COMP 250

Lecture 11

doubly linked lists

Jan. 31, 2022

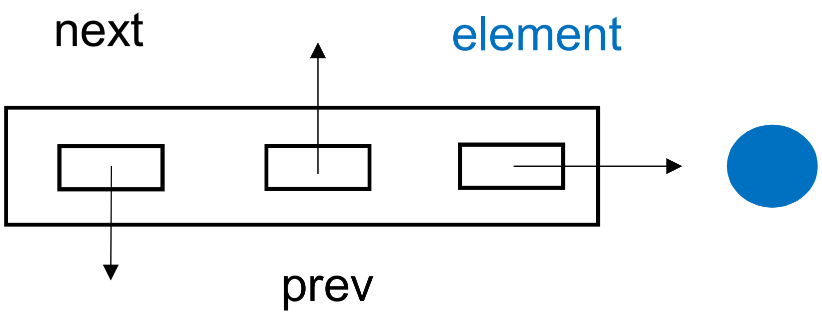
Lists

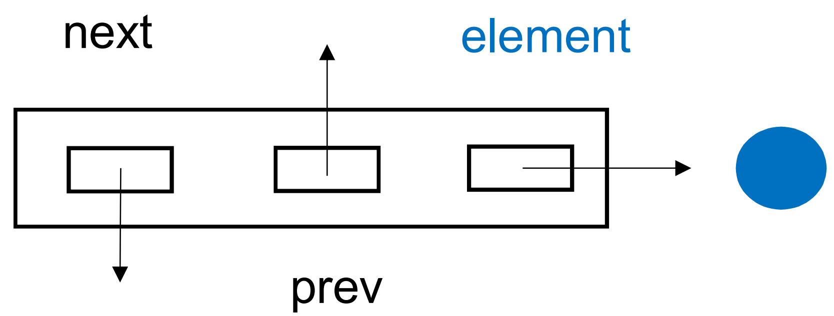
- array list
- singly linked list
- doubly linked list

  :

Doubly linked list

Each node in the list has a reference to the next node and to the previous node, and to an element object.

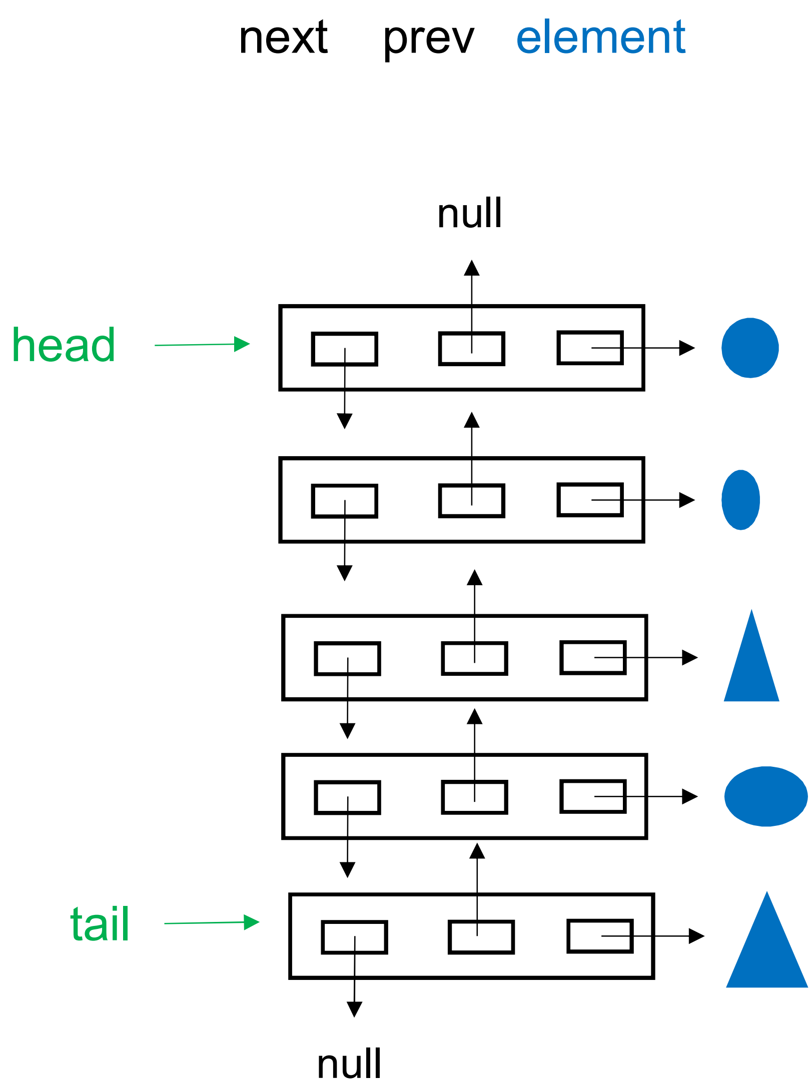





```
class DNode< E > {

        DNode< E >      next;
        Dnode< E >      prev;
        E               element;

       // constructor

        DNode( E      e ) {
              element = e;
              prev = null;
              next = null;
        }
}
```

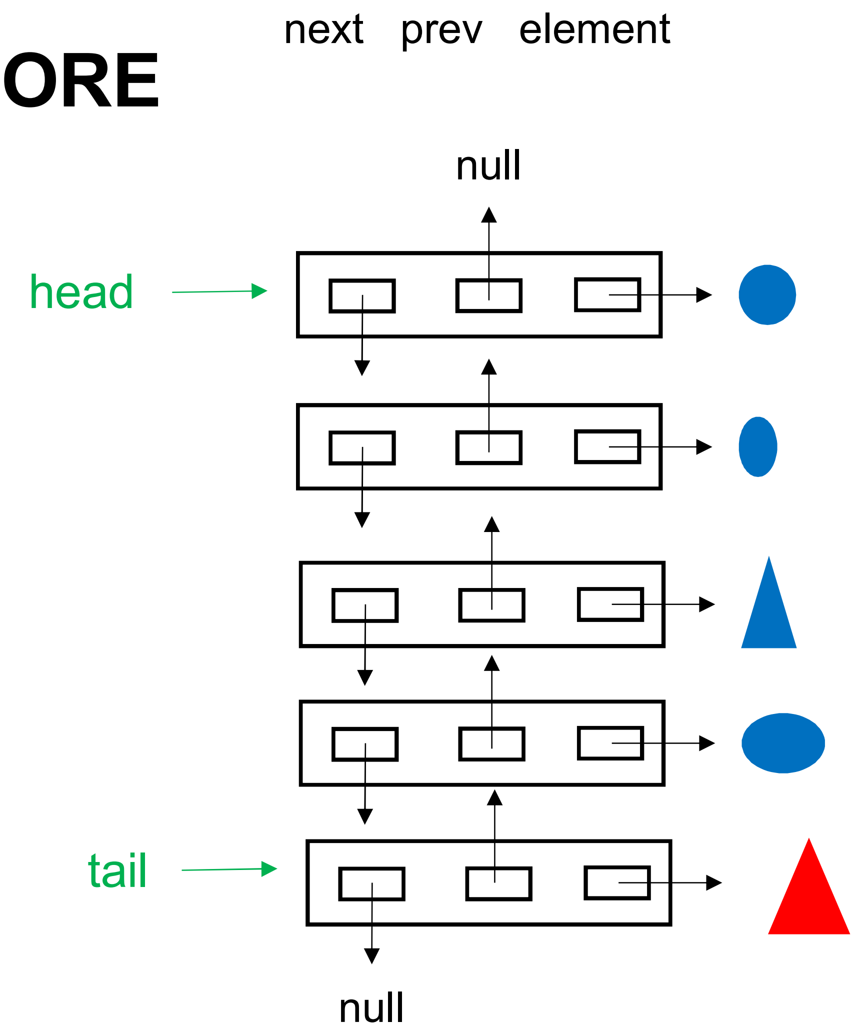
Doubly linked list

As with a singly linked list, the doubly list list uses a head and tail reference.



For a doubly linked list, removing the last element is fast.

BEFORE



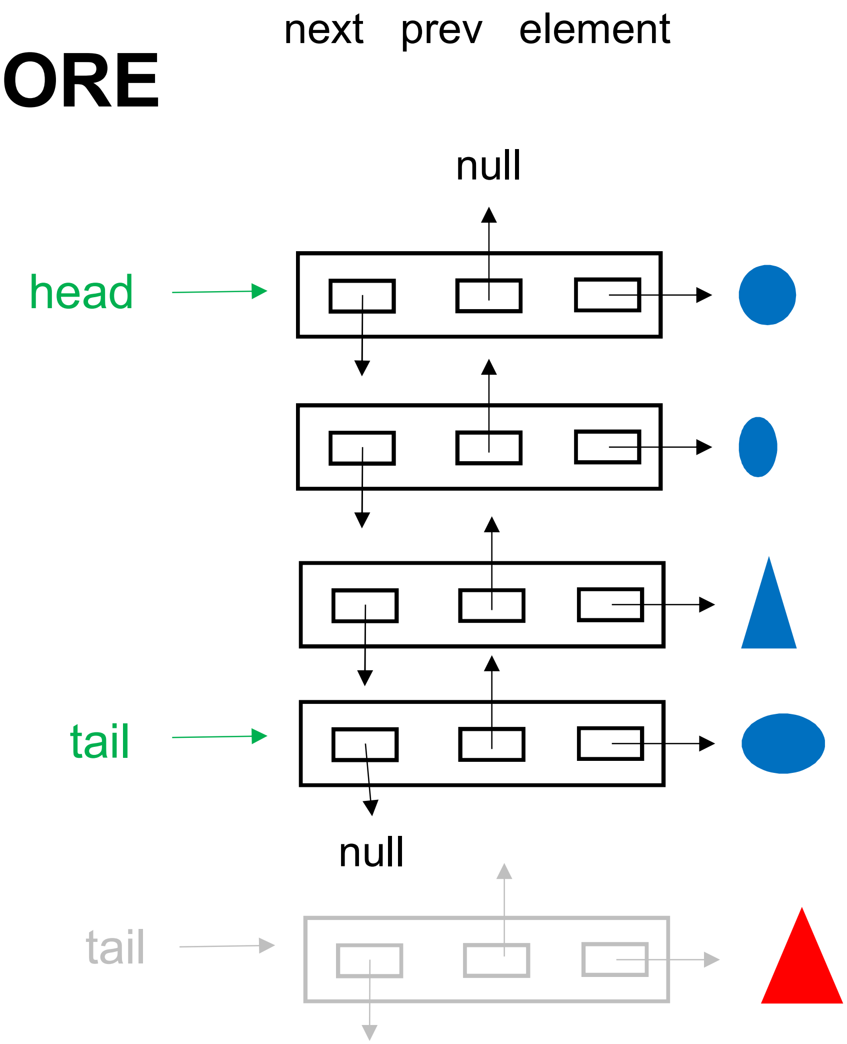
removeLast(){

}

Unlike for a singly linked list, removing the last element of a doubly linked list is fast.

BEFORE

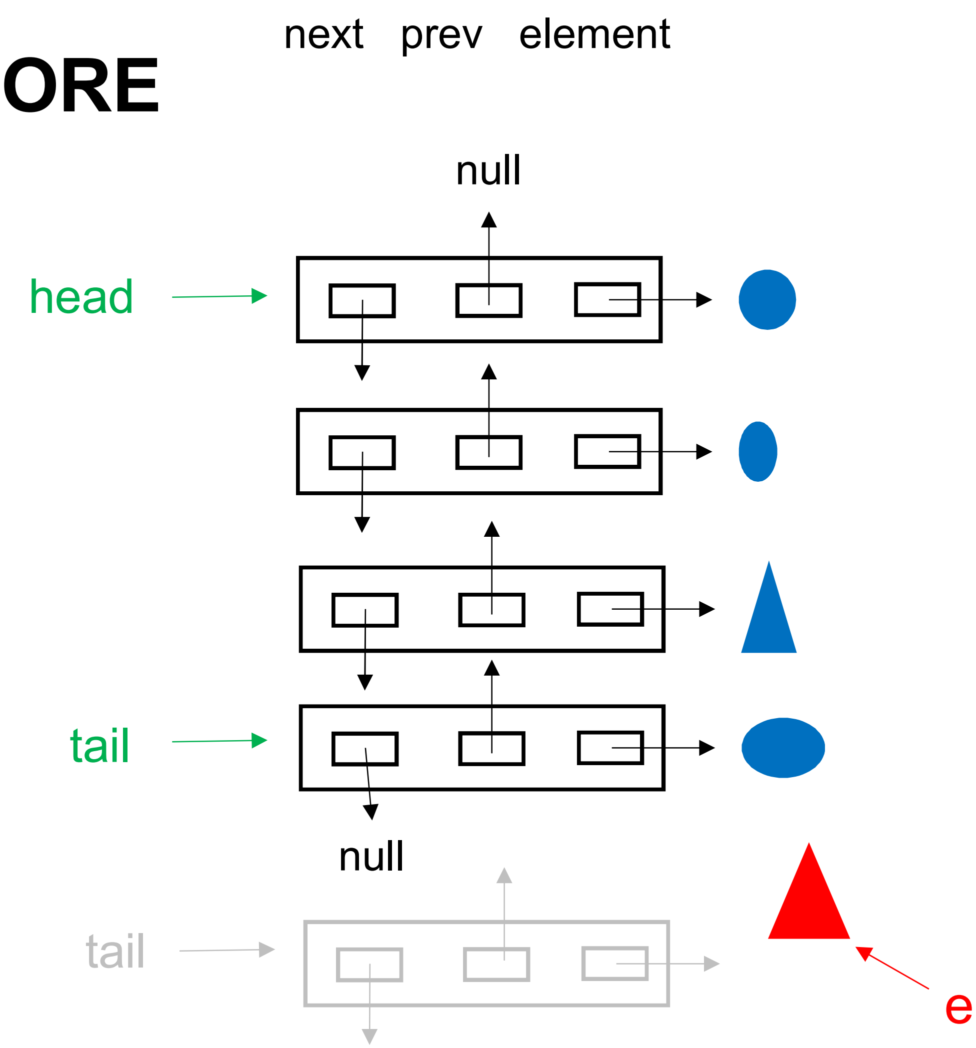
```
removeLast(){
        tail      = tail.prev
        tail.next = null
        size = size – 1
}
```



Unlike for a singly linked list, removing the last element of a doubly linked list is fast.

BEFORE

```
removeLast(){
      e            = tail.element
      tail         = tail.prev
      tail.next = null
      size = size – 1
      return e
}
```



Suppose we want to access node i in a doubly linked list.

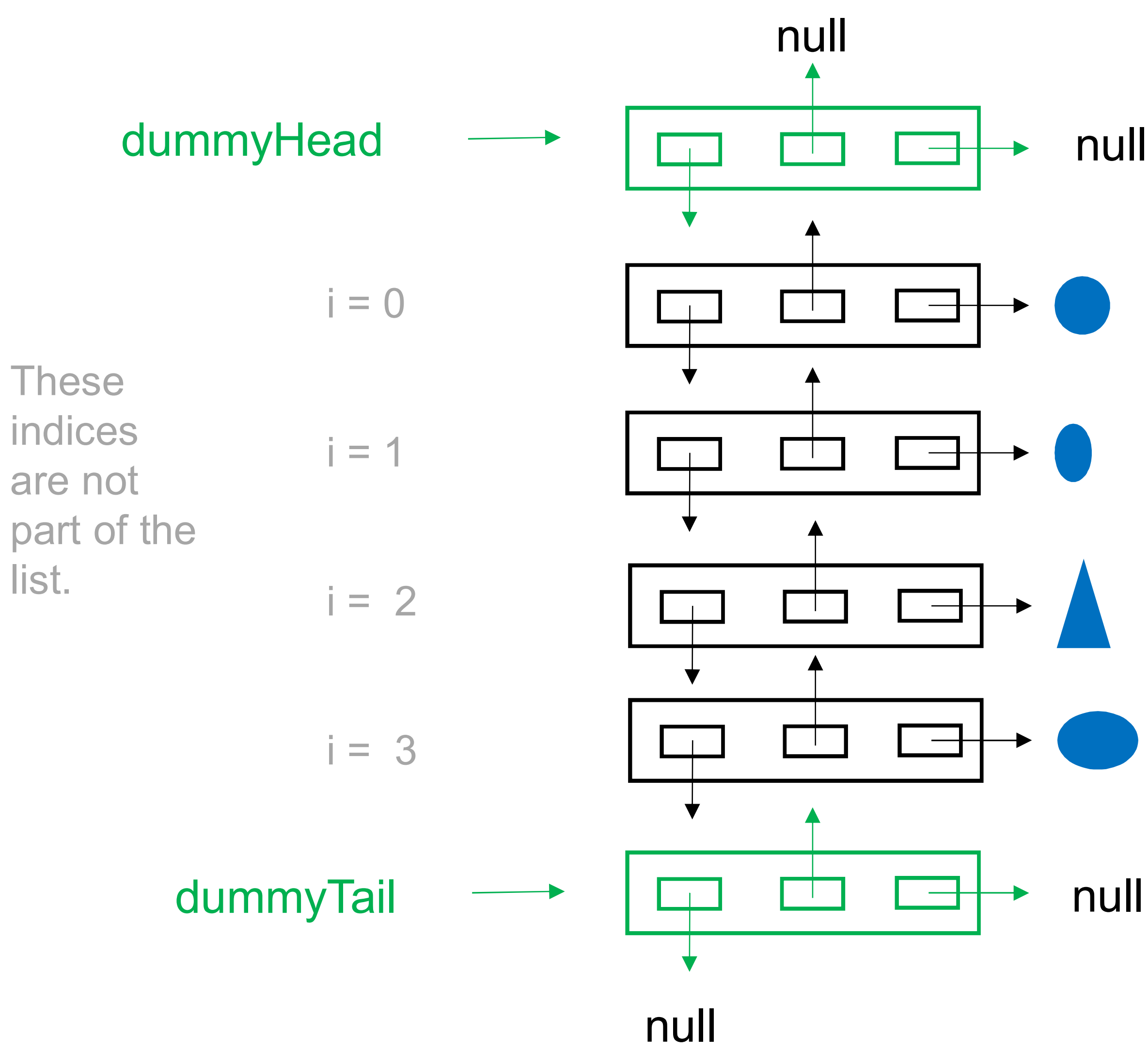
One issue is that edge cases (i = 0, i = size – 1) require special treatment: node 0 has a null prev field and node size-1 has a null next field.

We would like to avoid testing special cases for each method, since this is error prone.

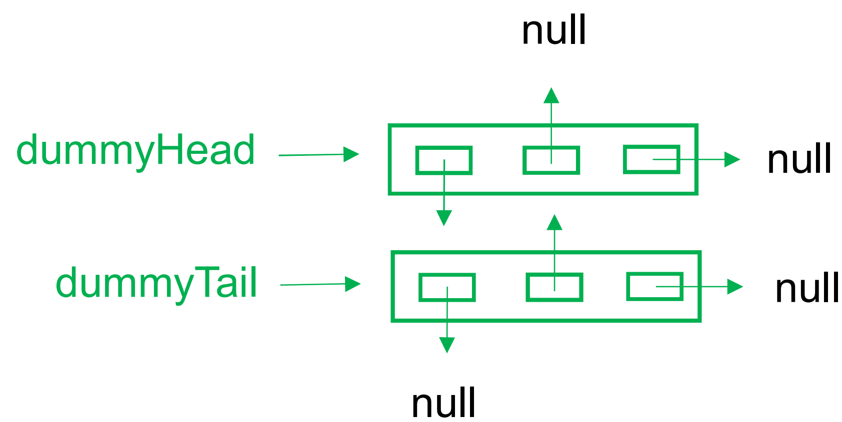
For example, in the removeLast() method on the last slide, what if there was only one node? That code would not work. We forgot to adjust head!

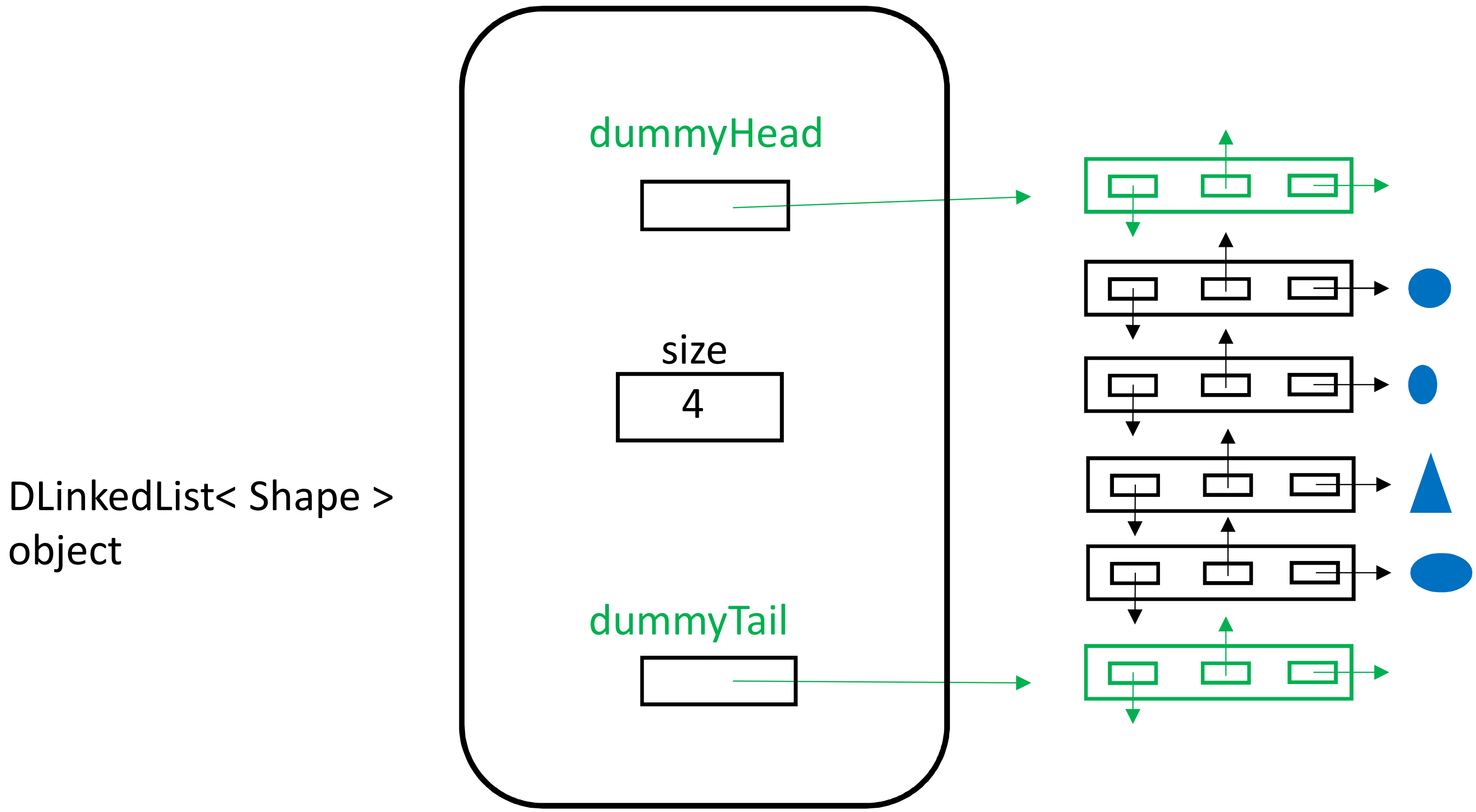
[ADDED Feb. 10] Moreover, the instruction **tail.next = null** *would cause a null pointer exception.]*

Avoid edge cases with "dummy nodes"



```java
class  DLinkedList<E>{    //  Java code

    DNode<E>        dummyHead;
    DNode<E>        dummyTail;
    int             size;
     :

    //  constructor

    DLinkedList<E>(){
        dummyHead =   new   DNode<E>();
        dummyTail   =   new   DNode<E>();
        dummyHead.next   =   dummyTail;
        dummyTail.prev     =   dummyHead;
        size      = 0;
    }

    private class  DNode<E>{ … }
}
```





Q: How many objects in total in this figure?

A: 1 + 6 + 4 = 11

Other List Operations

Many list operations require access to node i.

(This is so for singly linked lists also.)

:

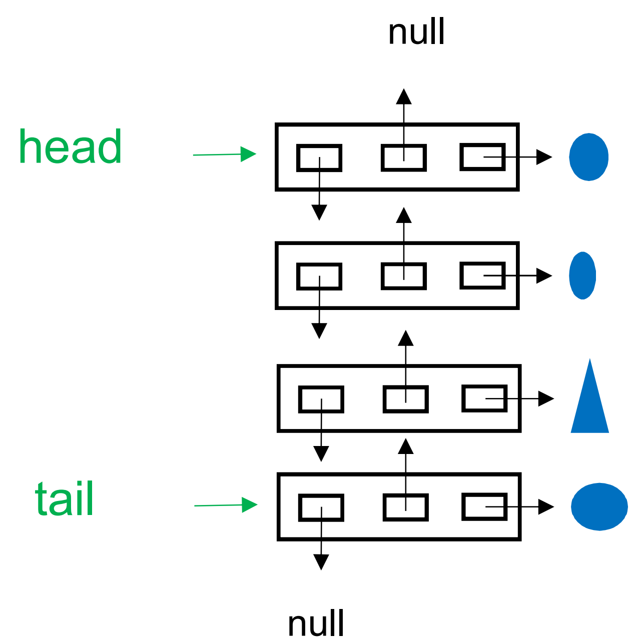
get(i)

set(i,e)

add(i,e)

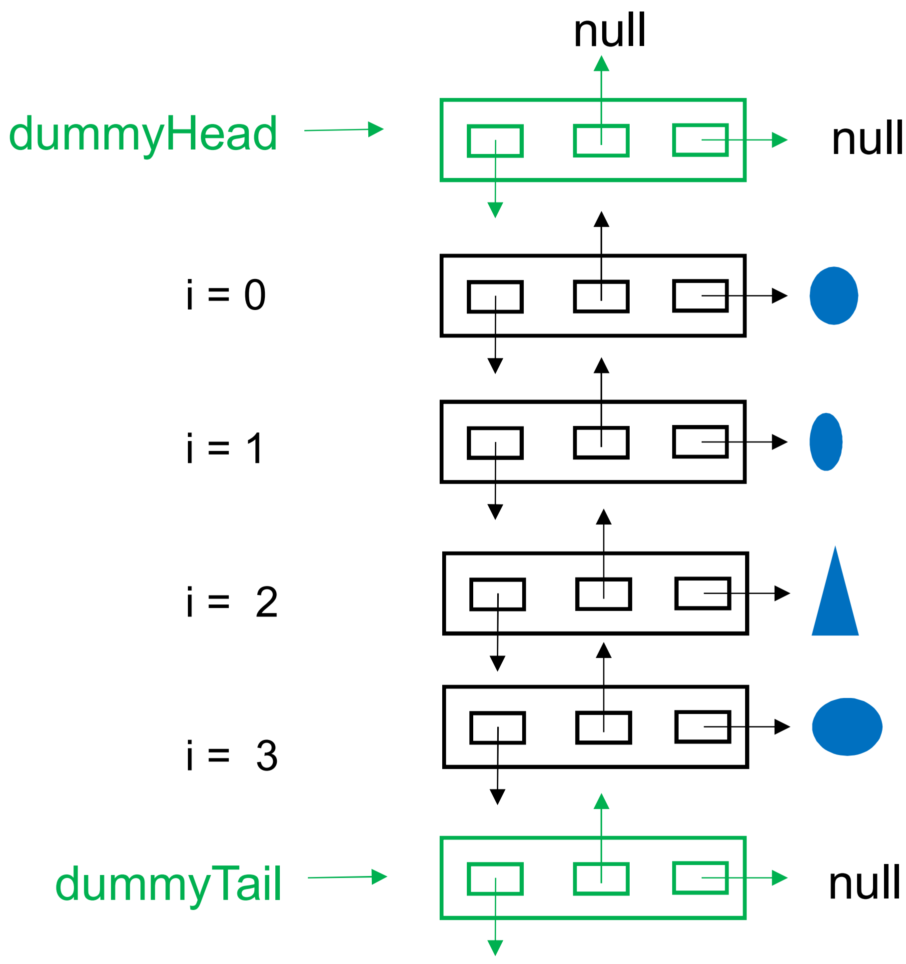
remove(i)

:



get(i) {   // returns the *element* at index i of list

}



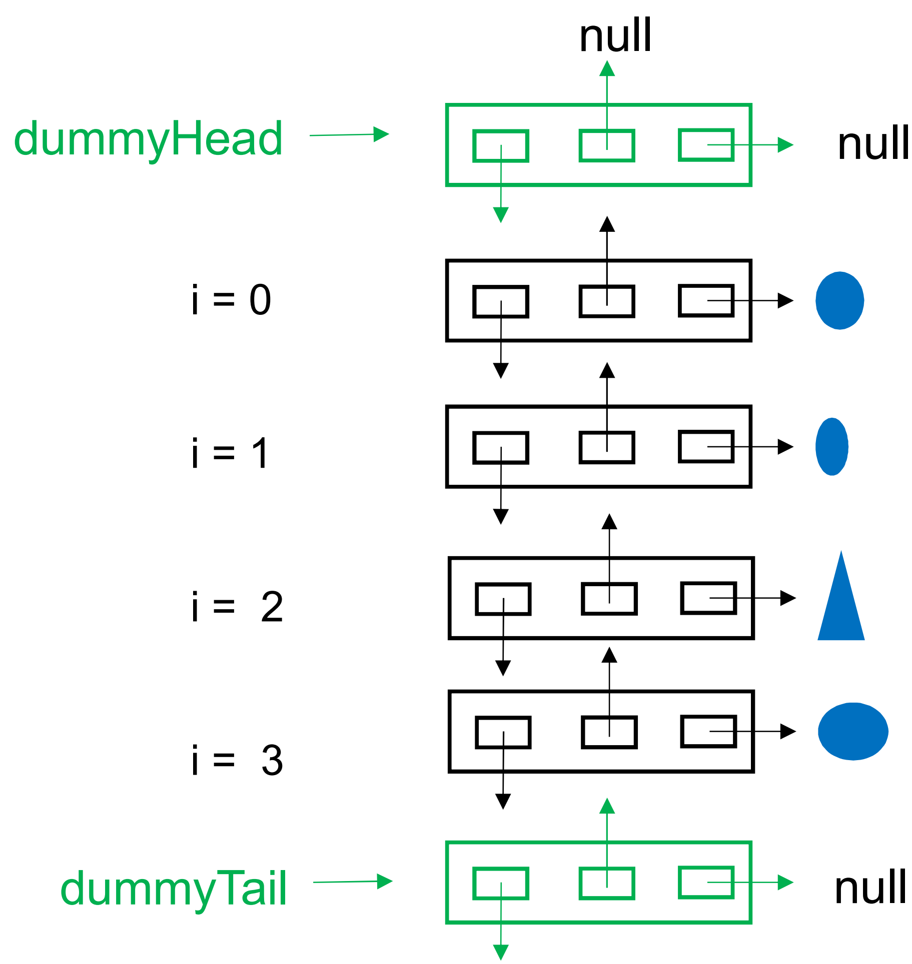
get(i) {   // returns the element at index i of list

    return getNode(i).element

}

getNode() is a helper method discussed on next slide

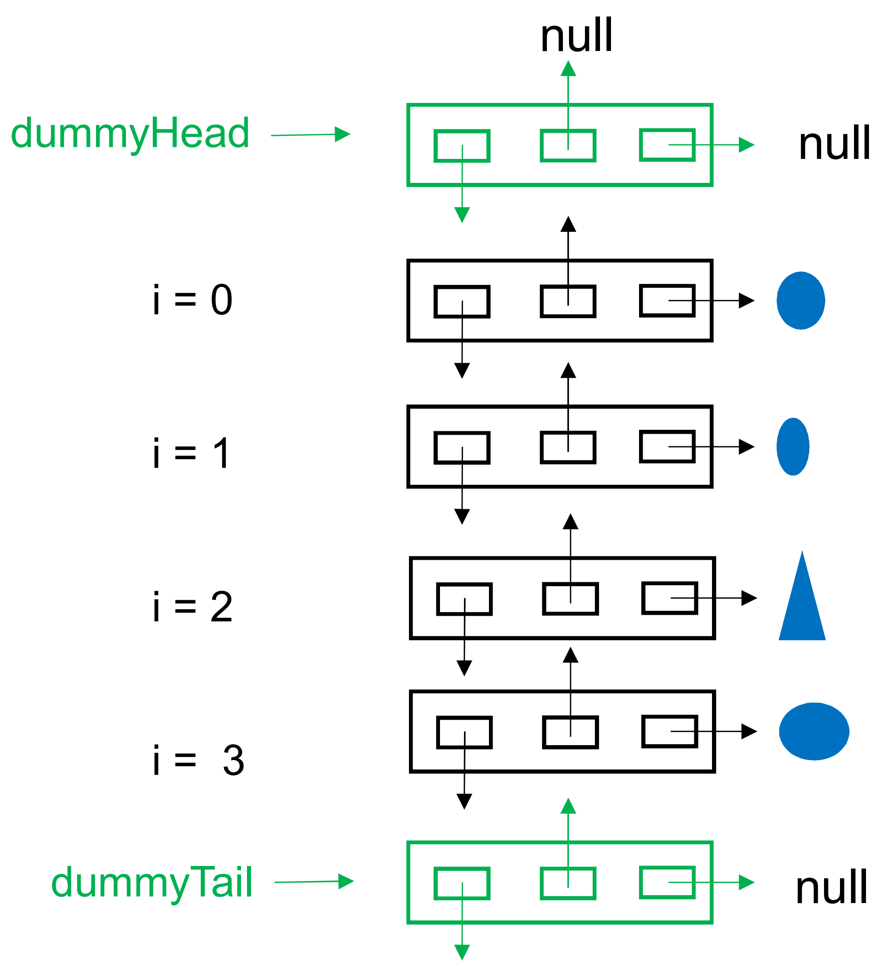
In Java, it would normally be a private method.



getNode(i) {   // helper,   returns a DNode

// Omitting verification that 0 <= i < size

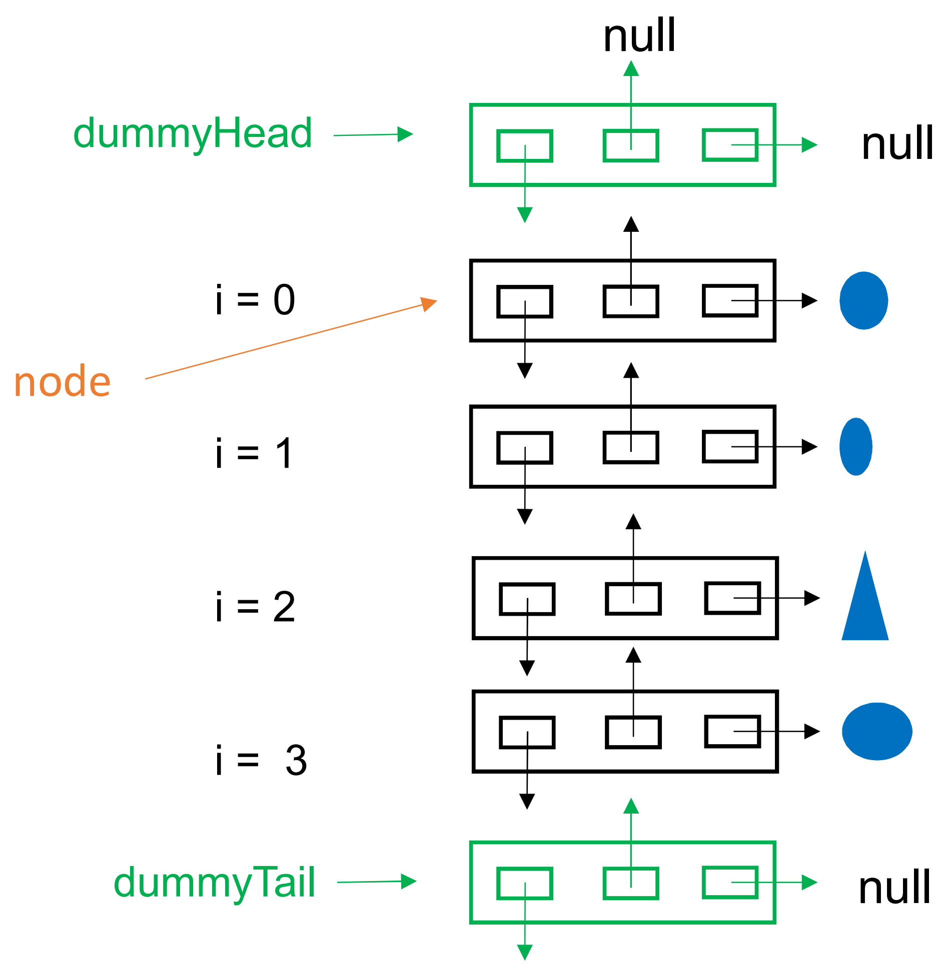
}



getNode(i) {   // returns a DNode

// Omitting verification that 0 <= i < size

```
node = dummyHead.next
 for (k = 0; k <  i ;   k ++)
      node = node.next
 return node
}
```



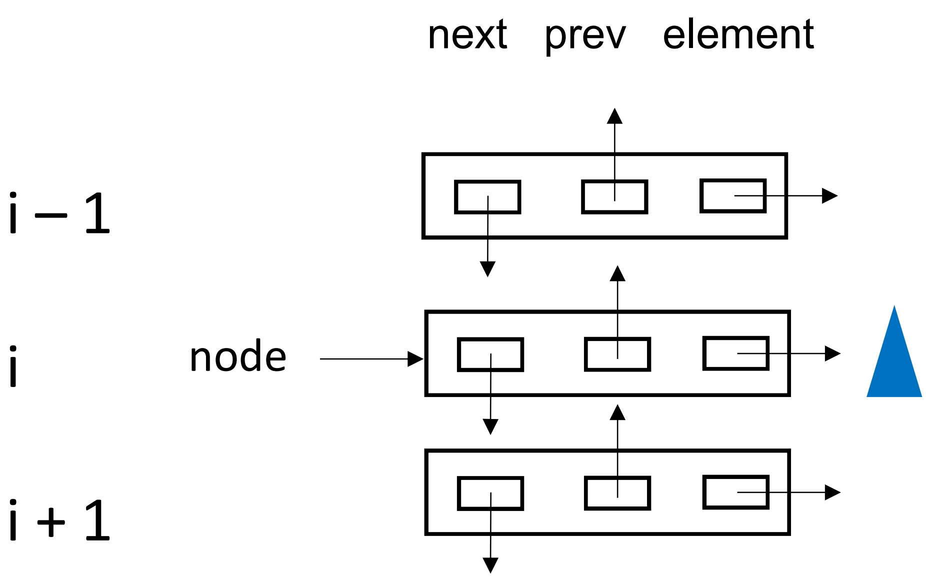
Ideas for how to speed this up?

Faster version of getNode()…

```
getNode( i ) {                            //  returns a DNode

    if ( i < size/2 ){                    //  iterate from head
        node = dummyHead.next
        for (k = 0; k < i;   k ++)
            node = node.next              //  exits loop when k==i
        }
    else{                                 //  iterate from  tail
        node = dummyTail.prev
        for ( k = size-1;   k >  i;   k -- )   //  exits loop when k==i
            node = node.prev
    }
    return node
}
```

```
remove( i ) {
    node = getNode( i )

}
```

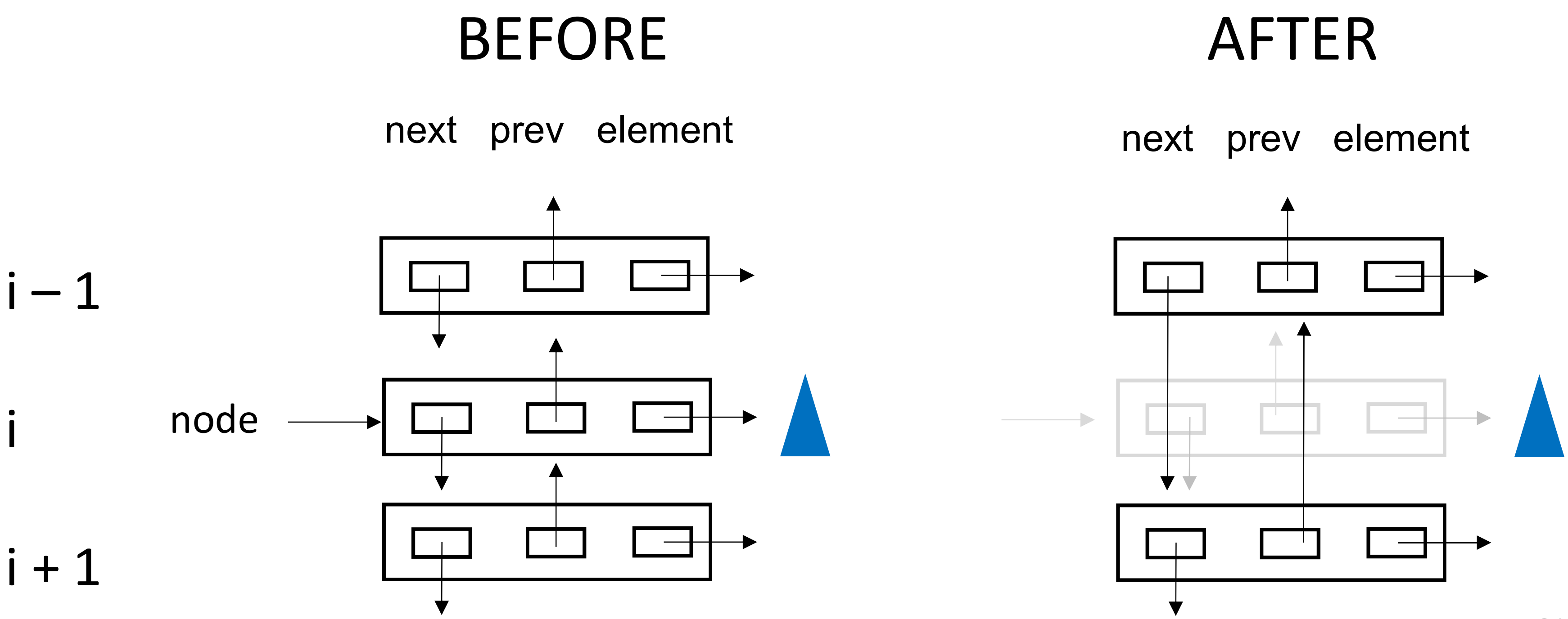
BEFORE



AFTER

?

```
remove( i ) {
    node  =  getNode( i )

            See exercises

}
```



Java LinkedList class

https://docs.oracle.com/javase/8/docs/api/java/util/LinkedList.html

It uses a *doubly linked list* as the underlying data structure.

It has some methods that ArrayList doesn't have e.g.

- addFirst()
- removeFirst()

- addLast()
- removeLast()

Why ?

Computational Complexity (N = list size)

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

Only if there is available space.
Worst case is O(N).

Best cases are O(1).
Worst cases are O(N).

Q: What is the time complexity of the following?

```
//    Assume E is some actual type
//    N is some constant

LinkedList< E > list = new LinkedList< E >();

for (k = 0; k < N;   k ++)
           list.addFirst(  new  E( …. )   );
```

A: $\mathbf{1} + \mathbf{1} + \mathbf{1} + \ \ldots . \ \mathbf{1} \ = \boldsymbol{N} \quad \Rightarrow \quad \boldsymbol{O}(\ \boldsymbol{N}\ )$

where ‘$\mathbf{1}$’ means constant time, i.e. do instructions 1 time

Q: What is the time complexity of the following ?

```
//  Let size == N

for (k = 0; k < list.size();   k++)
       list.get(  k  );
```

A: $$\mathbf{1+2+3+\cdots+N} = \frac{\boldsymbol{N(N+1)}}{\mathbf{2}} \Rightarrow \boldsymbol{O(\ N^2\ )}$$

Java 'enhanced for loop'

A more efficient way to iterate through elements in a Java `LinkedList` is to use:

```
for  (E    e :   list)  { …   }
```

'list' references a `LinkedList< E >` object.

`e` is a local variable to the loop. It is of type **`E`**, namely the type of element in the linked list.

You can use **`e`** and **`list`** within the loop, but don't modify **`list`**.

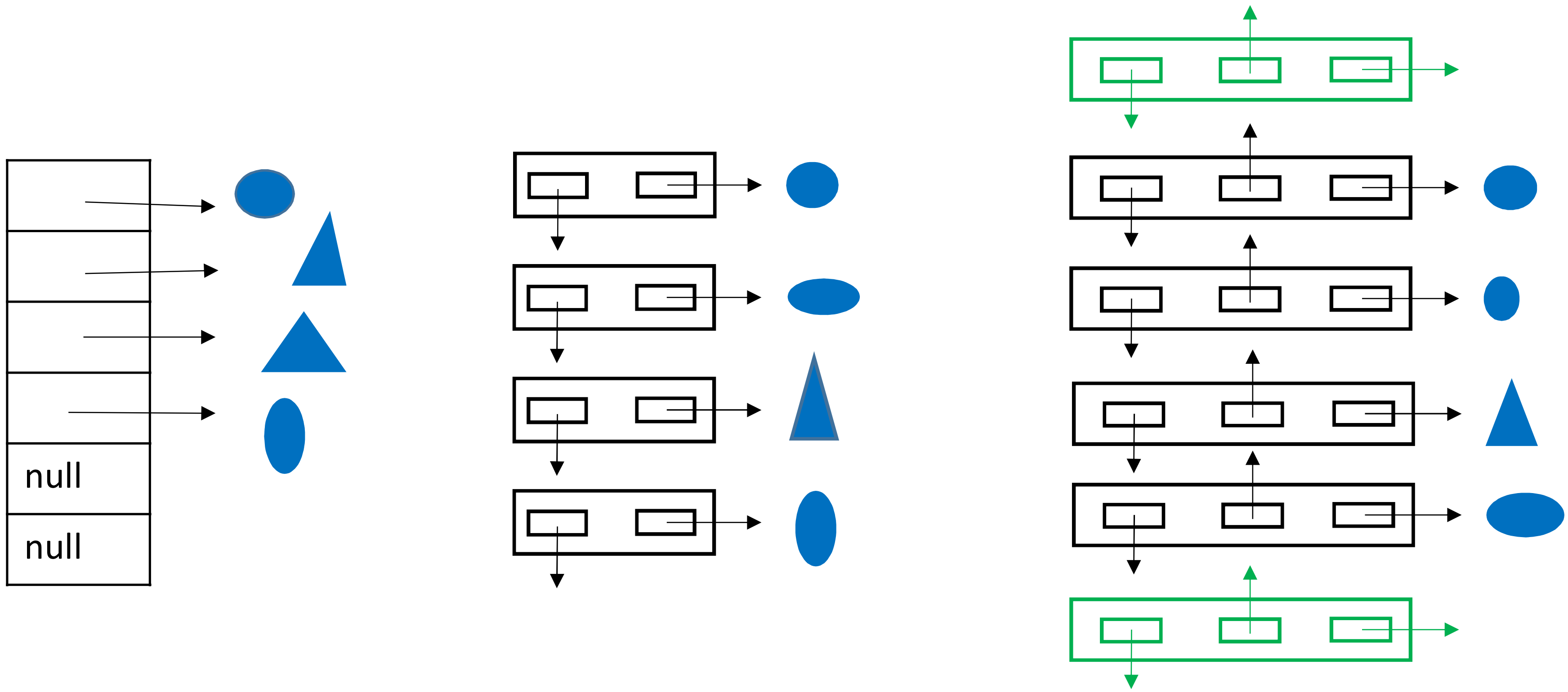
Java ‘enhanced for loop’

```
for  (E    e :   list)  {
        // do something
}
```

When E is a `LinkedList`, this is implemented roughly as

```
node = head     // or write it using the dummyhead idea
while (node != null){
        e = node.element
        //  do something with e
        node = node.next
}
```

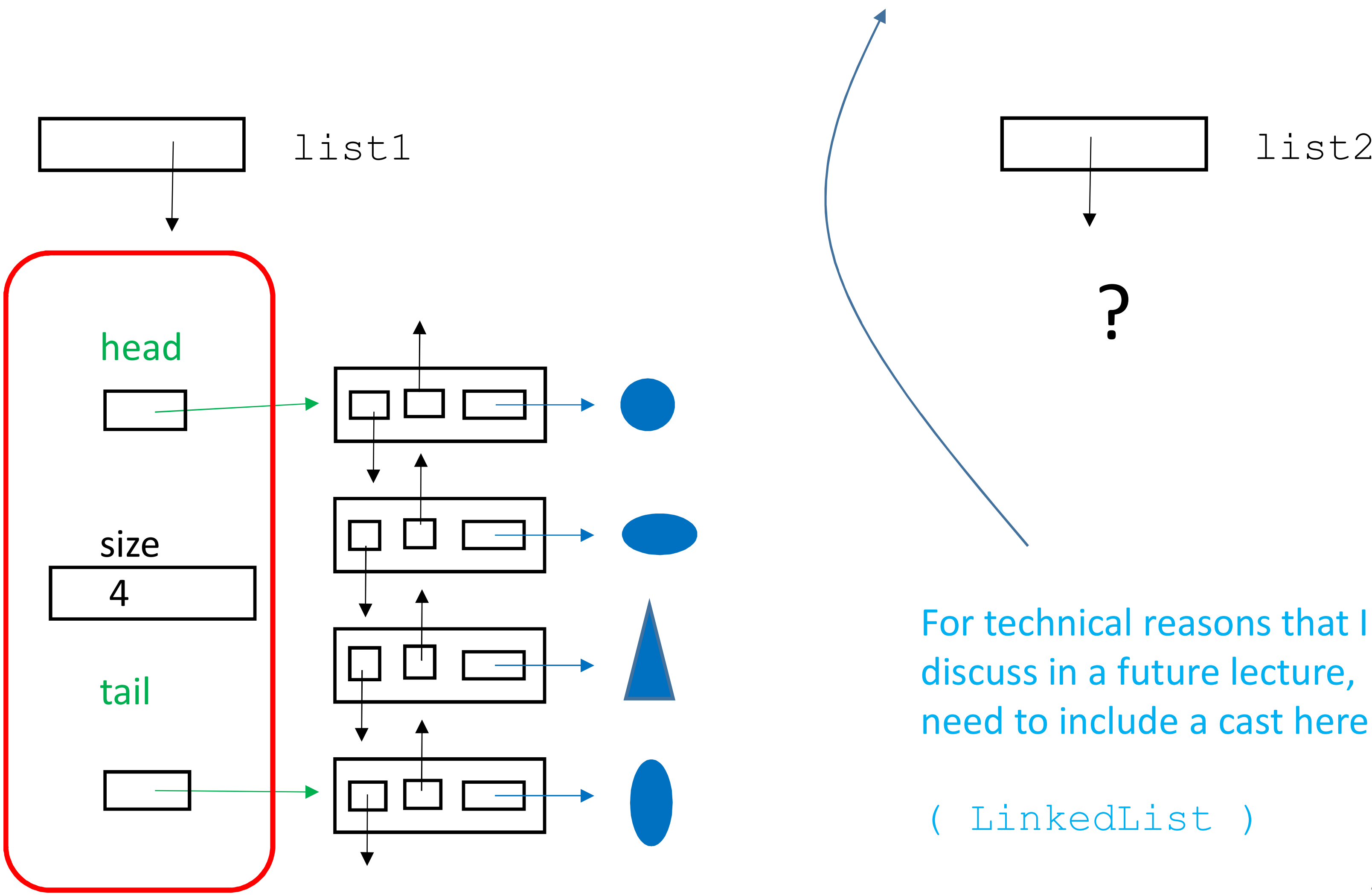
What about "Space Complexity" ?



We say all three data structures use space O(N) for a list of size N. But linked lists use more than 2x (single) or 3x (double) as much space as arraylists.

How to “clone” a list i.e. make a copy?

```
LinkedList<Shape>  list2 =  list1.clone();
```

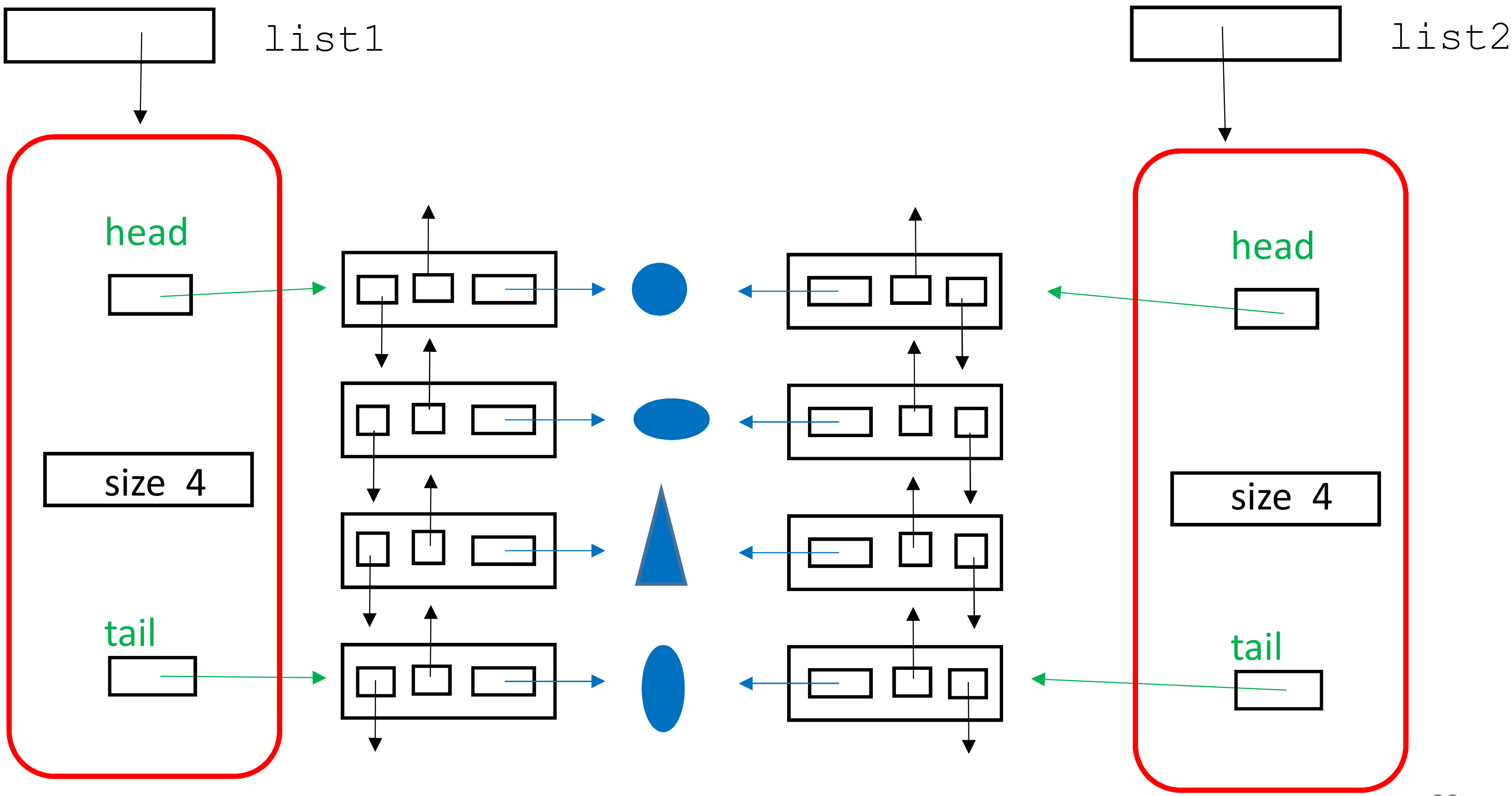


For technical reasons that I will discuss in a future lecture, you need to include a cast here:

```
( LinkedList )
```

“Shallow copy”

The list object and the list nodes *are* copied.
But the Shape objects *are not* copied.



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

LinkedList<T>.clone() makes a shallow copy.

clone

public Object clone()

Returns a shallow copy of this LinkedList. (The elements themselves are not cloned.)

Overrides:

clone in class Object

Returns:

a shallow copy of this LinkedList instance https://docs.oracle.com/javase/7/docs/api/java/util/LinkedList.html

Next week you will understand why this says Object rather than LinkedList. This is the reason that we need to cast, as I mentioned two slides ago.

clone

```
public Object clone()
```

Returns a shallow copy of this `LinkedList`. (The elements themselves are not cloned.)

Overrides:

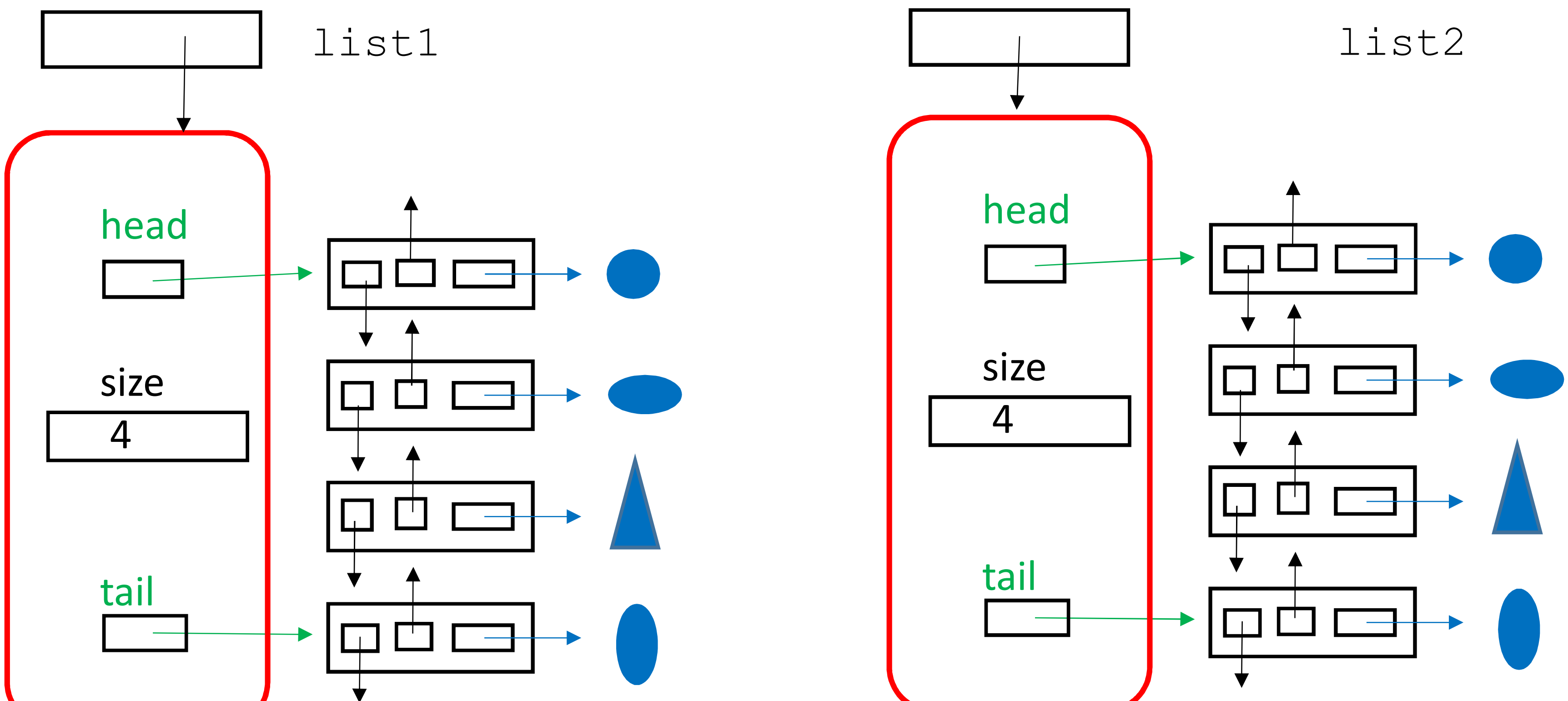
`clone` in class `Object`

Returns:

a shallow copy of this `LinkedList` instance

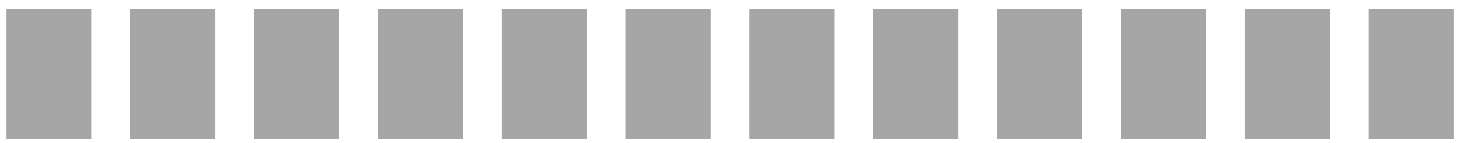
“Deep copy”

The linkedlist object, the list nodes, and the list elements are all copied. The Java `LinkedList` class does *not* have a built-in method to make a deep copy.

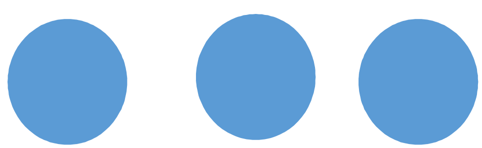


Real Example – Shallow Copy

Suppose have a list of midterm exams for a course.   The exams need to be graded by hand.



Each grader (TA) is responsible for grading certain questions.   So each grader will have a list of exams, and will write on each of exams.

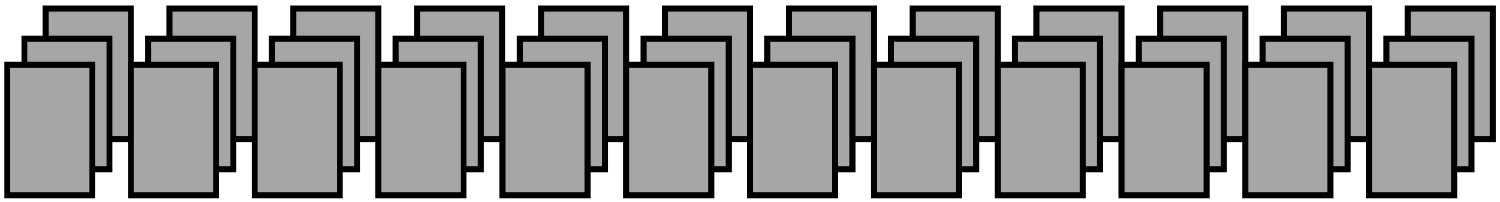


Each grader needs a *shallow* copy of the list of exams.

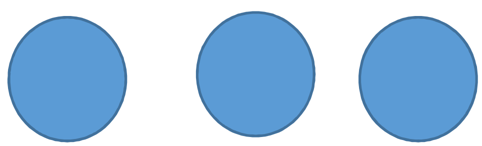
For this example, we don't care if it is a linked list or array list.

Real Example – Deep Copy

Suppose have a list of job applications, which will examined by different people in a company. Suppose the employer wants independent assessment of applications by different people.



Each person assessing the applications will mark up the PDF of each application.



Each assessor needs a *deep* copy of the list of applications. They should not be allowed to see each other's assessments.

Coming up…

Lectures

Wed. Feb. 2

Quadratic Sorting i.e. $O(N^2)$

- bubble sort
- selection sort
- insertion sort

Fri. Feb. 4

Object Oriented Design 1

(Inheritance)

Assessments

Assignment 1

- due on Friday, Feb. 11