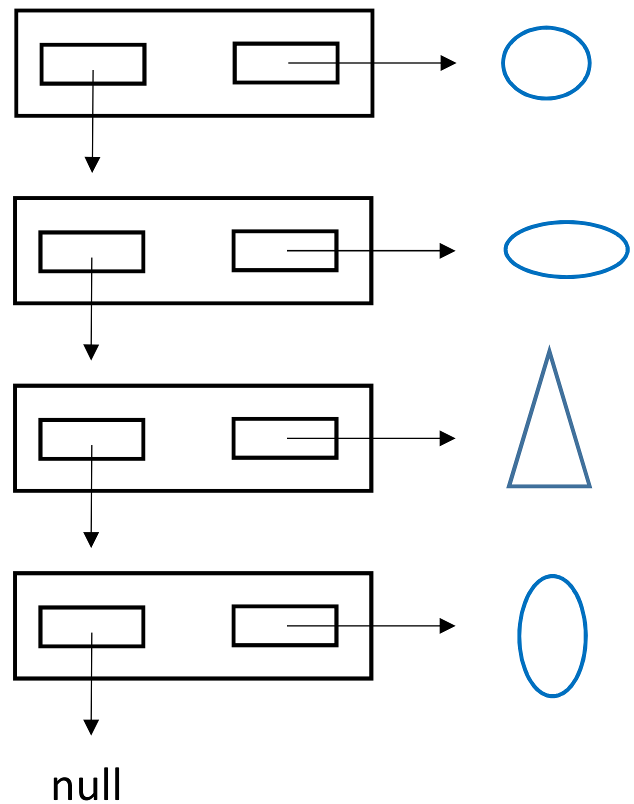
size = 4

array list



Array slots are in *consecutive locations* (addresses) in memory. Objects can be anywhere.

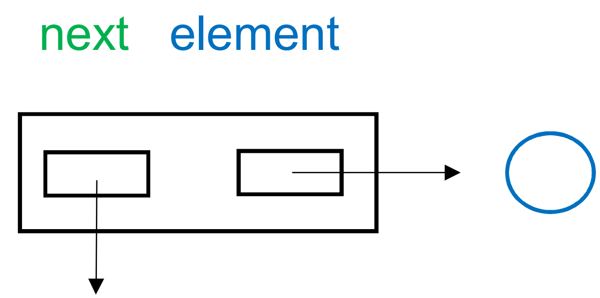
linked list



Linked list “nodes” (and objects that they reference) can be *anywhere in memory*.

Singly linked list node ("S" for singly)

```
class    SNode<E> {

          SNode<E>    next;
          E           element;
          :
}
```

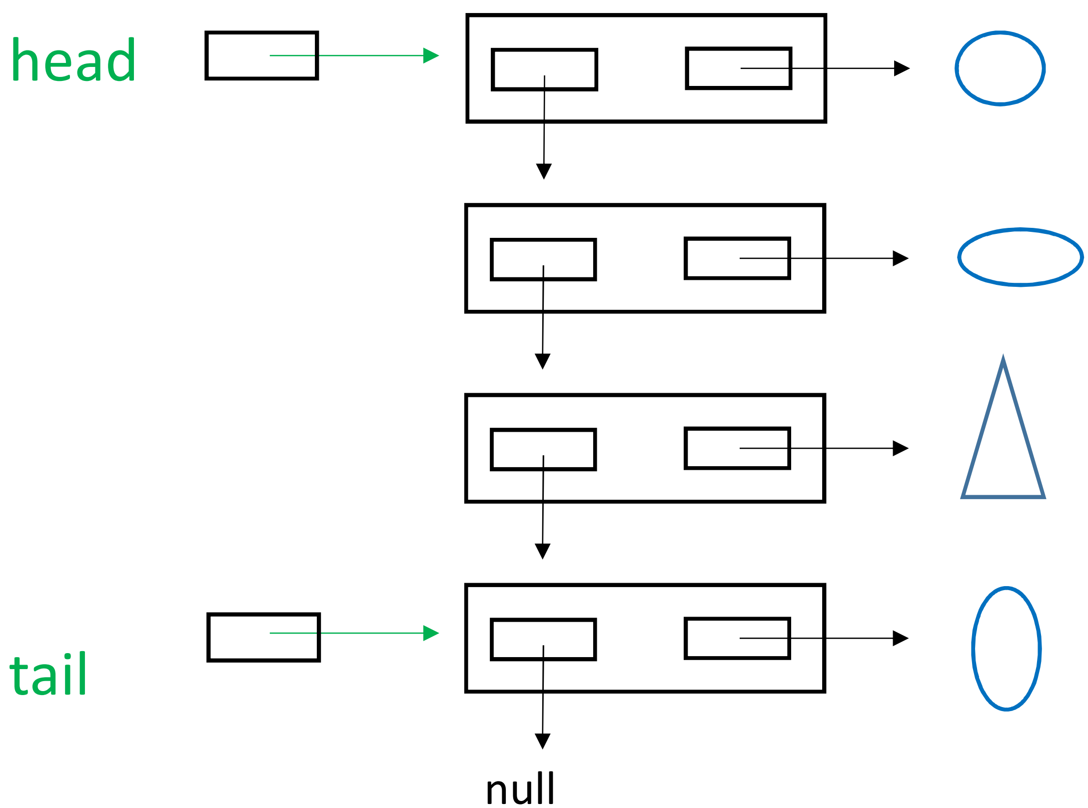


e.g. **E** might be Shape

As we saw last lecture, a generic type **E** *must* be a reference type.

The next field references an `SNode` object, namely the "next" node in the list.

A linked list consists of a sequence of nodes, along with a reference to the first (head) and last (tail) node.



Example: a singly linked class

```
class   SLinkedList<E> {

        SNode<E>   head;
        SNode<E>   tail;
        int        size;

         :

         private   class   SNode<E> {

                   // inner class

                  SNode<E>   next;
                  E          element;
                   :
         }
}
```

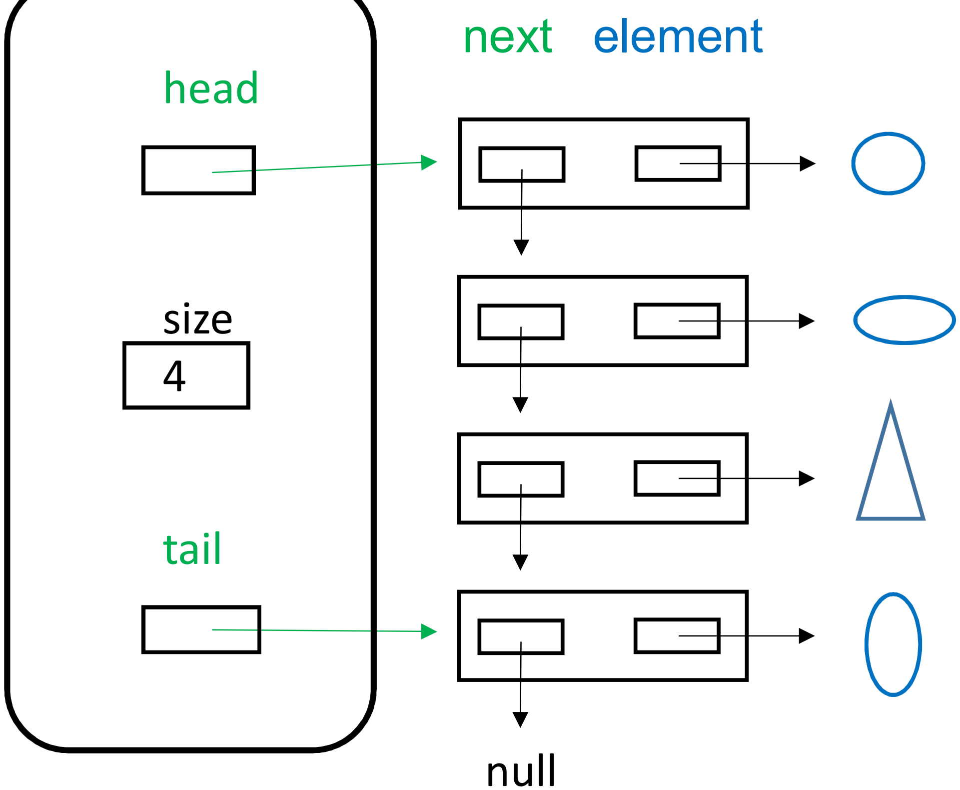
“Inner” (versus “Outer”) Class

```
class    SLinkedList<E> {

         SNode<E>   head;
         SNode<E>   tail;
         int        size;

          :

          private   class   SNode<E> {
                    // inner class

                   SNode<E>   next;
                   E          element;
                    :
          }
}
```

Outer classes have either public or package visibility, but not private.

Often inner classes are private.

How many objects?



1 + 4 + 4 = 9

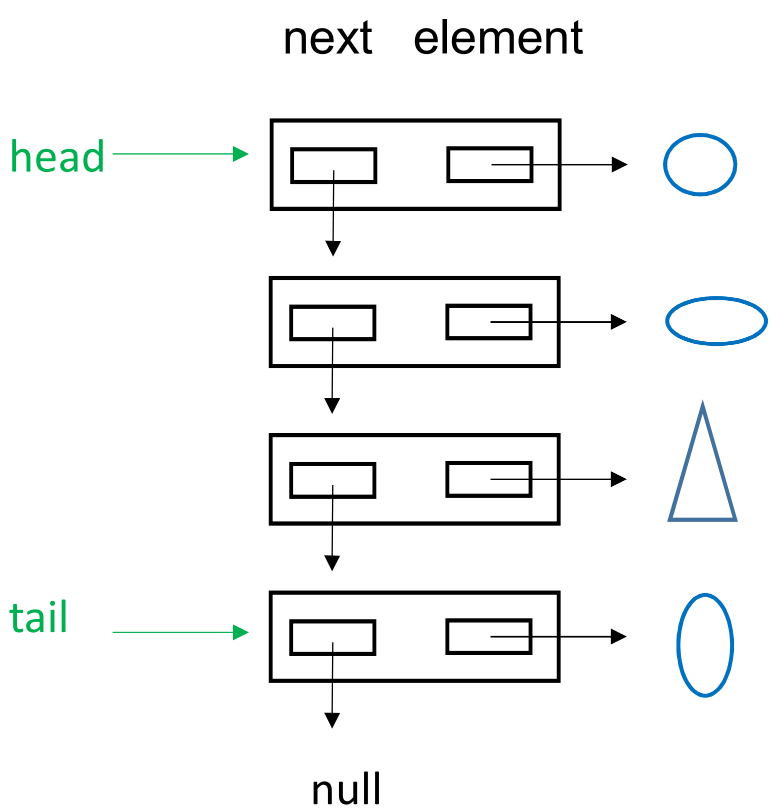
SLinkedList **SNode** **Shape**

Linked list operations

- addFirst (e) - adds new item to front of list
- removeFirst( )
- addLast (e) - adds new item to back (end) of list
- removeLast( )
- …….

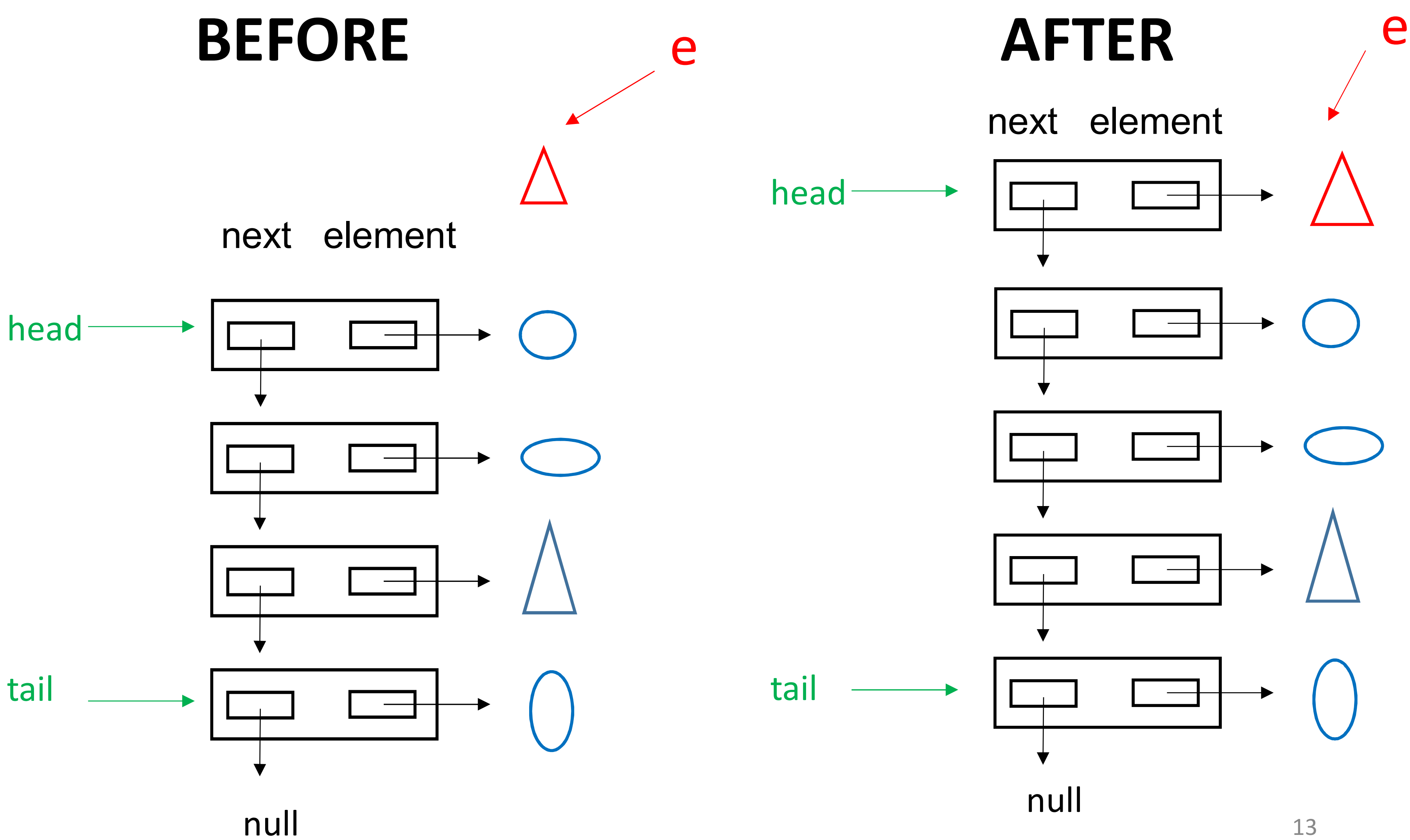
addFirst (e)

BEFORE

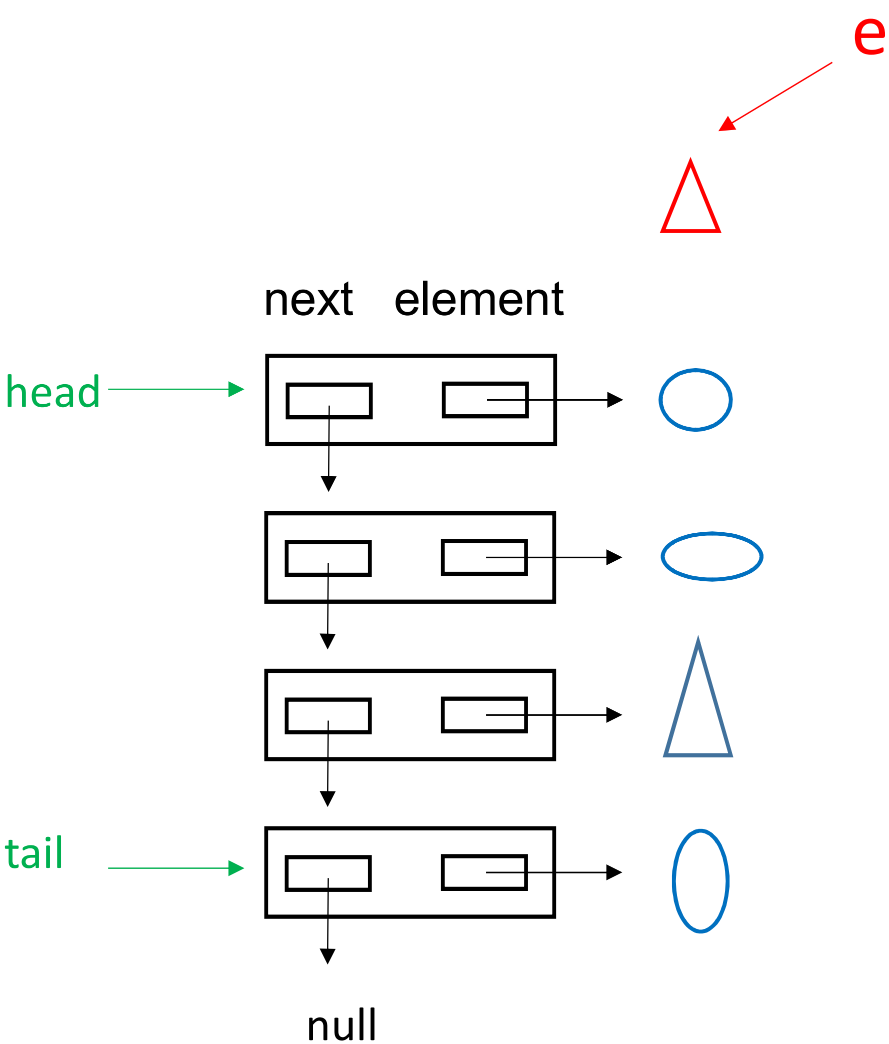


AFTER

addFirst (e)

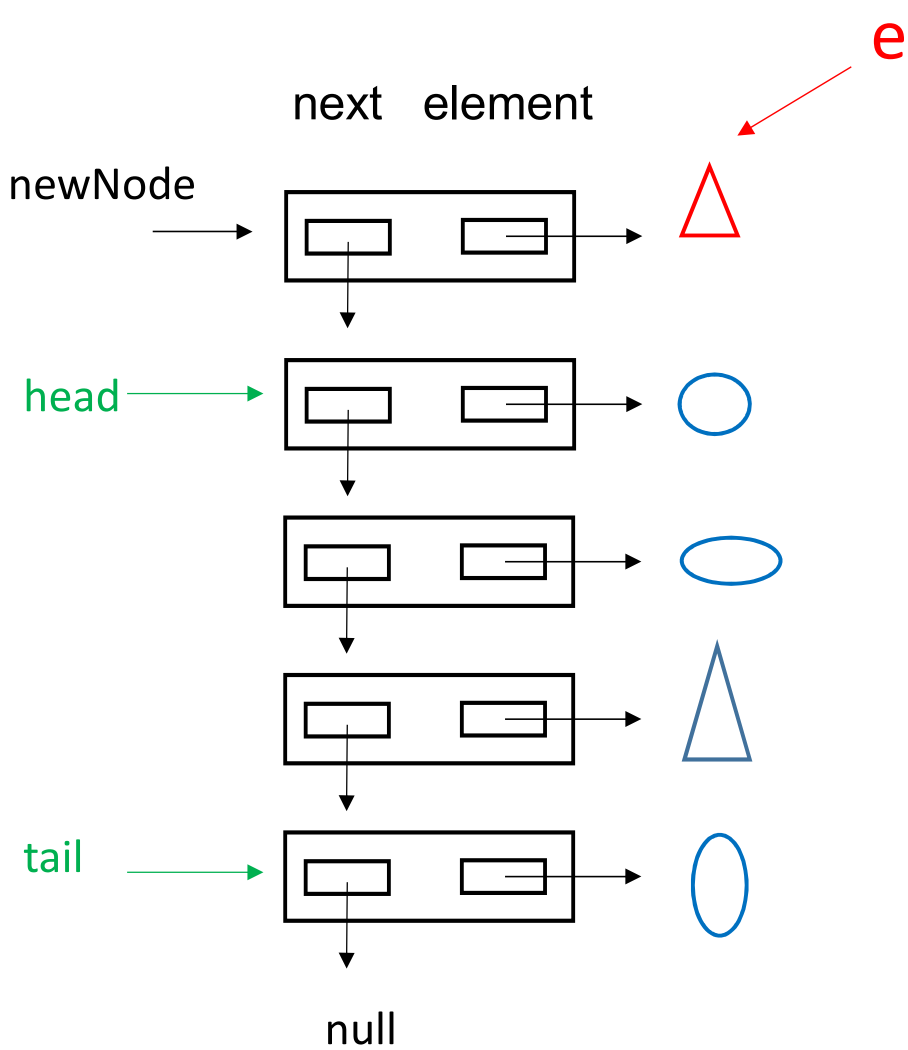

BEFORE
e
next element
head
tail
null
AFTER
e
next element
head
tail
null

addFirst (e)



What to do first ?

addFirst (e)

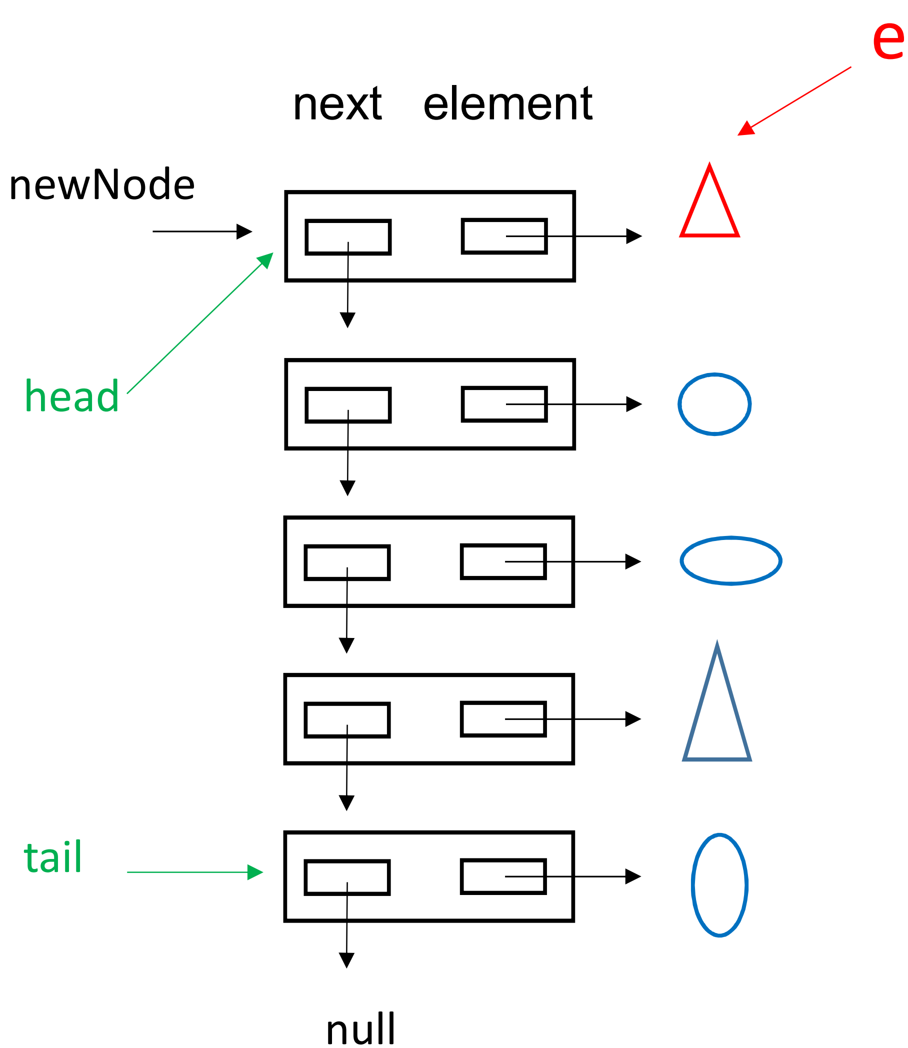


construct newNode

newNode.element = e

newNode.next = head

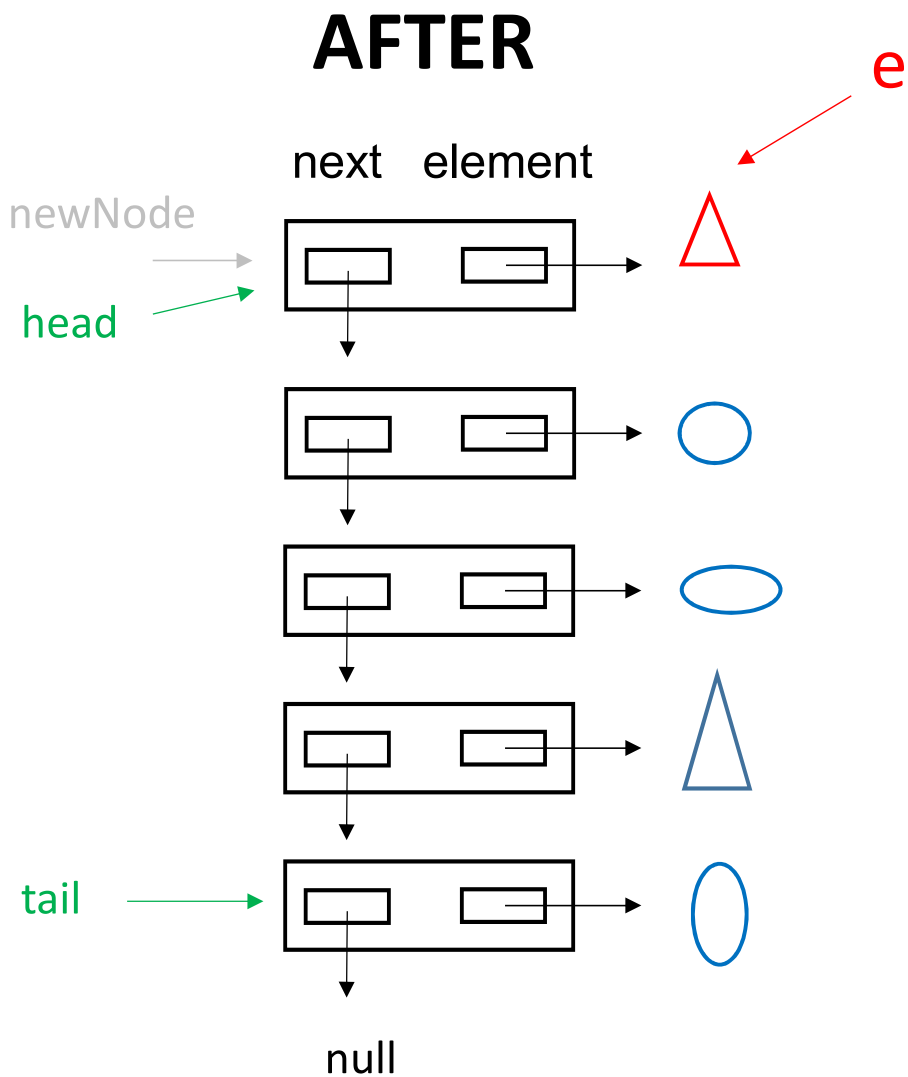
addFirst (e)



construct newNode
newNode.element = e
newNode.next = head

head = newNode
size = size+1

addFirst (e)



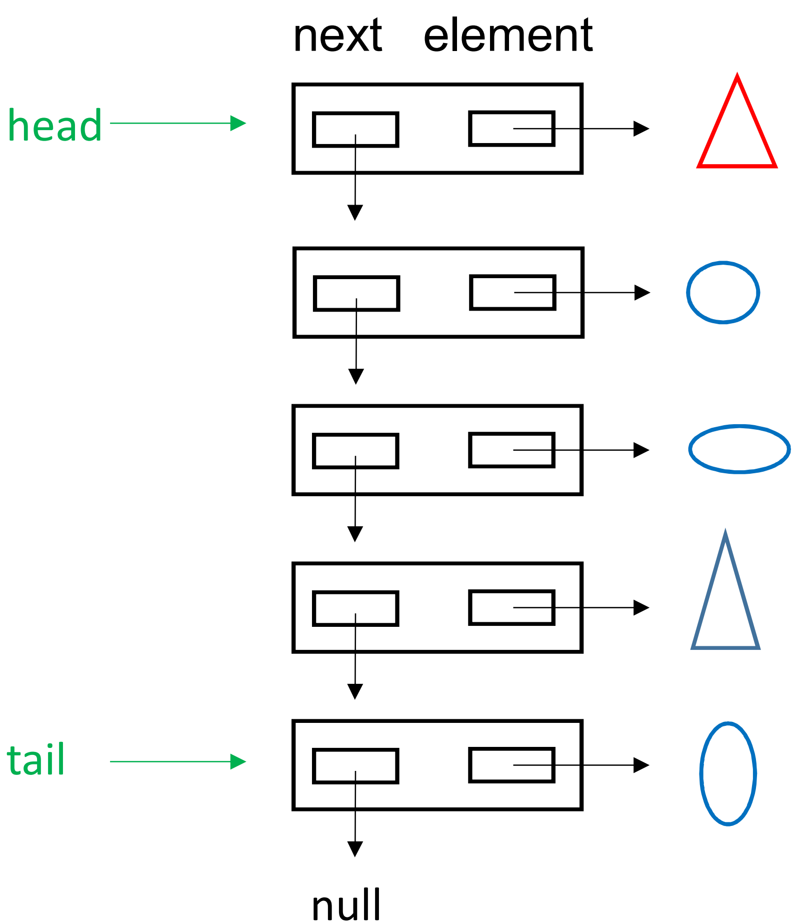
construct newNode
newNode.element = e
newNode.next = head

head = newNode
size = size+1

// special case: list was empty
if size == 1
 tail = head

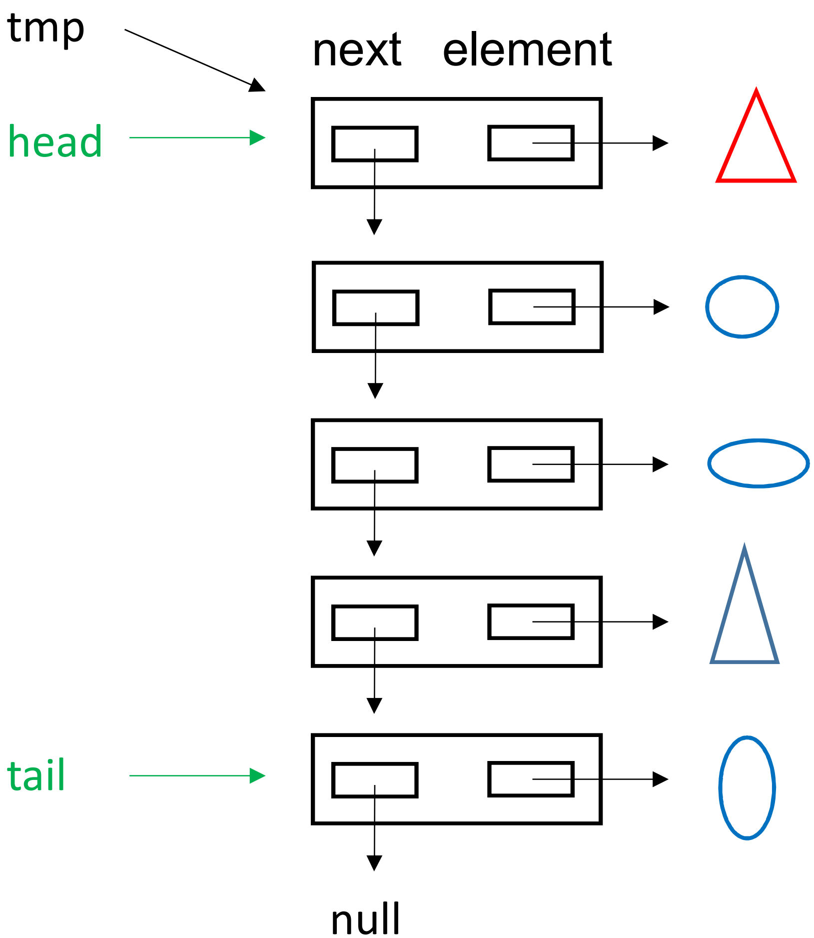
What about **removeFirst ( )** ?

BEFORE



removeFirst ( )

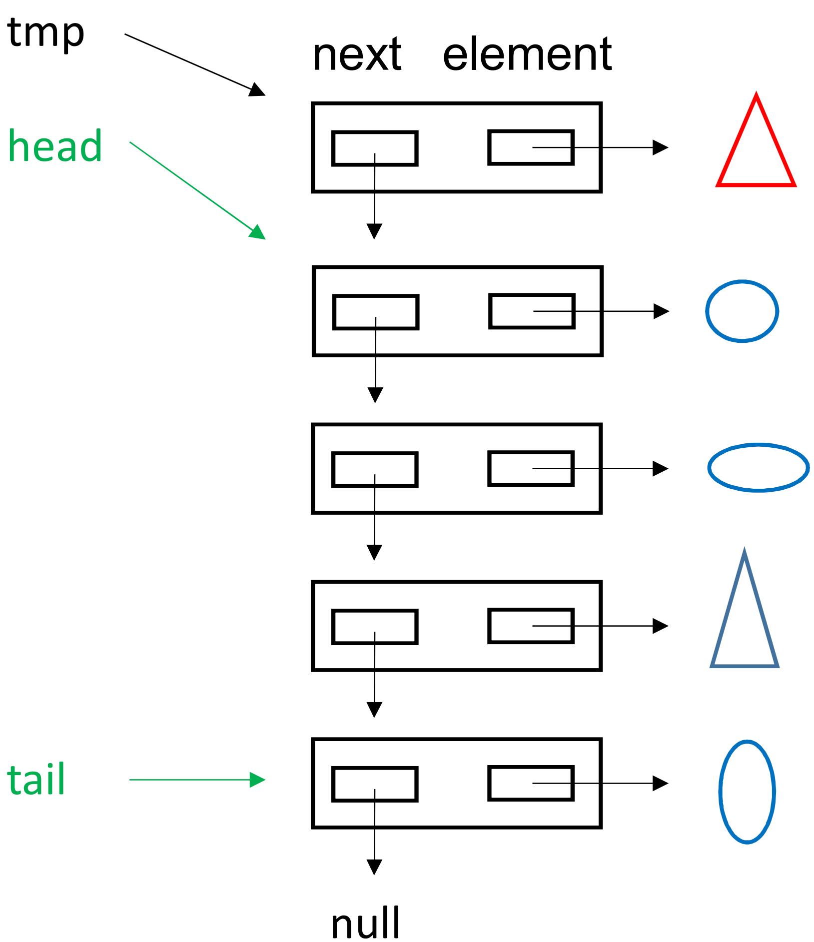
BEFORE



tmp = head
if (size == 0)
  throw exception

return tmp.element

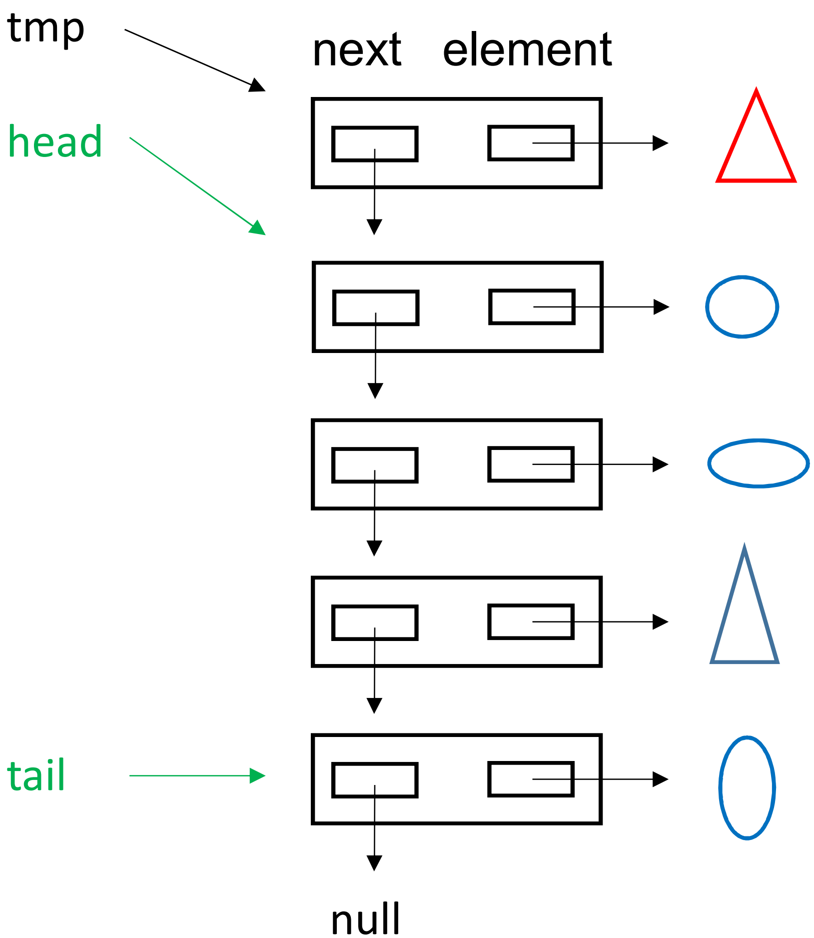
removeFirst ( )




```
tmp  =  head
if (size == 0)
   throw exception

head    =  head.next

return tmp.element
```

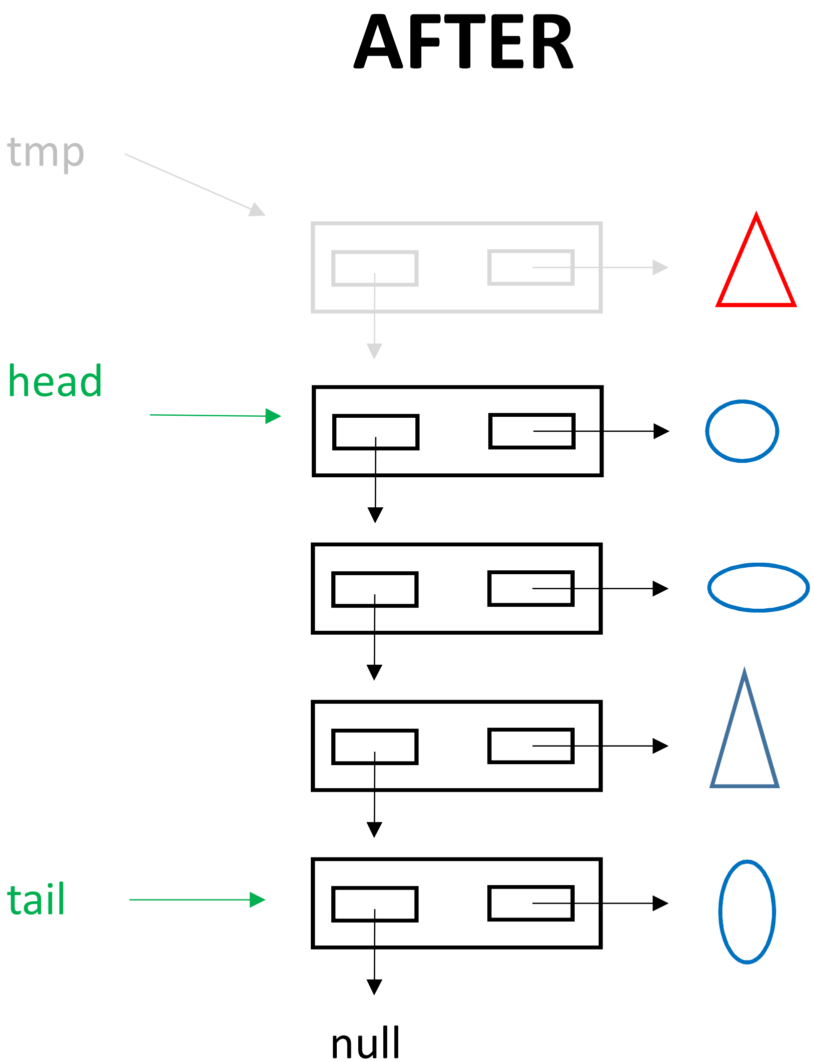
removeFirst ( )

AFTER



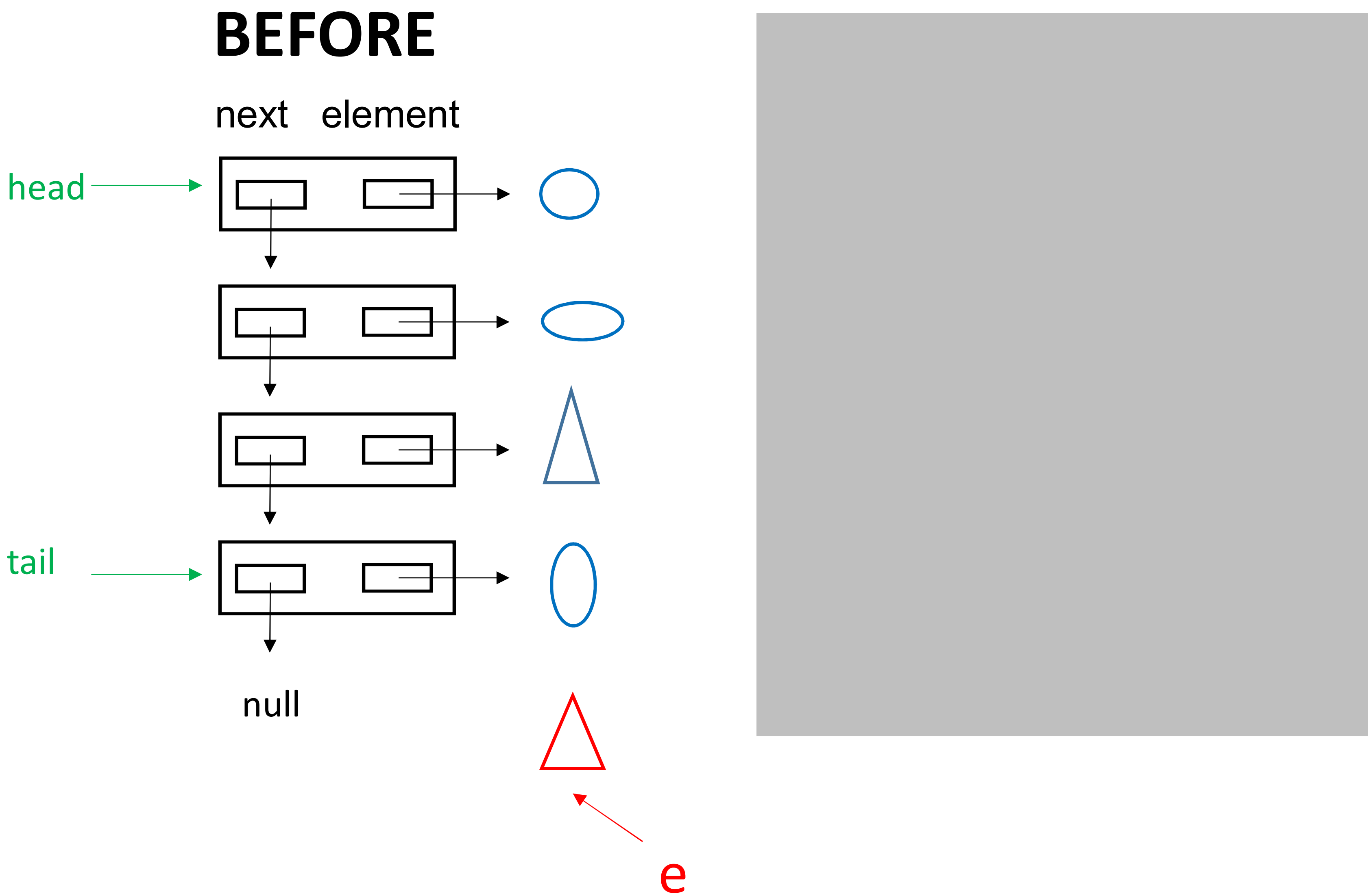

```
tmp  =  head
if (size == 0)
   throw exception

head     =  head.next
size = size – 1

//  special case: list is empty
if  (size == 0)
     tail = null

return tmp.element
```

removeFirst ( )

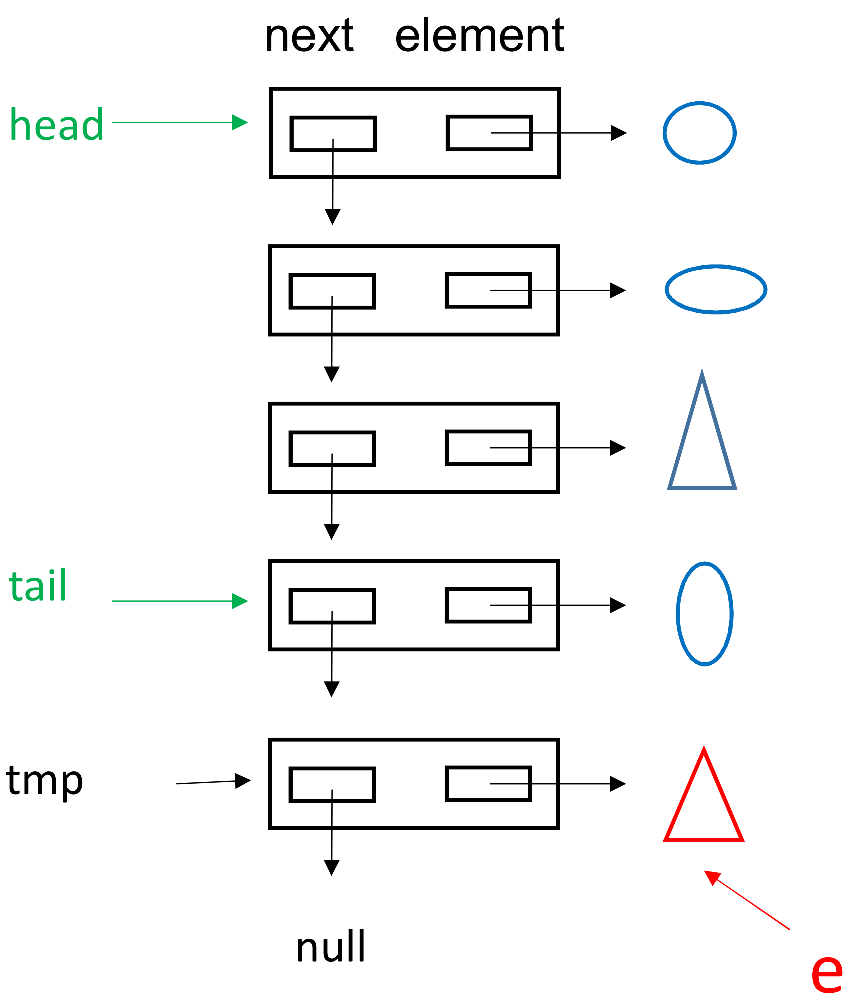


When removeFirst() finishes, the tmp variable is out of scope, and the gray node is garbage (no longer accessible).

addLast (e)



addLast (e)



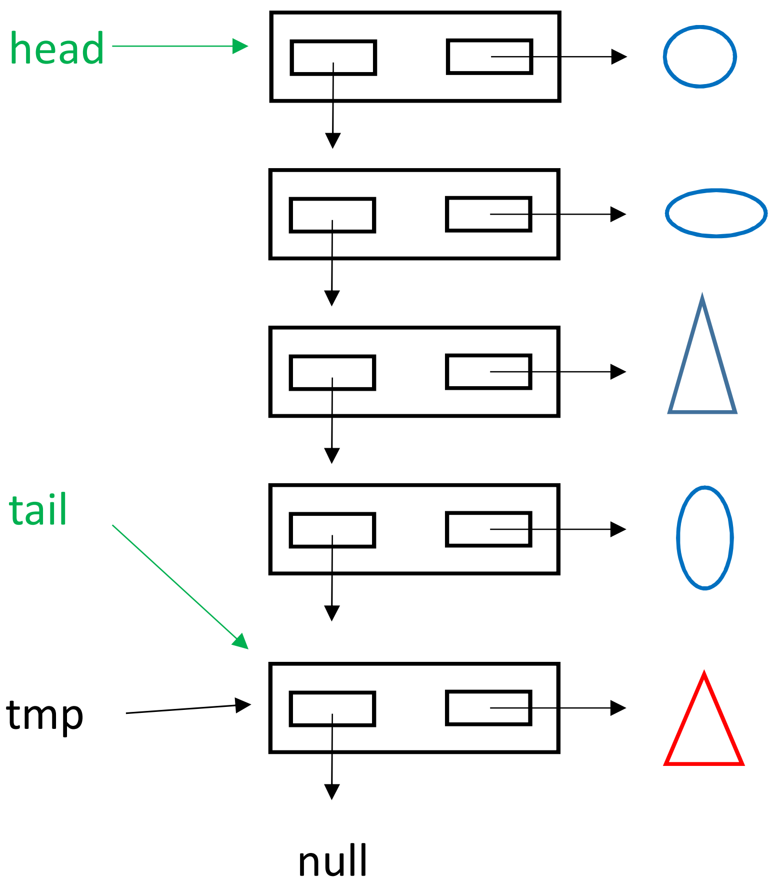
tmp = new Node

tmp.element = e

tail.next = tmp

addLast (e)

AFTER



tmp = new Node

tmp.element = e

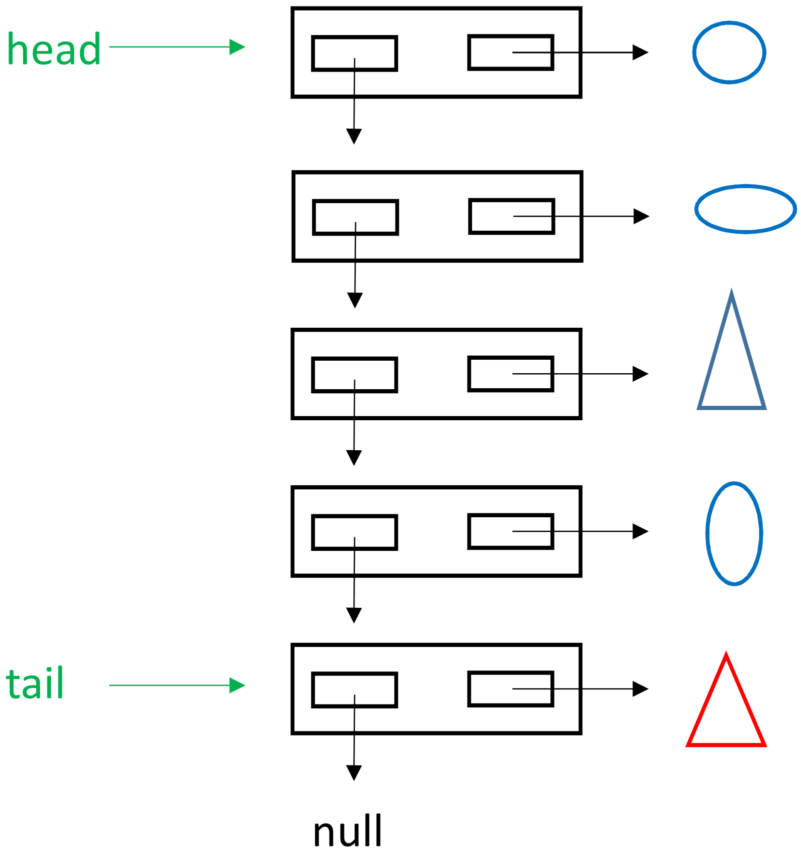
tail.next = tmp

tail = tail.next // or tail = tmp

size = size+1

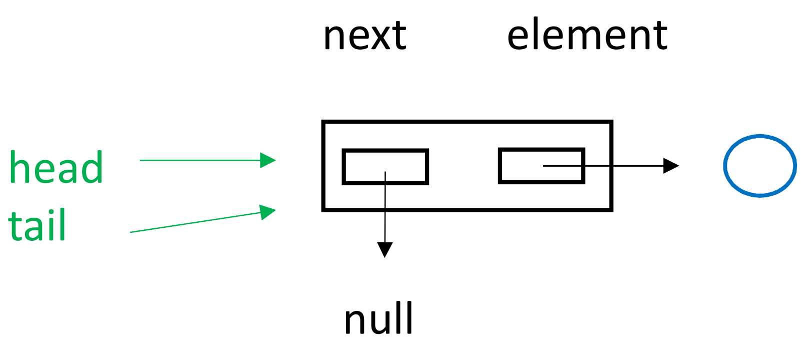
What about removeLast ( ) ?

BEFORE



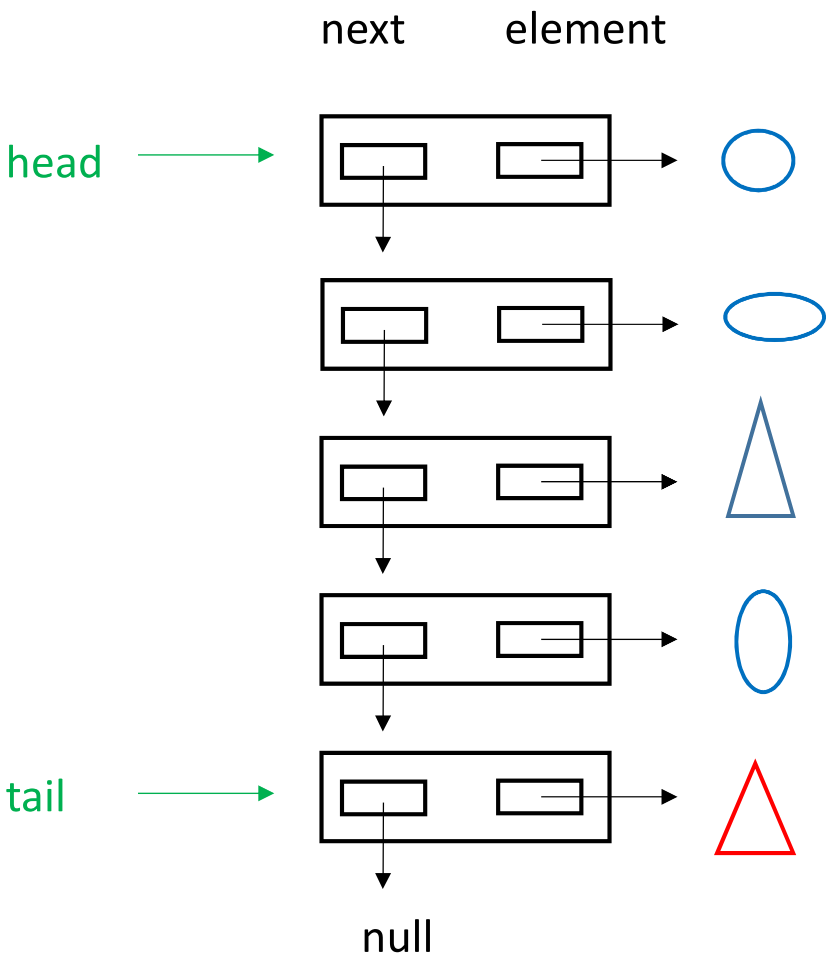
Problem: we have no *direct* way to access the node before tail.

removeLast ( ) → case of size == 1




```
if  (size == 1){
     head = null
     tail     = null
}
else {

}
size = size - 1
```

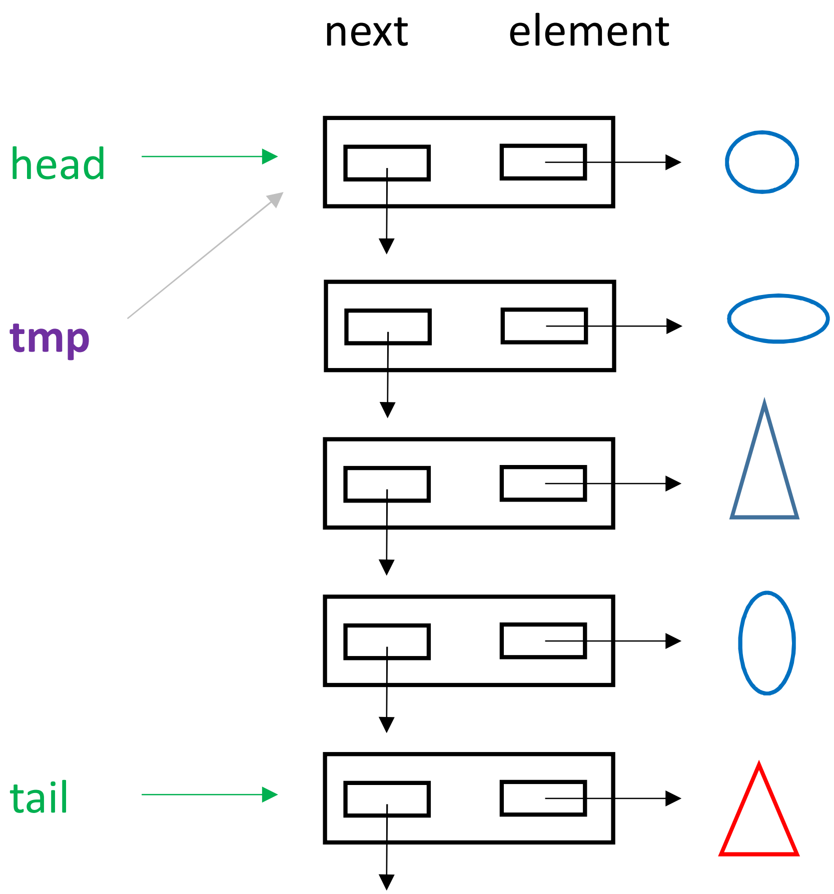
removeLast ( ) general case



if (size == 1){
    head = null
    tail = null
}
else {

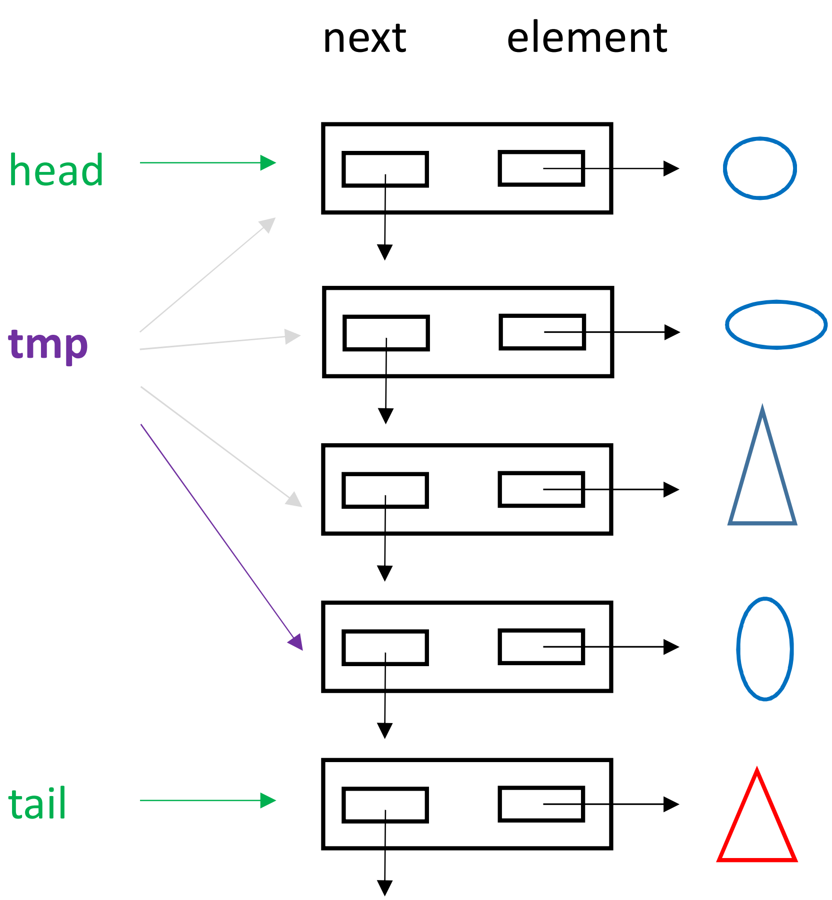
}
size = size - 1

removeLast ( )



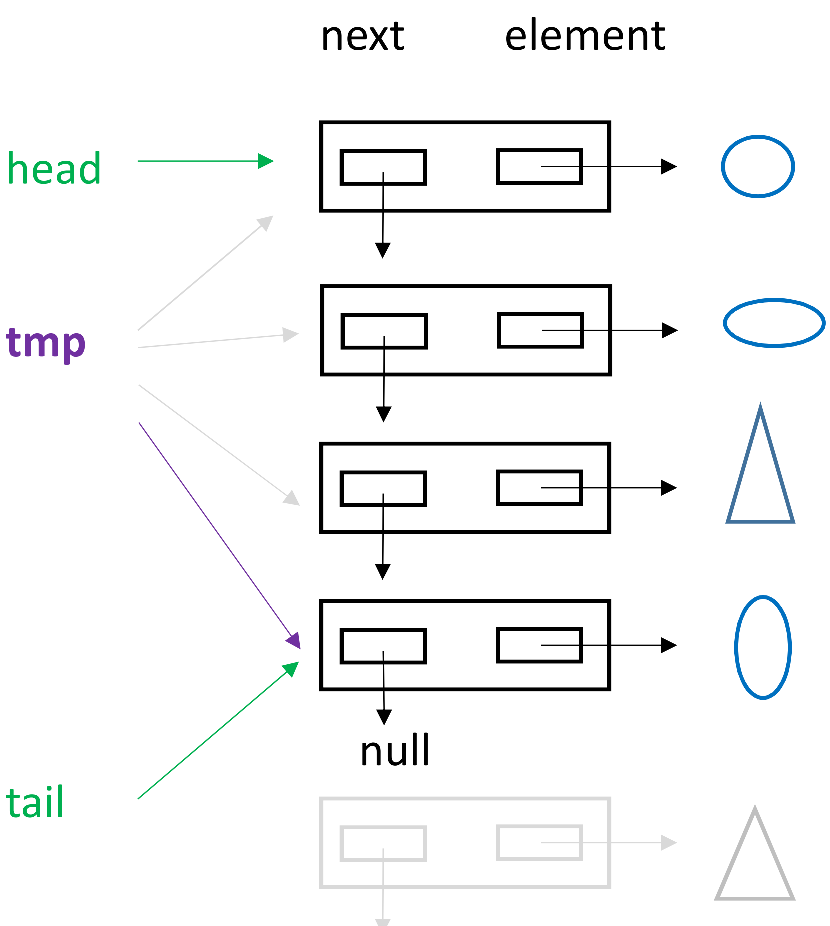

```
if  (size == 1){
    head = null
    tail    = null
}
else {


   tmp  = head

}
size = size - 1
```

removeLast (   )



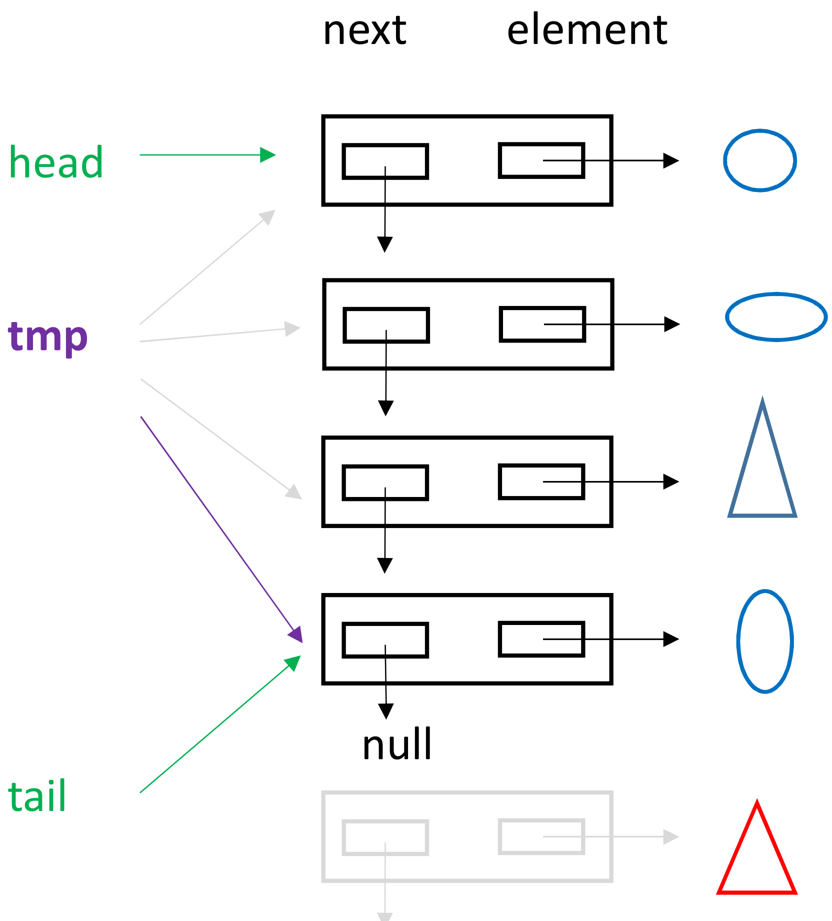

```
if  (size == 1){
     head = null
     tail     = null
}
else {

    tmp  = head
    while ( tmp.next  != tail )
            tmp = tmp.next

}
size = size - 1
```

removeLast (   )



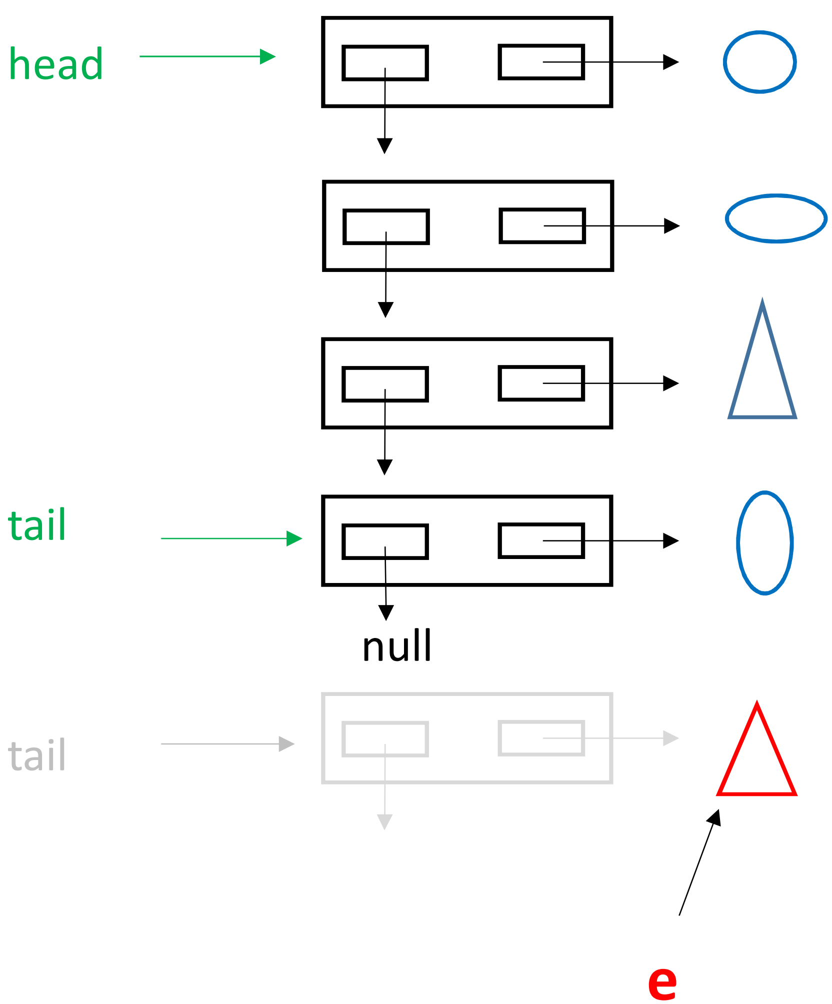

```
if  (size == 1){
     head = null
     tail     = null
}
else {

    tmp  = head
    while ( tmp.next  != tail )
            tmp = tmp.next

    tail = tmp
    tail.next = null
}
size = size - 1
```

removeLast ( )



To return the last element, we need to do a bit more.

removeLast ( )




```
e =  tail.element
if  (size == 1){
     head = null
     tail     = null
}
else {

    tmp  = head
    while ( tmp.next  != tail )
            tmp = tmp.next

    tail = tmp
    tail.next = null
}
size = size – 1
return e
```

Computational Complexity

Let N be the size of the input.

If an operation takes "constant time" (independent of N), then we say it has complexity O(1).

If an operation takes time proportional to N, then we say it has complexity O(N).

If an operation takes time proportional to N^2, then we say it has complexity O(N^2).

Computational Complexity (N = list size)

	array list	SLinkedList
addFirst		?
removeFirst		?
addLast		
removeLast		

Computational Complexity (N = list size)

	array list	SLinkedList
addFirst		O(1)
removeFirst		O(1)
addLast		?
removeLast		?

Computational Complexity (N = list size)

	array list	SLinkedList
addFirst	?	O(1)
removeFirst	?	O(1)
addLast		O(1)
removeLast		O(N)

Computational Complexity (N = list size)

	array list	SLinkedList
addFirst	O(N)	O(1)
removeFirst	O(N)	O(1)
addLast		O(1)
removeLast		O(N)

An array list would use add(0, e)
rather than addFirst(e) and
remove(0) instead of removeFirst(), ….

Computational Complexity (N = list size)

	array list	SLinkedList
addFirst	O(N)	O(1)
removeFirst	O(N)	O(1)
addLast	O(1)	O(1)
removeLast	O(1)	O(N)

If the array is full, then it still takes time O(N). We need to make a bigger array and then write N+1 elements to the bigger array.

But only if there is available space. Otherwise it takes time O(N).

Coming up...

Lectures

Mon. Jan. 31

Doubly Linked Lists

Wed. Feb. 2

Quadratic Sorting

Fri. Feb. 4

Object Oriented Design 1

(Inheritance)

Assessments

Today

Quiz 1 (lectures 1-7)

Assignment 1

- you will have 2 weeks to do it