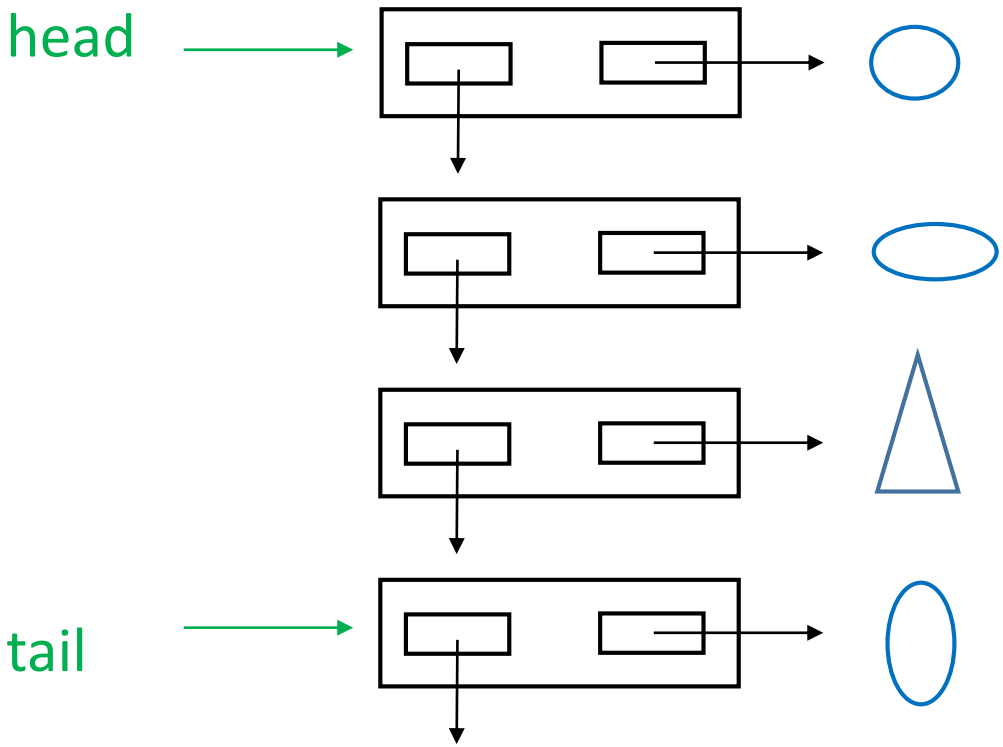
COMP 250

Lecture 11

doubly linked lists

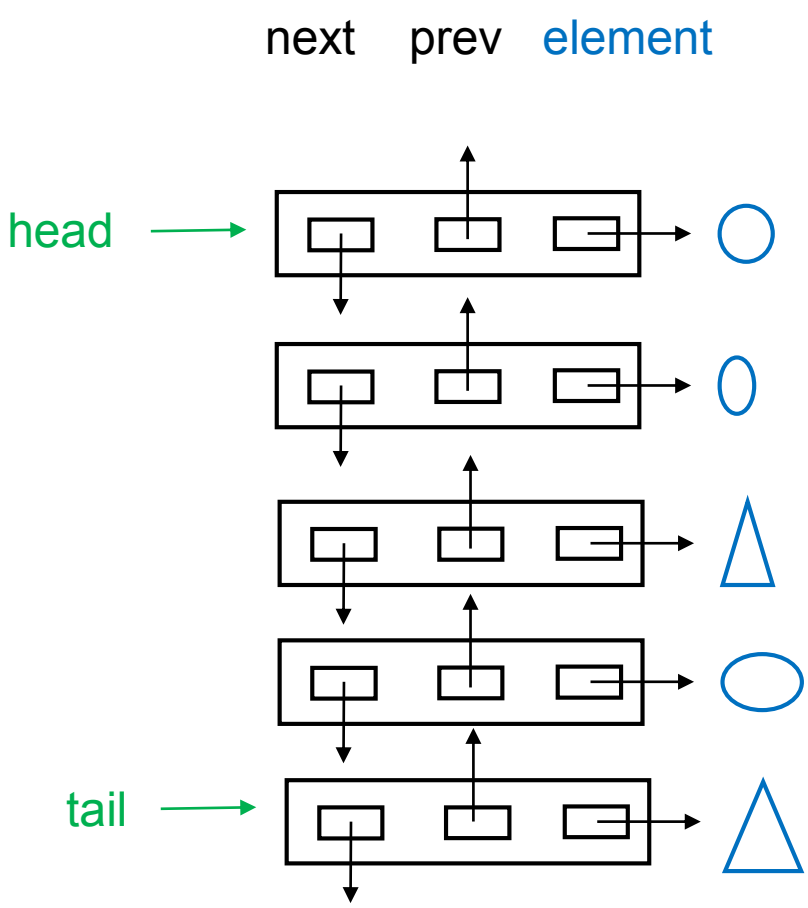
Oct. 3, 2018

Singly linked list

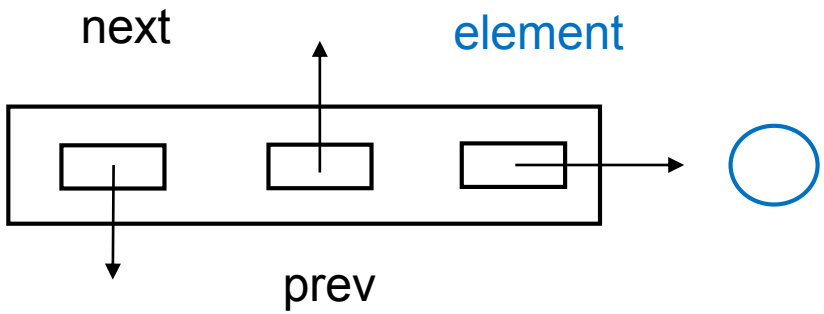


Doubly linked list

Each node has a reference to the next node and to the previous node.



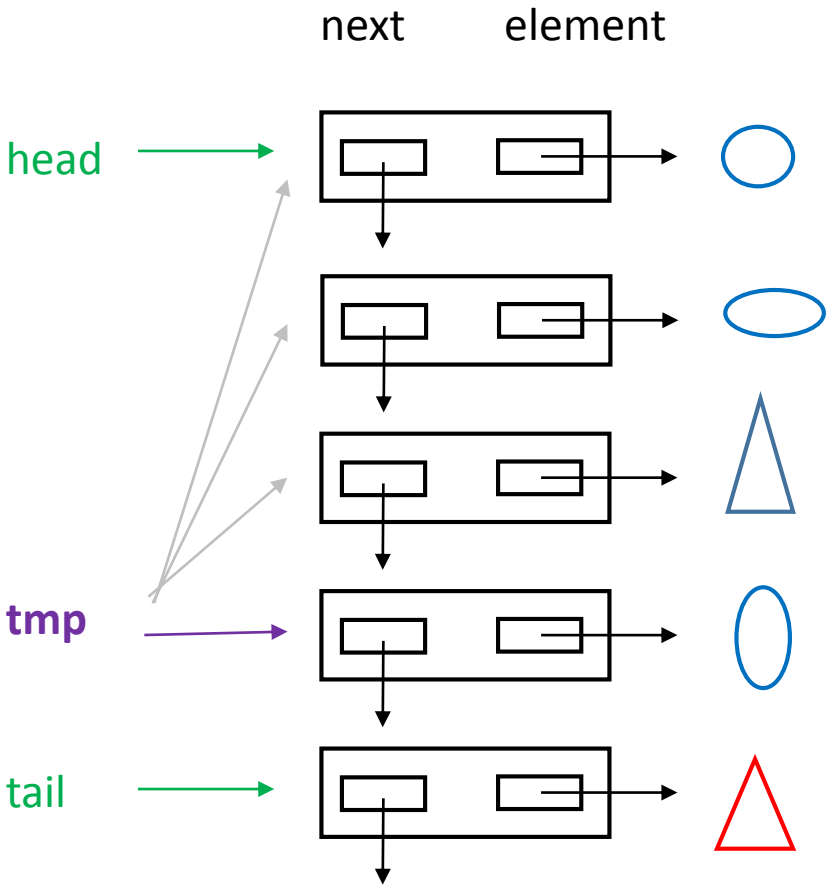
```
class DNode< E > {

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

   // constructor

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

}
```



Motivation for doubly linked lists:

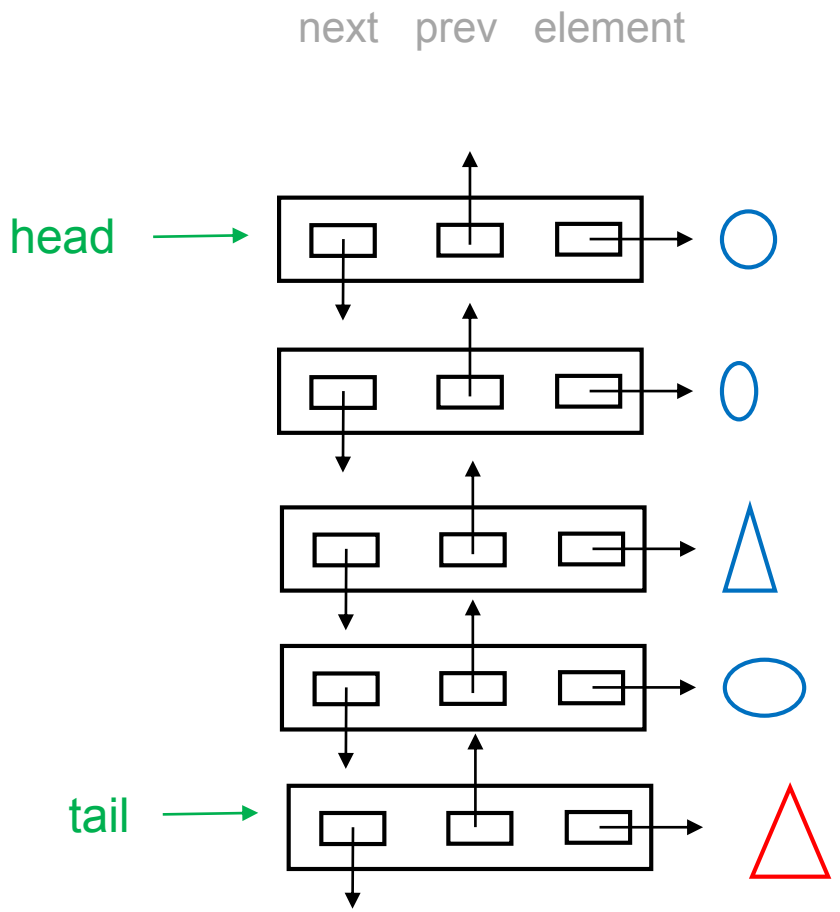
recall removeLast ( ) for singly linked lists

The only way to access the element before the tail was to loop through all elements from the head.



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

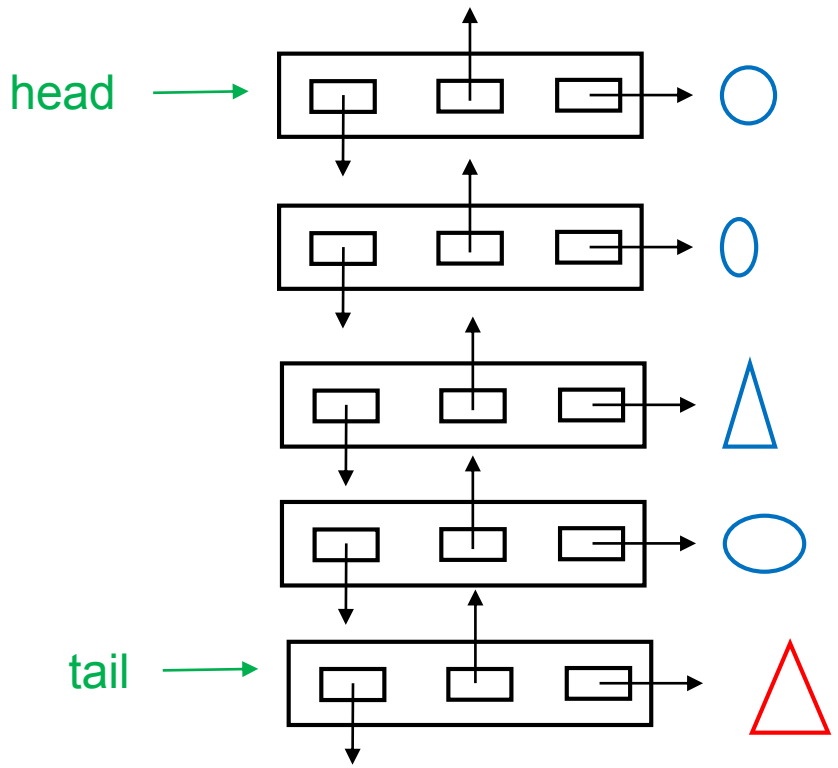
```
removeLast(){

        ?

    size = size – 1

}
```



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

```
removeLast(){

        tail        = tail.prev

        tail.next = null
        size = size – 1

}
```

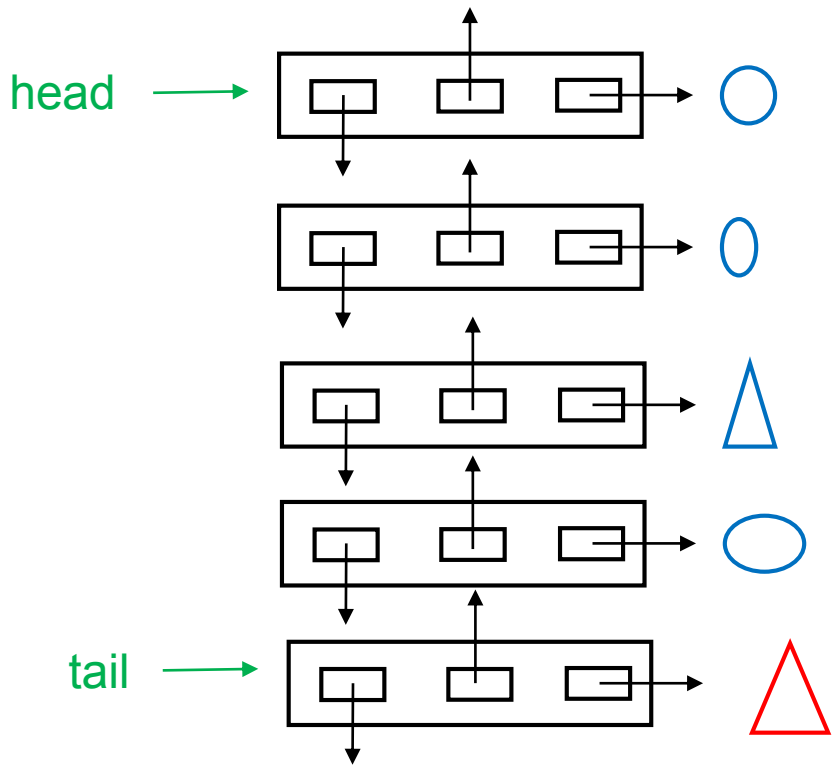
BEFORE



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

BEFORE

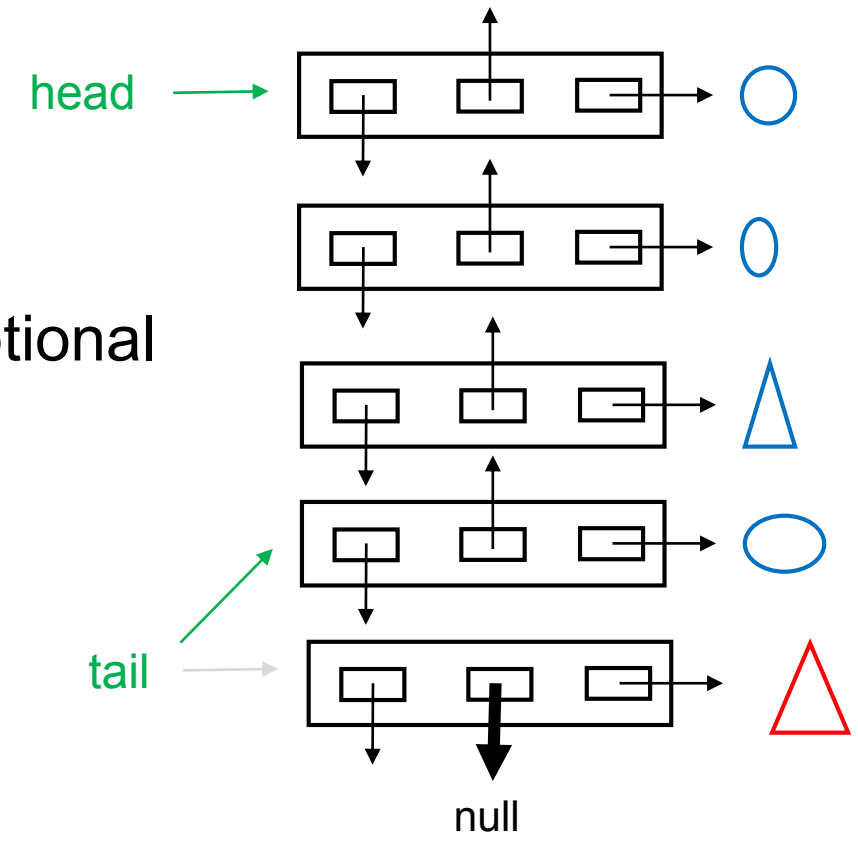
```
removeLast(){
        e       = tail.element
        tail    = tail.prev

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



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

AFTER

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



Time 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)**

Other List Operations

:

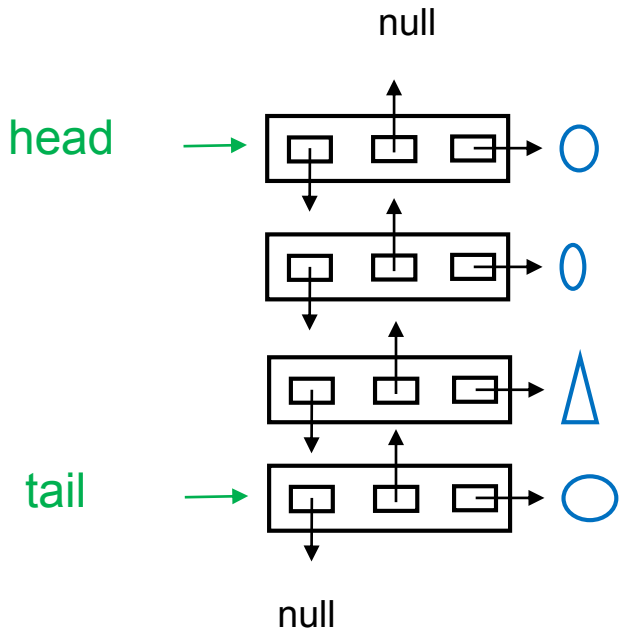
get(i)

set(i,e)

add(i,e)

remove(i)

:



Many list operations require access to node i.

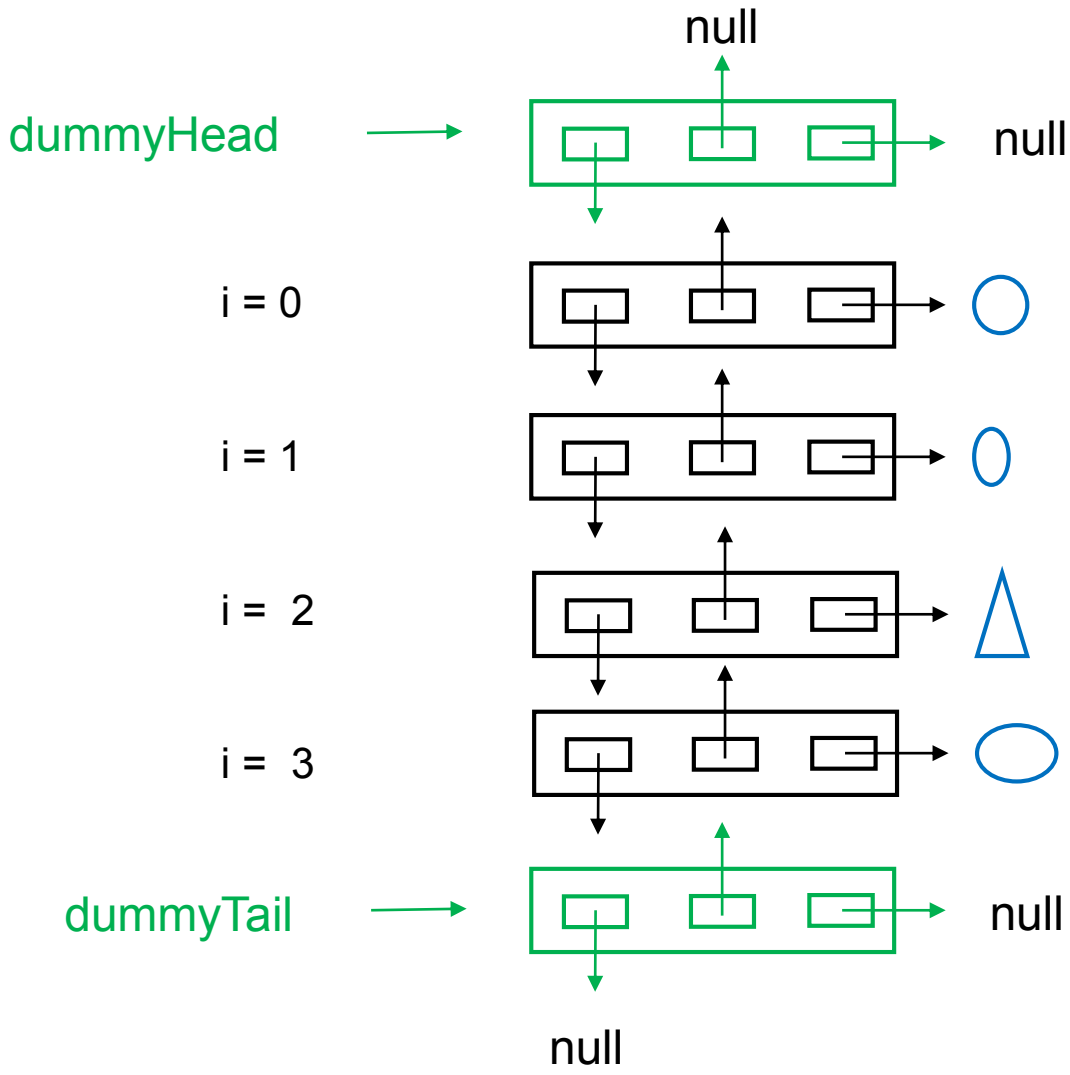
Suppose we want to access node i in a doubly linked list.

One issue is that edge cases (i = 0, i = size – 1) require special treatment in many methods, which can lead to coding errors.

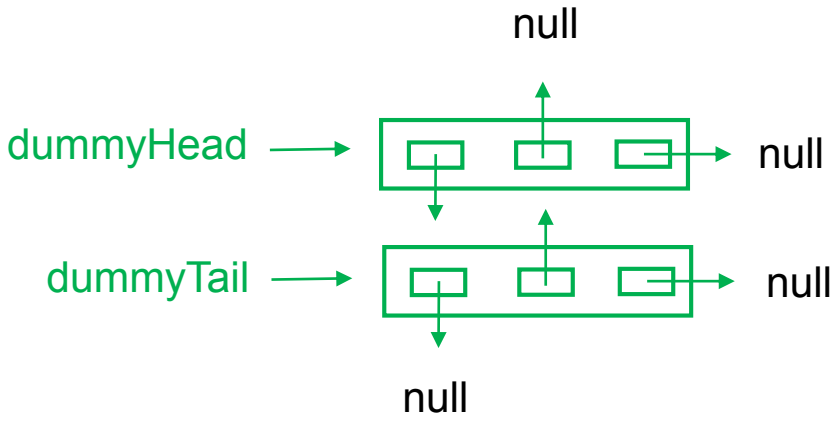
Node 0 has a null prev field. Node size-1 has null next field.

(The same issue comes up in singly linked lists, but we ignore it.)

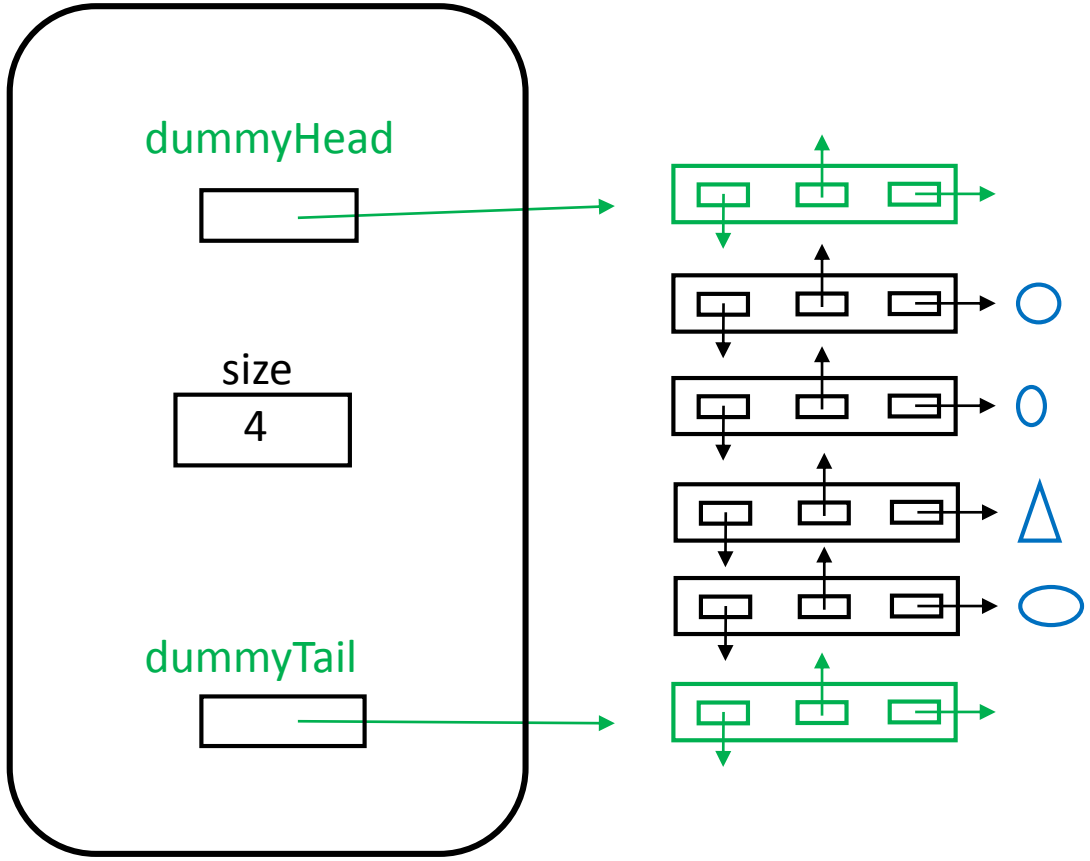
Avoid edge cases with “dummy nodes”



```
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>{ … }
}
```



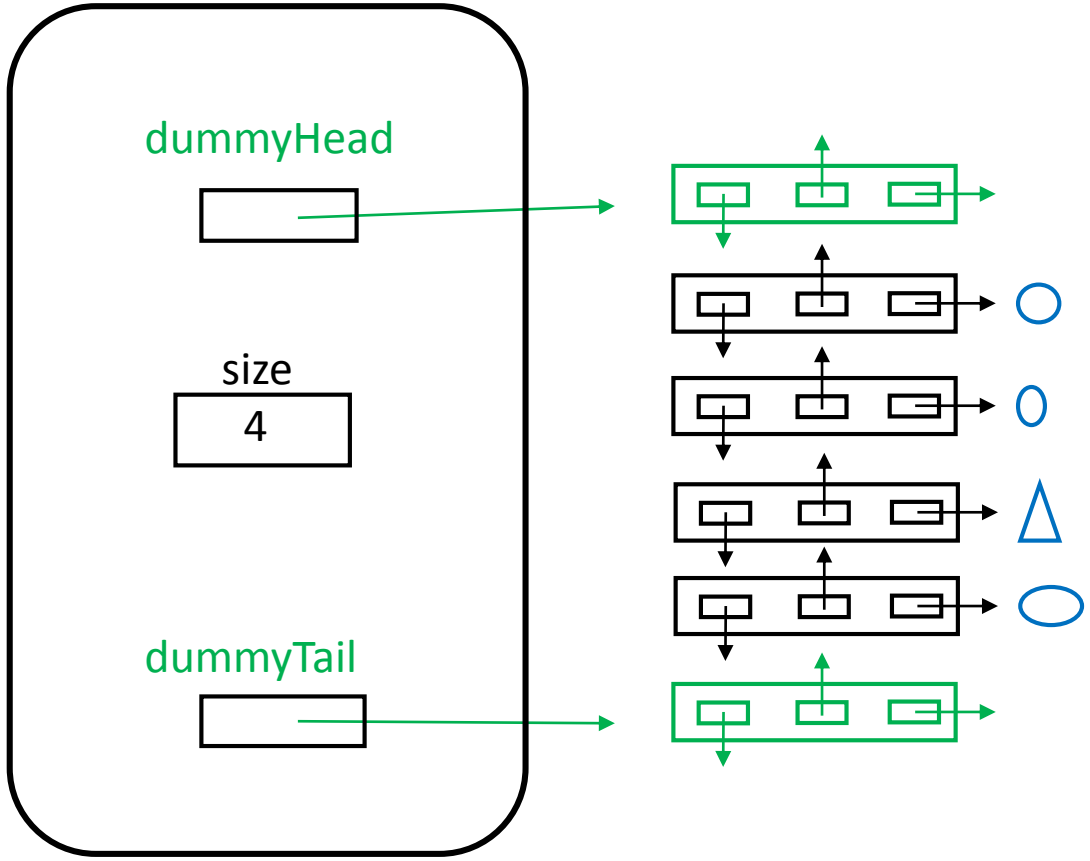
DLinkedList< Shape >
object



Q: How many objects in total in this figure?

A:

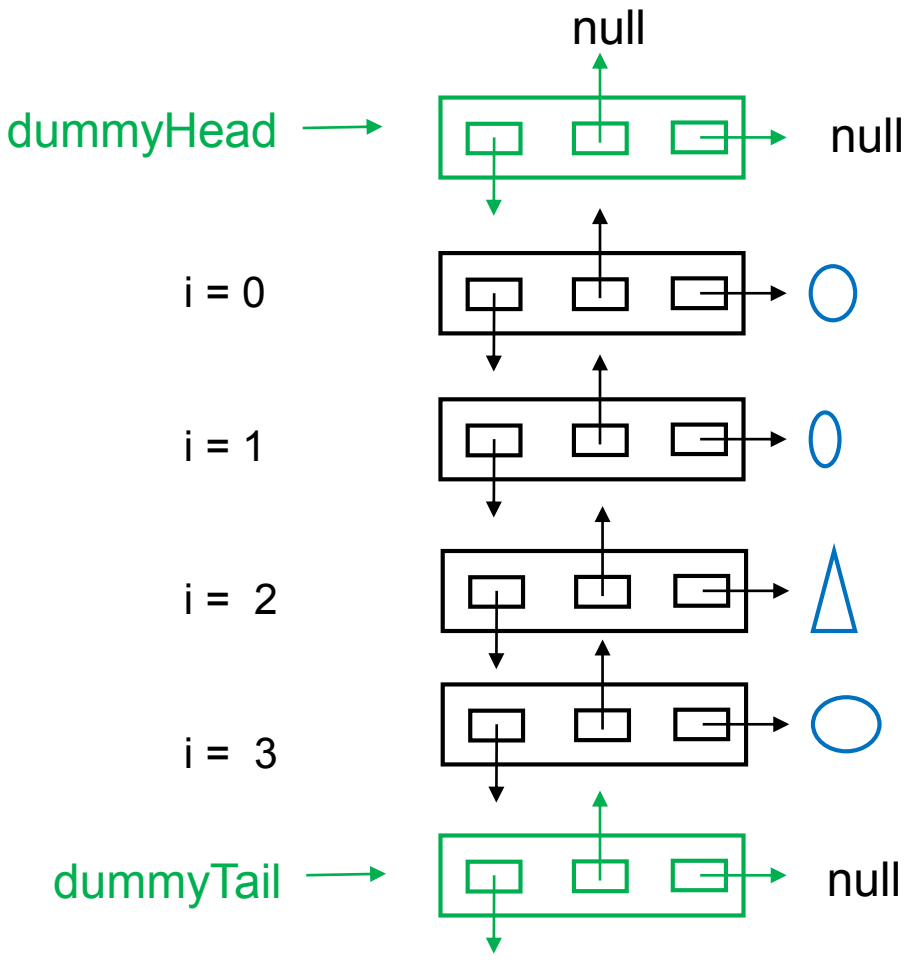
DLinkedList< Shape >
object



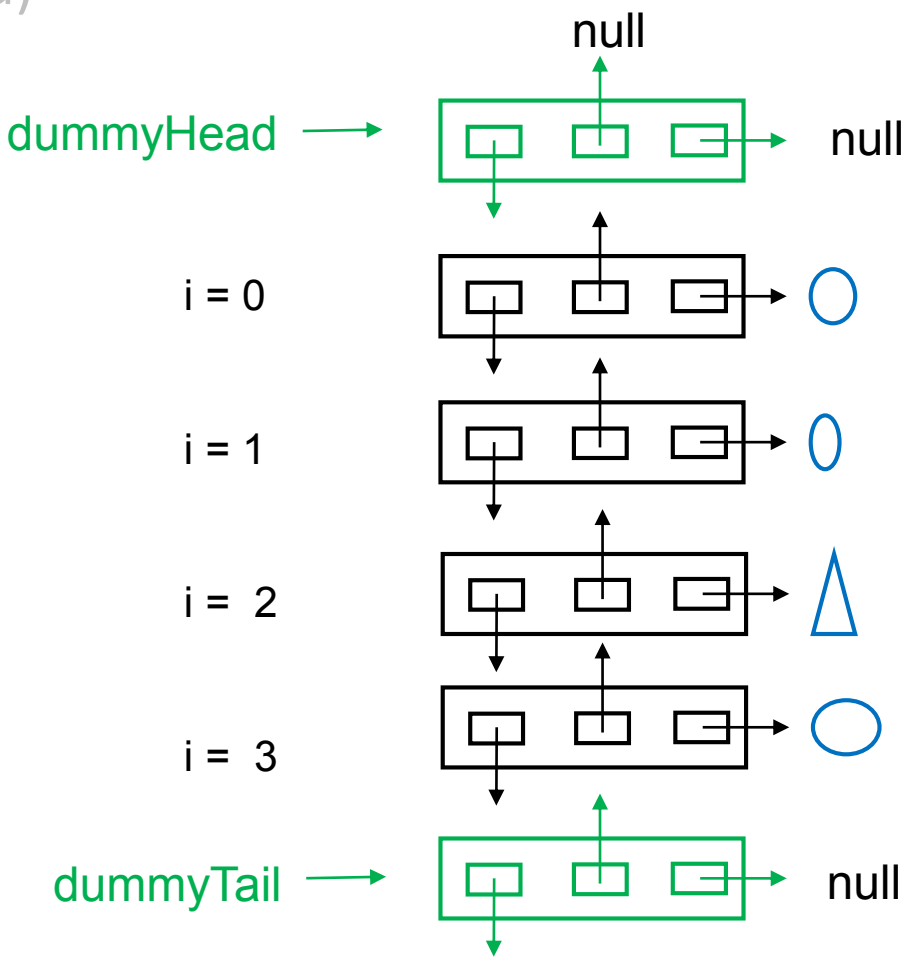
Q: How many objects in total in this figure?

A: 1 + 6 + 4 = 11

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

    node = getNode(i);    // getNode() to be discussed next slide
    return node.element;

}
```



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

//    verify that 0 <= i < size   (omitted)

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



More efficient getNode()… half the time

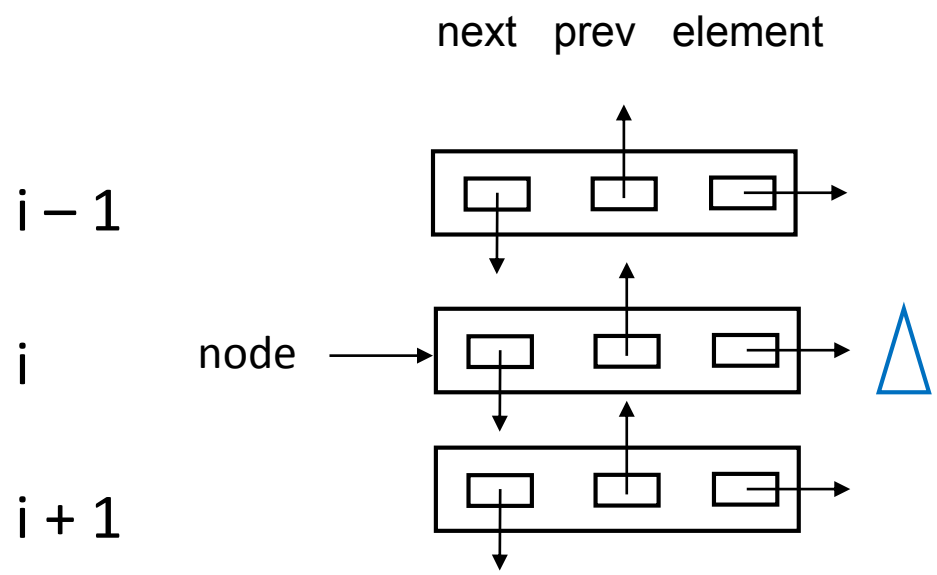
```
getNode( i ) {                          //  returns a DNode

     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 -- )
               node = node.prev          //  exits loop when k==i
           }
      return node
}
```

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

}
```

Exercise (see online code)

BEFORE

AFTER

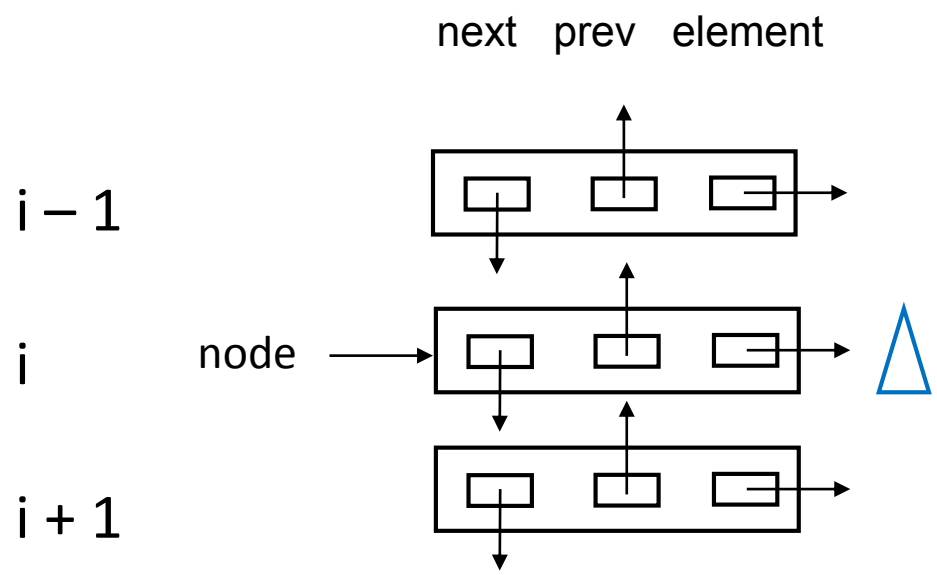


```
remove( i ) {
        node  =  getNode( i )
```

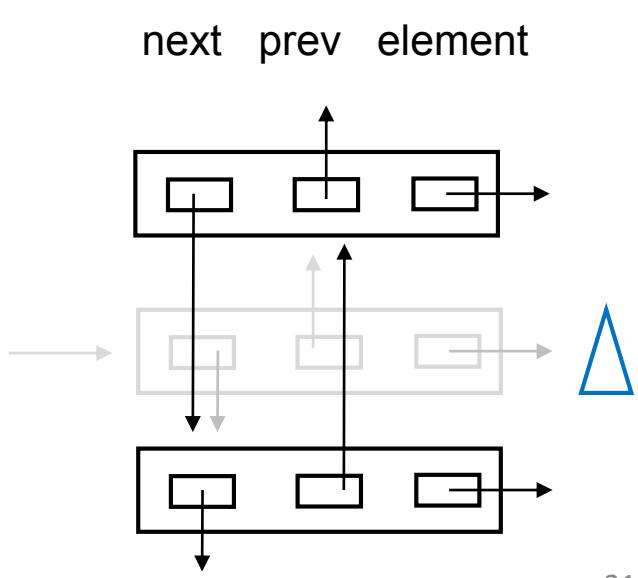
Exercise (see online code)

```
}
```

BEFORE



AFTER



Time 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)**
remove(i)	?	?	?

Time Complexity: *Worst Case*
(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)
remove(i)	**O(N)**	**O(N)**	**O(N)**

O( ) ignores constant factors !

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 ?

Q: What is the time complexity of the following ?

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

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

Q: What is the time complexity of the following ?

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

for (k = 0; k < N;   k ++)                    //  N is some constant
        list.addFirst(  new  E( …. )   );     // or addLast(..)
```

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

where ‘1′ means constant.

Q: What is the time complexity of the following ?

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

Assume here that getNode(i) always starts at the head.

Q: What is the time complexity of the following ?

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

Assume here that getNode(i) always starts at the head.

A: $\mathbf{1 + 2 + 3 + \ldots . N}$

Q: What is the time complexity of the following ?

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

Assume here that getNode(i) always starts at the head.

A: $\mathbf{1 + 2 + 3 + \ldots . N}$

$$= \frac{N\,(N+1)}{2} \quad \Rightarrow \quad O(N^2)$$

Q: What is the time complexity of the following ?

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

More generally for a doubly linked list….

A: $\mathbf{1 + 2 + 3 + \ldots . \frac{N}{2}} \quad + \quad \mathbf{1 + 2 + 3 + \ldots . \frac{N}{2}}$

$$= \frac{N}{2}\left(\frac{N}{2} + 1\right) \Rightarrow \quad O(N^2)$$

(see Exercises for linked lists)

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 the LinkedList< E > object. It needs to be already defined in your program.

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

Java ‘enhanced for loop’

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

is implemented roughly as

```
node = head
while (node != null){
      //   do something
           node = node.next
}
```

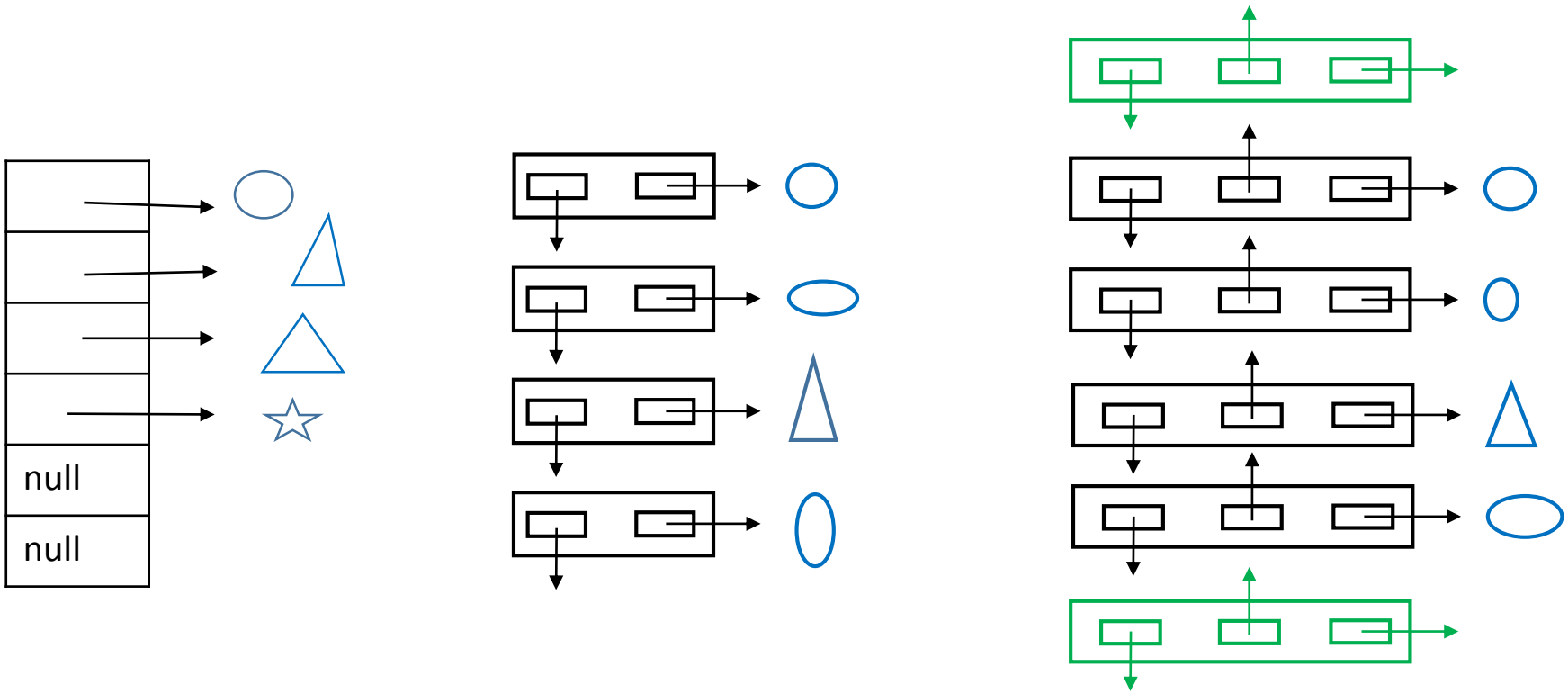
Heads up (iterators)

Java allows you to define “iterators” using the Iterable interface.

You will learn what this means in a few weeks.

You will see an iterator in the linked list class in Assignment 2, but you won’t need to use it.

What about “Space Complexity” ?



All three data structures use space O(N) for a list of size N.
But linked lists use 2x (single) or 3x (double).

Announcements

- A1 is due on Monday Oct 8 at midnight

- Eclipse and IntelliJ Tutorials still going on
  - Wednesday, Oct 3, 4:30-6 (IntelliJ)
  - Thursday Oct 4, 4:30-6 (Eclipse)
  - Friday Oct 5, 4-5:30 (IntelliJ)

  See TA's in office hours if you need help.