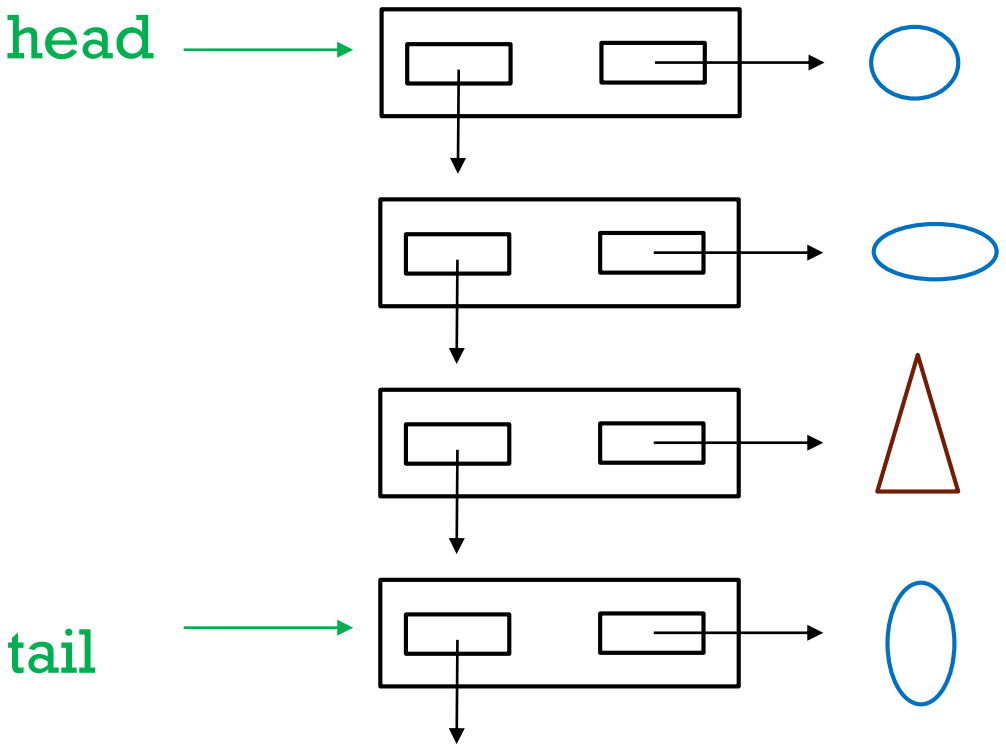
COMP 250
INTRODUCTION TO COMPUTER SCIENCE

Lecture 14 – Lists II

Roman Sarrazin-Gendron, Winter 2020

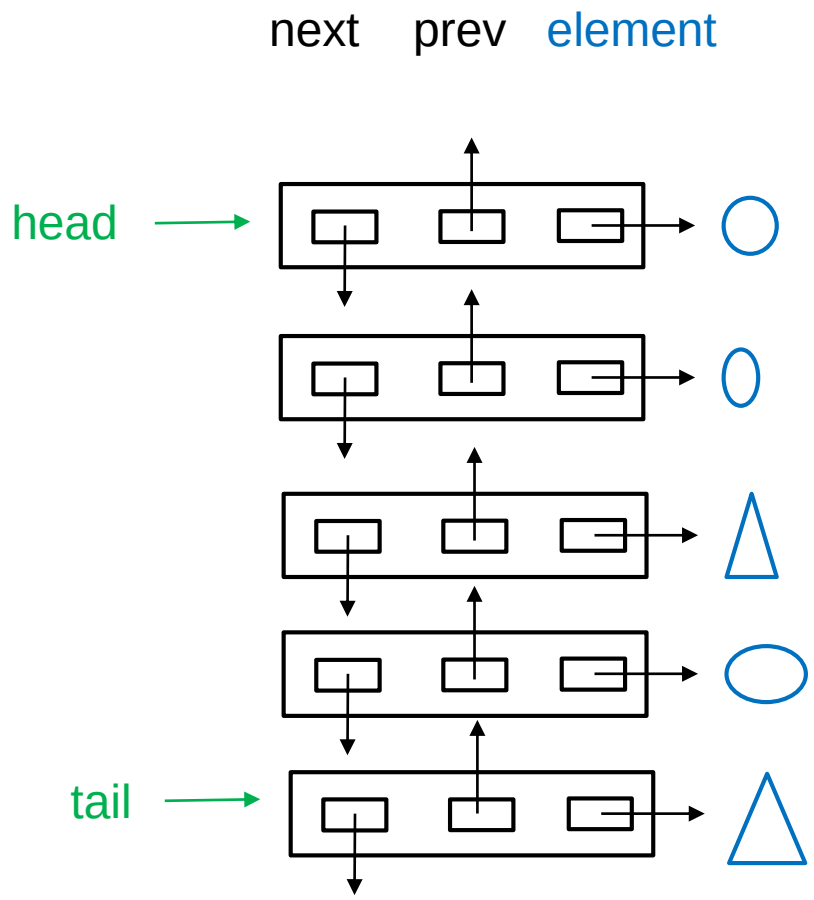
Slides very much based on Michael Langer and Giulia Alberini

LAST LECTURE : SINGLY LINKED LIST



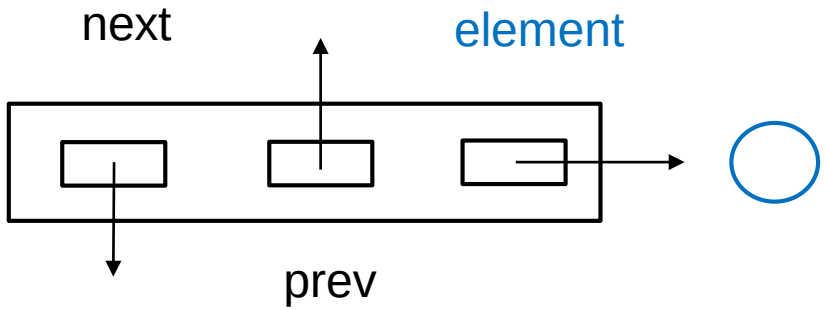
TODAY : DOUBLY LINKED LIST

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



JAVA CLASS FOR DOUBLY LINKED LIST

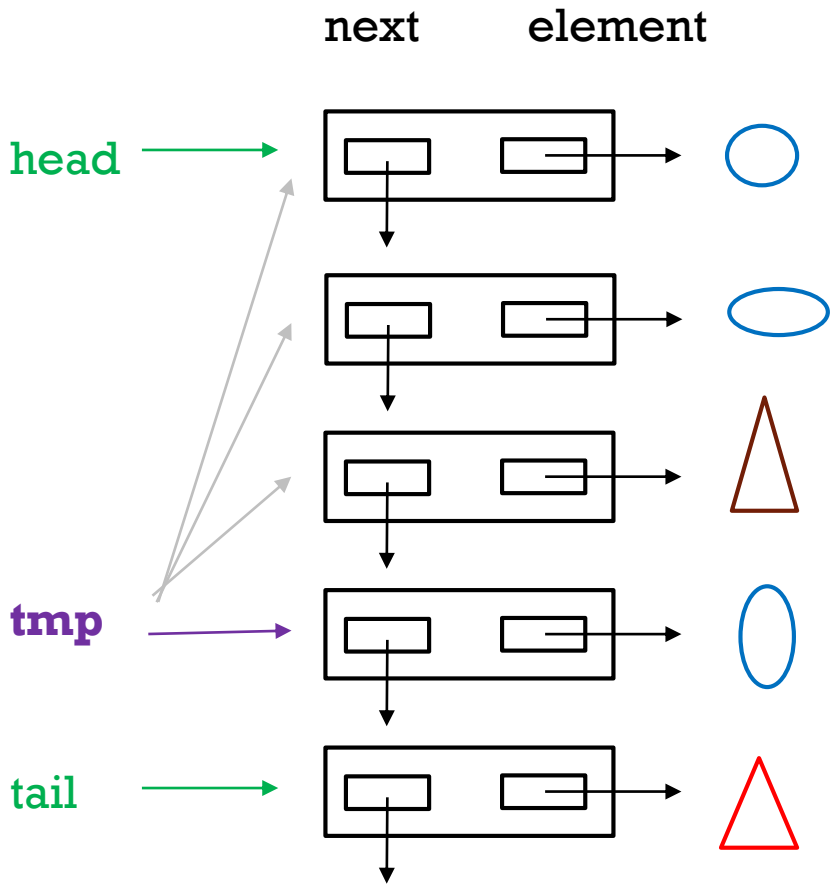
```
class DNode< E > {

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

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



MOTIVATION FOR A DOUBLY LINKED LIST

Recall removeLast ( ) for singly linked lists.

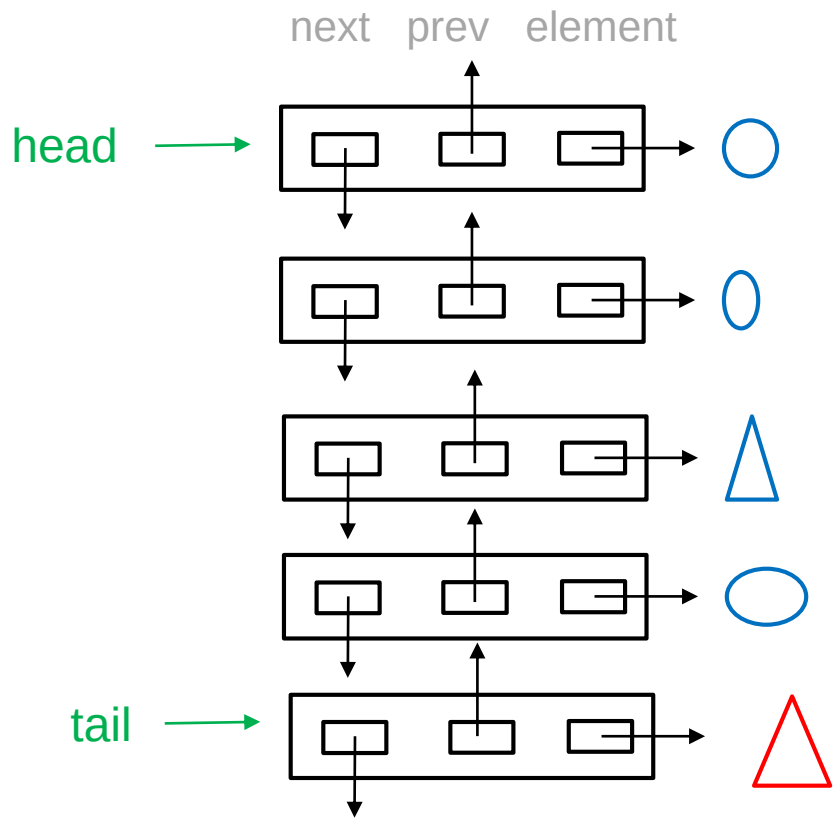
The only way to access the element before the tail was to loop through all elements from the head.



REMOVING THE LAST ELEMENT OF A DOUBLY LINKED LIST

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

```
removeLast(){

        tail        = tail.prev
        tail.next.prev = null
        tail.next = null
        size = size – 1
                :
}
```



TIME COMPLEXITY COMPARISON

	array list	**SLinkedList**	**DLinkedList**
addFirst	size	constant	constant
removeFirst	size	constant	constant
addLast	constant	constant	constant
removeLast	constant	**size**	**constant**

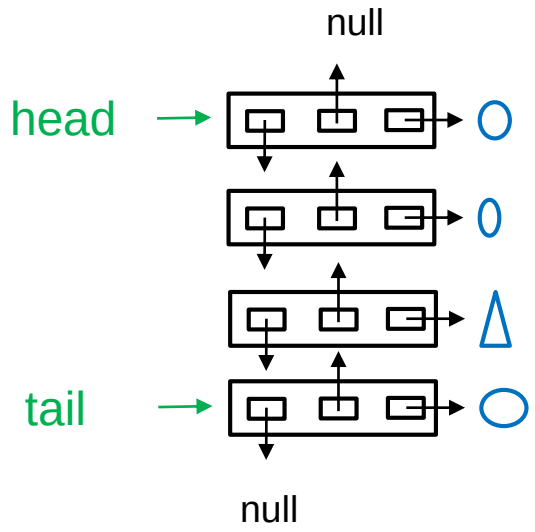
OTHER LIST OPERATIONS

get(i)

set(i,e)

add(i,e)

remove(i)



Many list operations require access to node i.

POTENTIAL PROBLEMS

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

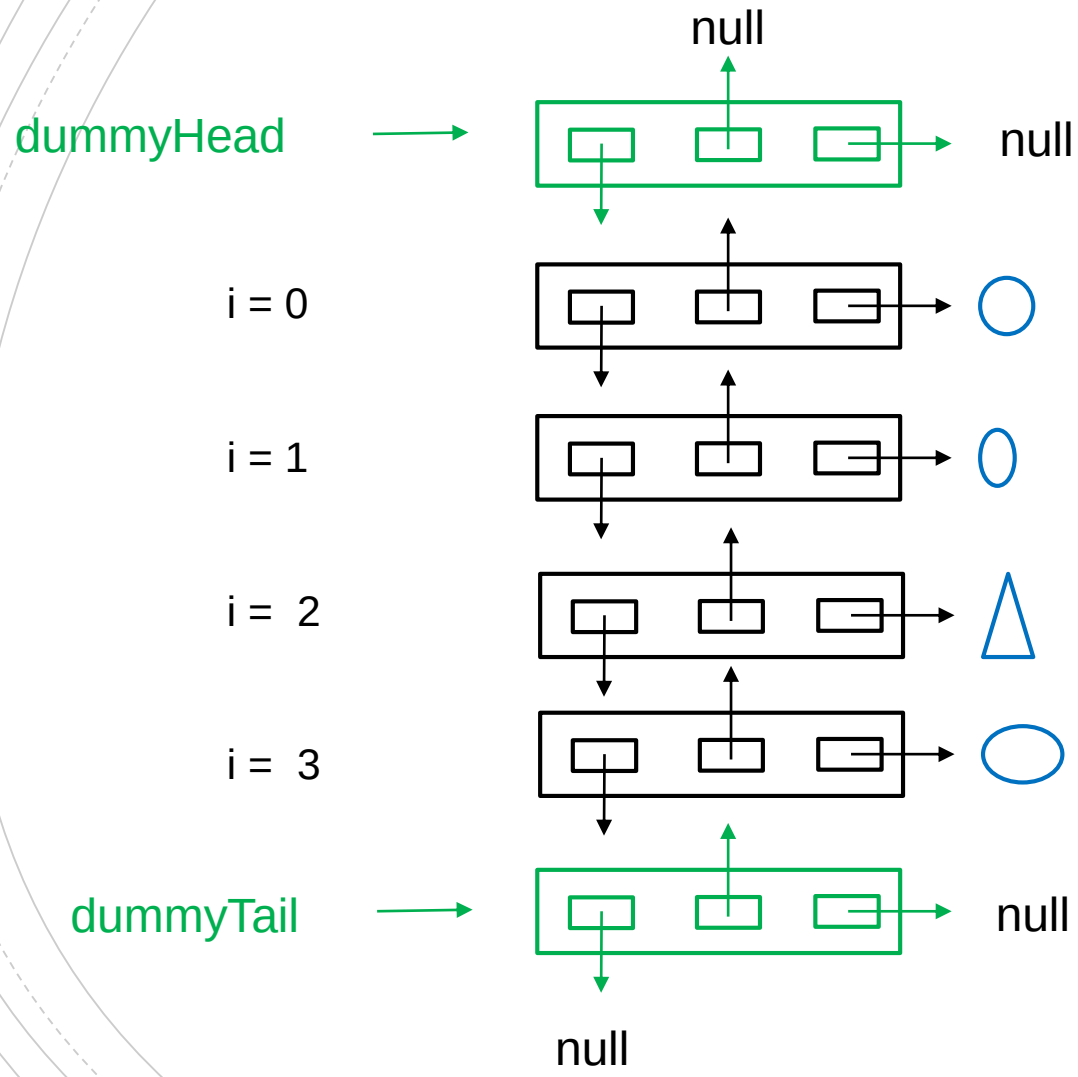
Two issues arise:

Edge cases (i = 0, i = size – 1) require extra code.

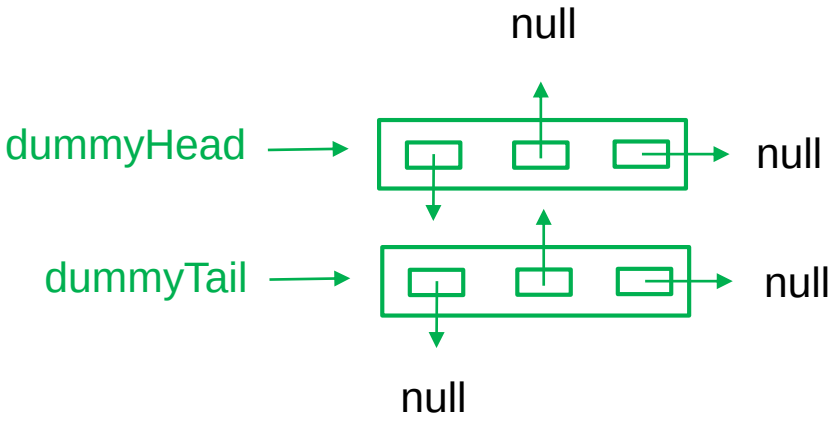
This is a pain and can lead to coding errors.

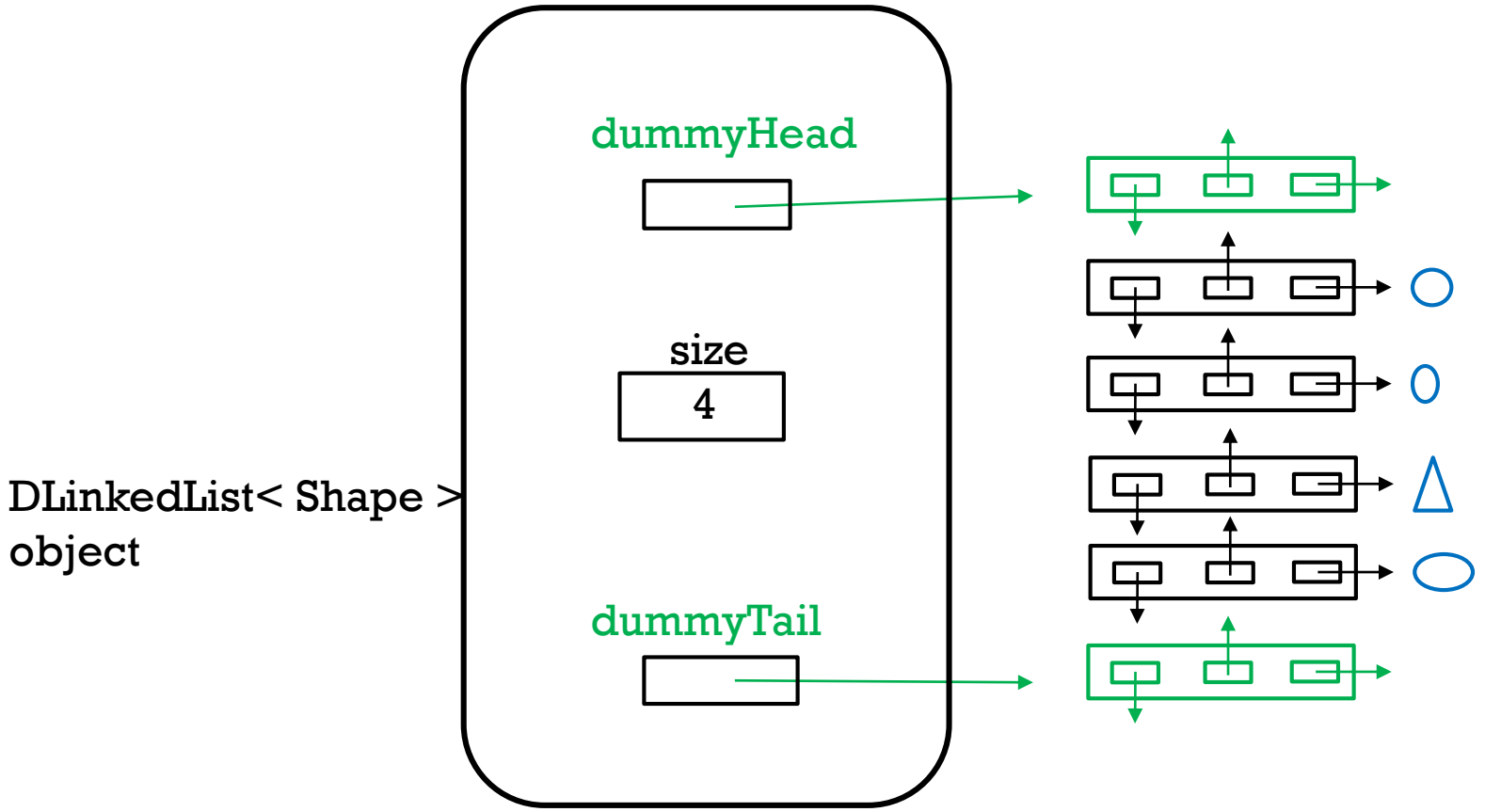
How long does it take to access node i ?

AVOID EDGE CASES WITH "DUMMY NODES"



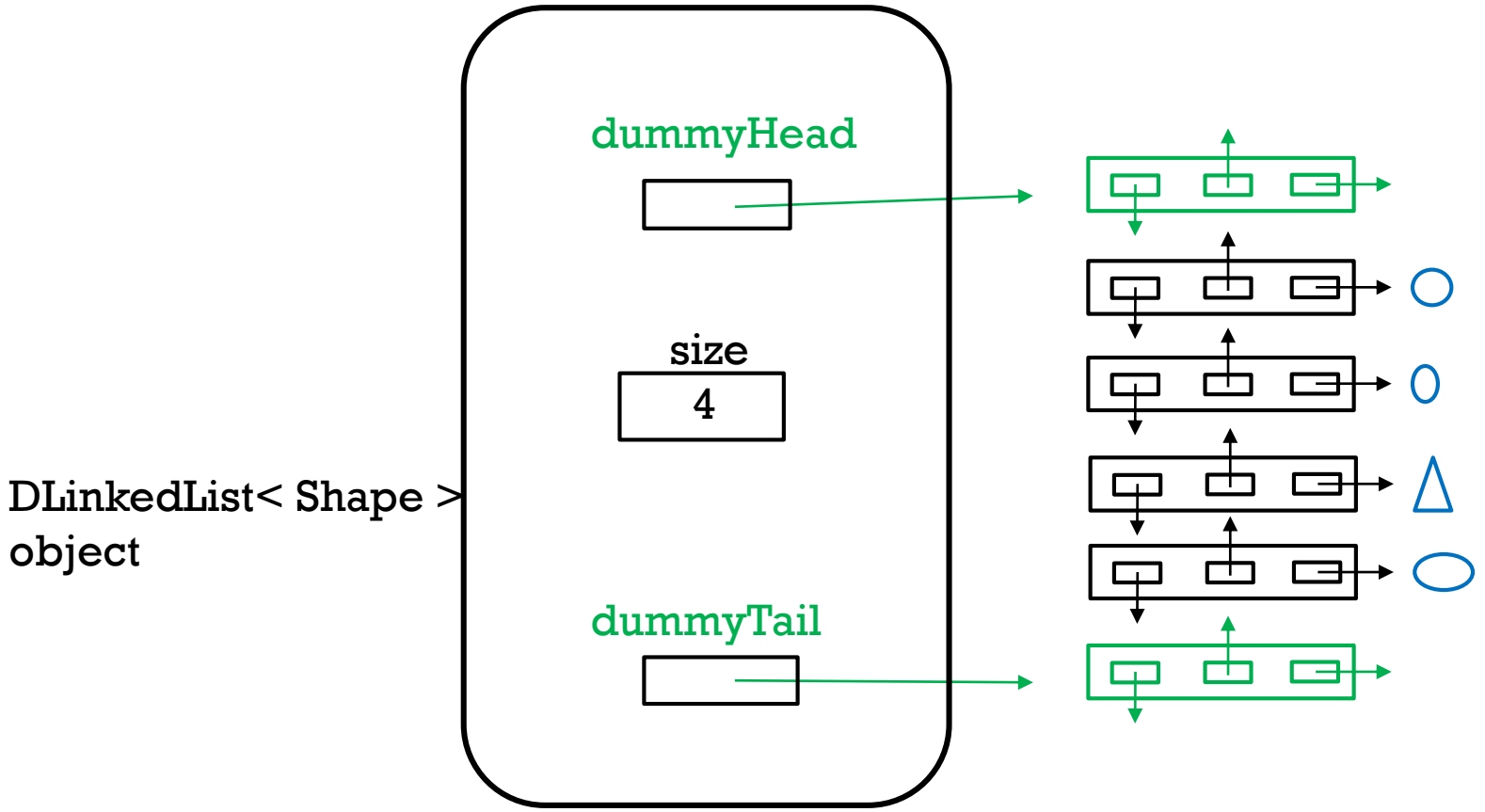
```
class  DLinkedList<E>{   //  Java code

    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>{ … }
}
```





Q: How many objects in total in this figure?

A:



Q: How many objects in total in this figure?

A: 1 + 6 + 4 = 11

EDGE CASES ARE NOW LESS OF A PROBLEM

We wanted to access general node i in a linked list.

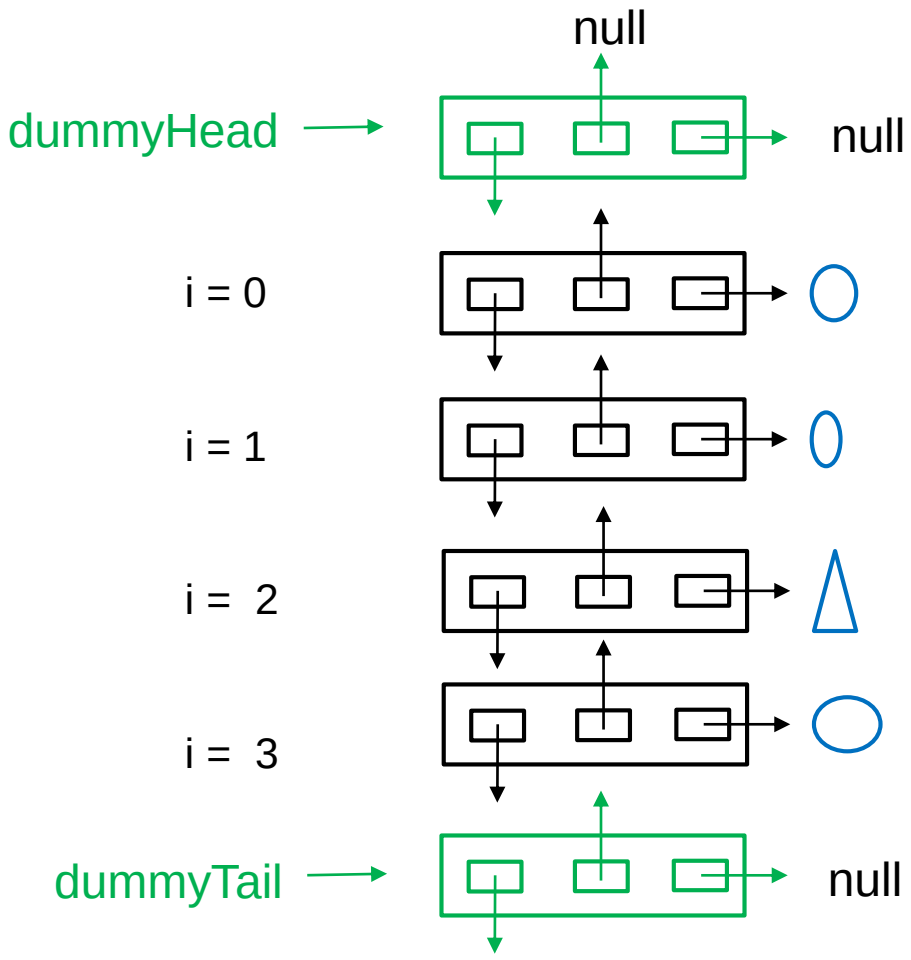
Two issues arise:

~~Edge cases (i = 0, i = size – 1) require extra code.~~
~~*This is a pain and can lead to coding errors*.~~

How long does it take to access node i ?

GET ELEMENT AT INDEX

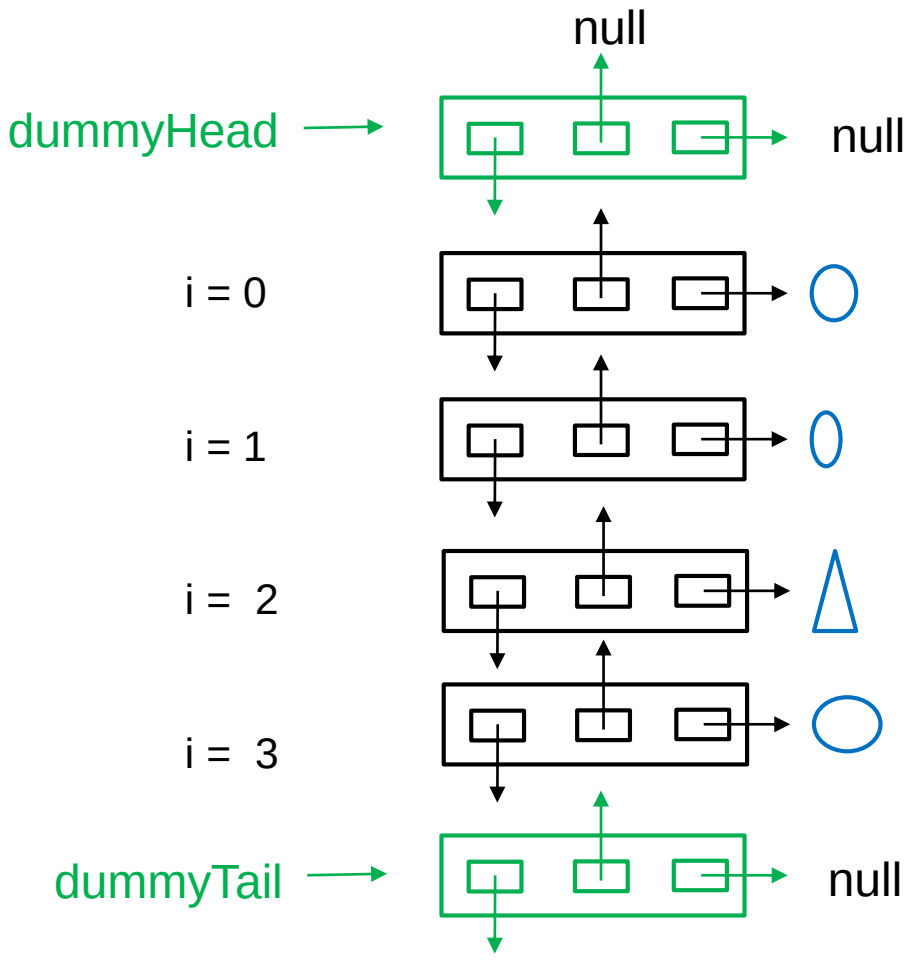
```
E    get( i ) {

     node =  getNode(i);
     return node.element;

}
```



CODE FOR GETTING AN ELEMENT AT INDEX I

```
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
}
```

Is this the most *efficient* way?

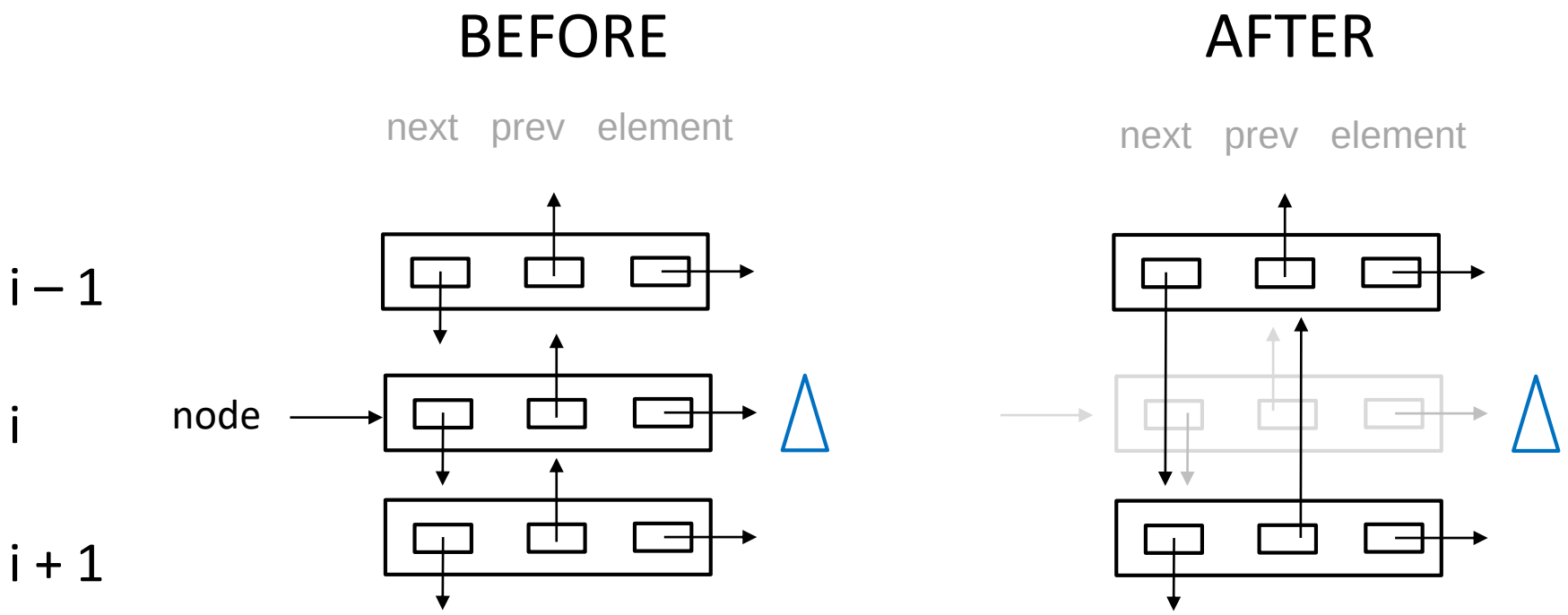


MORE EFFICIENT GETNODE… HALF THE TIME

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

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

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

}
```

Exercise (see MyCourses)



WHAT IS THE TIME COMPLEXITY OF REMOVE AT INDEX ?

- Does the number of operations to execute in **remove(i)** depend on a constant (**1**), or the size **N** of the list?
- In **Array Lists**?
  - **Best case** (removing the last position) : 1
  - **Worst case** (removing the first position) : N
  - **Average case**: ~N/2 -> depends on N
- In **Singly Linked Lists**?
  - Need to navigate to reach the position. **Worst case**: N. **Average case**: ~N/2. -> depends on N
- In **Doubly Linked Lists?**
  - **Worst case:** N/2
  - **Best case:** 1
  - **Average case:** ~N/4 -> depends on N

WHAT DOES THE TIME COMPLEXITY DEPEND ON *IN WORST CASE*

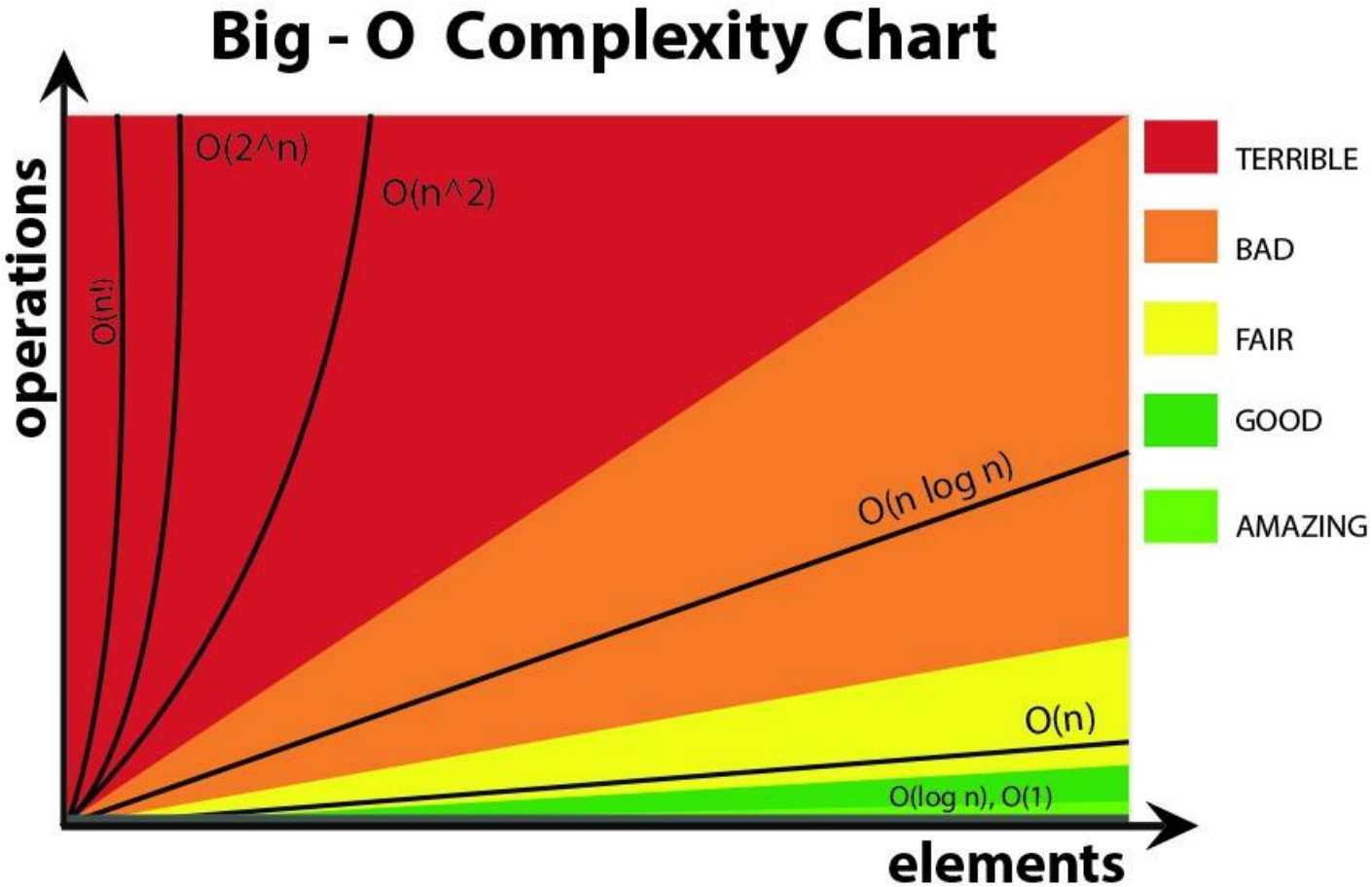
	array list	**SLinkedList**	**DLinkedList**
addFirst	size	constant	constant
removeFirst	size	constant	constant
addLast	constant	constant	constant
removeLast	constant	size	constant
remove(i)	**size**	**size**	**size**

Can we use a more formal notation to describe this?

FORMALIZING TIME COMPLEXITY

- We want to compare the worst case complexity of algorithms when the input size N grows.
- Let's call this value Big O, or O(__). O stands for **order** and describes very roughly how fast the function *grows* as N increases.

- In this notation, we do not consider constants.
- Returning the contents of an array at position i takes **constant time**, or **O(1)**.
- Iterating over all the elements of a list of size N takes linear time, or **O(N)**.
- Iterating over half the elements of a list of size N, or a quarter, is still O(N), because (1/4) is a constant and N/4 is a linear function of N, hence it still takes linear time, and it is O(N).



TIME COMPLEXITY *IN 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)**

ARRAY LIST VERSUS LINKED LIST ?

Array lists and linked lists both take **O(N)** time to add or remove from an arbitrary position in the list.

But this doesn't really say that much, right?

In practice and when N is large, array lists are faster. But the reasons are subtle and have to do with how computer memory works, in particular, how caches exploit contiguous memory allocation. You will learn about that topic in COMP 273.

In the context of this course, we will generally consider two algorithms with a time complexity of O(N) to be of the same efficiency.

DO YOU EVER NEED LINKED LISTS ?

Yes. Even if you prefer ArrayLists, you still need to understand LinkedLists. Linked lists are special cases of a general and widely used data structure called a ***tree*** which we will be discussing extensively.

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.

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

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

where ‘1′ means constant (remember we don’t care about constants with **O**, so all constants are equivalent to 1)

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

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

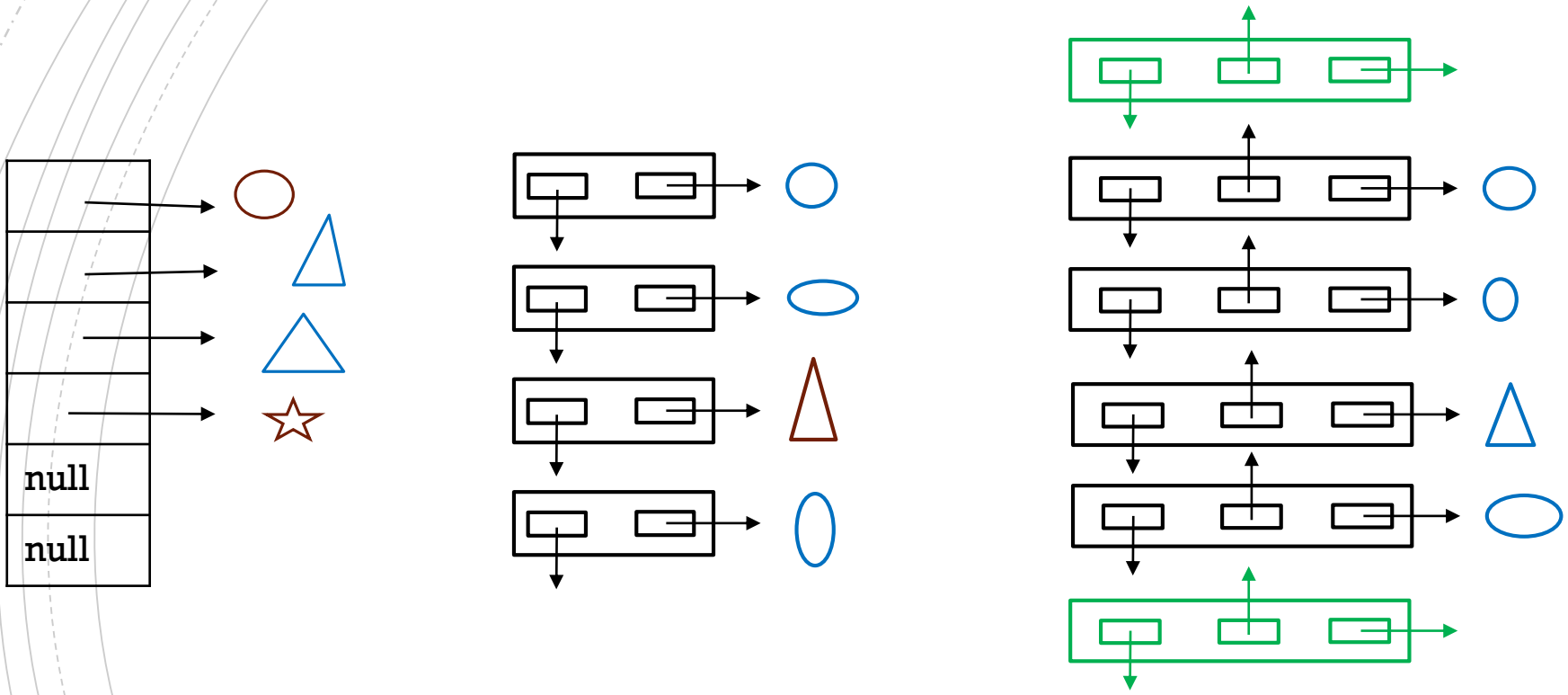
WAIT, WHAT JUST HAPPENED?

- $1 + 2 + 3 + \ldots + (N - 2) + (N - 1) + N$
- $1 + 2 + 3 + \ldots + (N - 2) + (N - 1) + N$

- $1 + N + 2 + (N - 1) + 3 + (N - 2) + \cdots + \frac{N}{2} + \frac{N}{2} + 1$
- $(N + 1) + (N + 1) + (N + 1) + \ldots + (N + 1)$

- $\frac{N}{2}(N + 1) \Rightarrow O(N^2)$

WHAT ABOUT "SPACE COMPLEXITY" ?



All three data structures use space O(N) for a list of size N.
Linked lists use 2x (single) or 3x (double) the number of references
O(N+1), O(2N) and O(3N) are all O(N)

ANNOUNCEMENTS

- Assignment 1 due on Sunday
  - Read the instructions carefully!
- Assignment 2 comes out on Monday