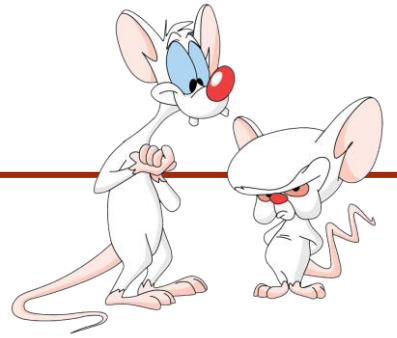
COMP 250

INTRODUCTION TO COMPUTER SCIENCE

Week 8-3 : OOD 10 Iterator

Giulia Alberini, Fall 2020

WHAT ARE WE GOING TO DO IN THIS VIDEO?



- Java interfaces `Iterable` and `Iterator`

ITERABLE

REMEMBER THE FOR-EACH LOOP?

```
int[] numbers = {1,2,3,4,5};
for(int element: numbers) {
    System.out.println(element);
}
```


The for-each loop (also called enhanced for loop) can make your code more readable and can be convenient to use. It is not helpful when you need to refer to the index of an element. For certain data structures is the only loop we can use…

ITERABLE AND ITERATOR

- The use of a *for-each* loop is made possible by the use of two interfaces: `Iterator` and `Iterable`.

- For beginners, the t nfusing. Even though they are similar, they refer to two di gs:
  - Objects of type `Iterable` are representations of a series of elements that can be iterated over. (e.g. a specific `ArrayList`)
  - Objects of type `Iterator` allows you to iterate through objects that represent a collection (a series of elements).

JAVA ITERABLE INTERFACE

```
public interface Iterable<T> {
  public Iterator<T> iterator();
}
```

```
public interface Iterator<T> {
   boolean hasNext();
   T next();   // returns current,
               // and advances to next
   void remove(); // optional, ignore it
}
```


- A class that implements `Iterable` needs to implement the `iterator()` method. The `iterator()` method returns an object of type `Iterator` that can then be used to iterate through the elements of *this* instance.

- A class that implements `Iterator` needs to implement the methods `hasNext()` and `next()`.

OBSERVATION

```
public interface Iterable<T> {
  public Iterator<T> iterator();
}
```

```
public interface Iterator<T> {
   boolean hasNext();
   T next();   // returns current,
               // and advances to next
   void remove(); // optional, ignore it
}
```

- The `iterator()` method returns an iterator to the start of the collection. Using `hasNext()` and `next()` you can move forward in the collection. If you want to traverse the collection again, you'll need a new `Iterator`.

ITERABLE AND FOR-EACH LOOP

- Implementing the `Iterable` interface allows an object to make use of the for-each lo ally calling the `iterator()` metho

HOW TO IMPLEMENT THE INTERFACES

- As always when implementing interfaces, a class that implements an interface must implement every method from such interface.
- Generally, when we writ the interface `Iterable` we also write a class that imp `rator`. Often, such class is defined as an inner class of the first class.
- Why? To implement `Iterable`, we need nt the method `iterator()`. Such method need to return an object of type `Iterator` that can iterate through the elements of a specific object of the outer class. We need a class that can create such object.

EXAMPLE

```
public class MyCollection<T> implements Iterable<T> {
    public MyIterator<T> iterator() {
        return new MyIterator<T>(this);
    }
}
```

```
public class MyIterator<E> implements It                {
    public MyIterator(MyCollection<E> c) {
        :
    }
}
```

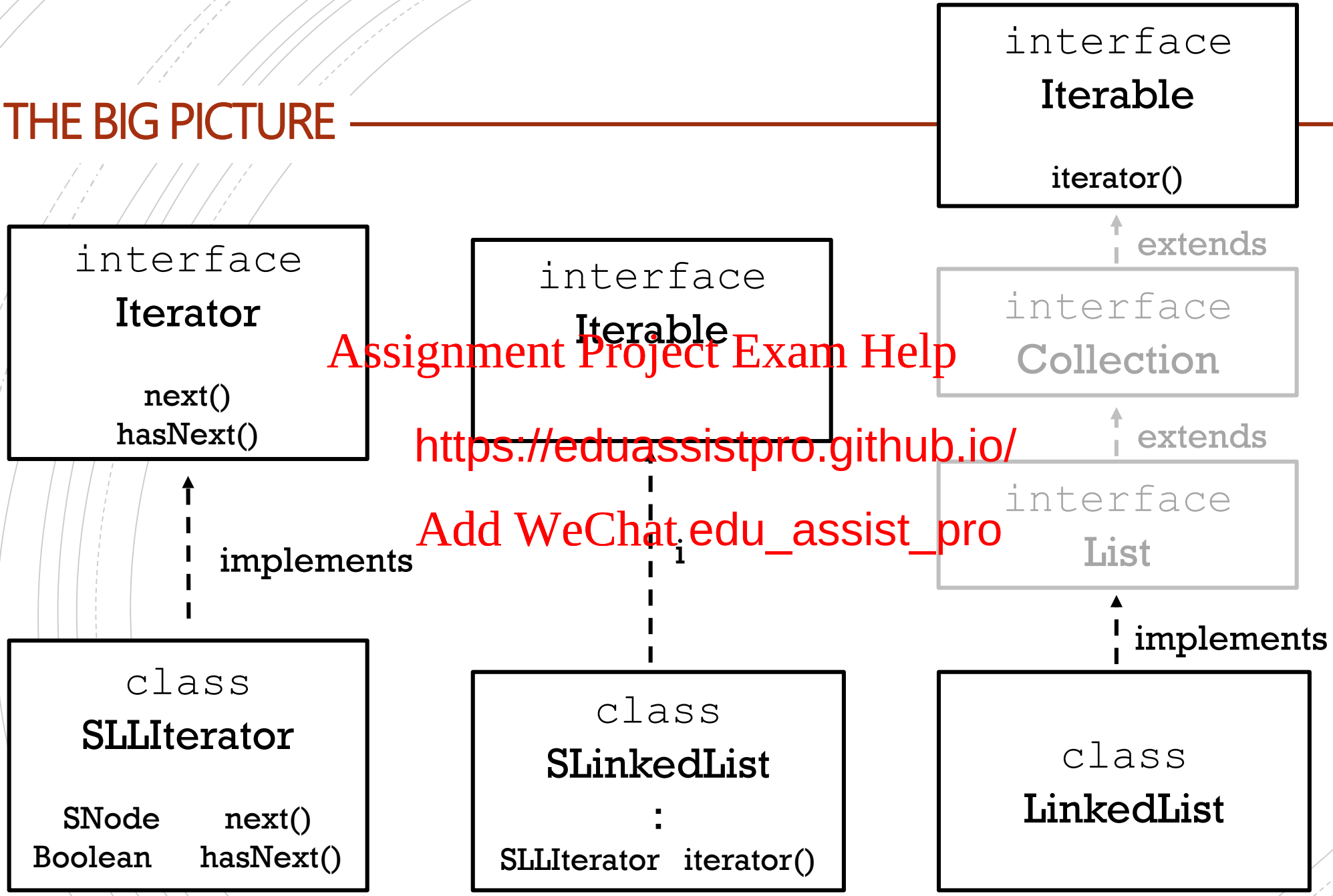

- **In general, if the class** `MyIterator` **is used only by the class** `MyCollection`**, good practice is to make that class a private inner class of** `MyCollection`**.**

SLinkedList

- `iterator()` **returns an object of type** `Iterator` **that points to the head of the provided list.**
- `next()` **returns the element of the list that the** `Iterator` **is currently referencing, and then moves to the next node.**

```
public class SLinkedList<E> implements Iterable<E> {
    private SNode<E> head;
    public SLLIterator iterator() {
        return new SLLIterator(this);
    }
    private class SLLIterator implements Iterator<E> {
        SNode<E> cur;
        SLLIterator(SLinkedList<E>  list) {
                                    ;
                                                  () {
            return                                ll);
        }
        public E next() {
            SNode<E> tmp = cur;
            cur = cur.next;
            return tmp.element;
        }
    }
}
```

THE BIG PICTURE



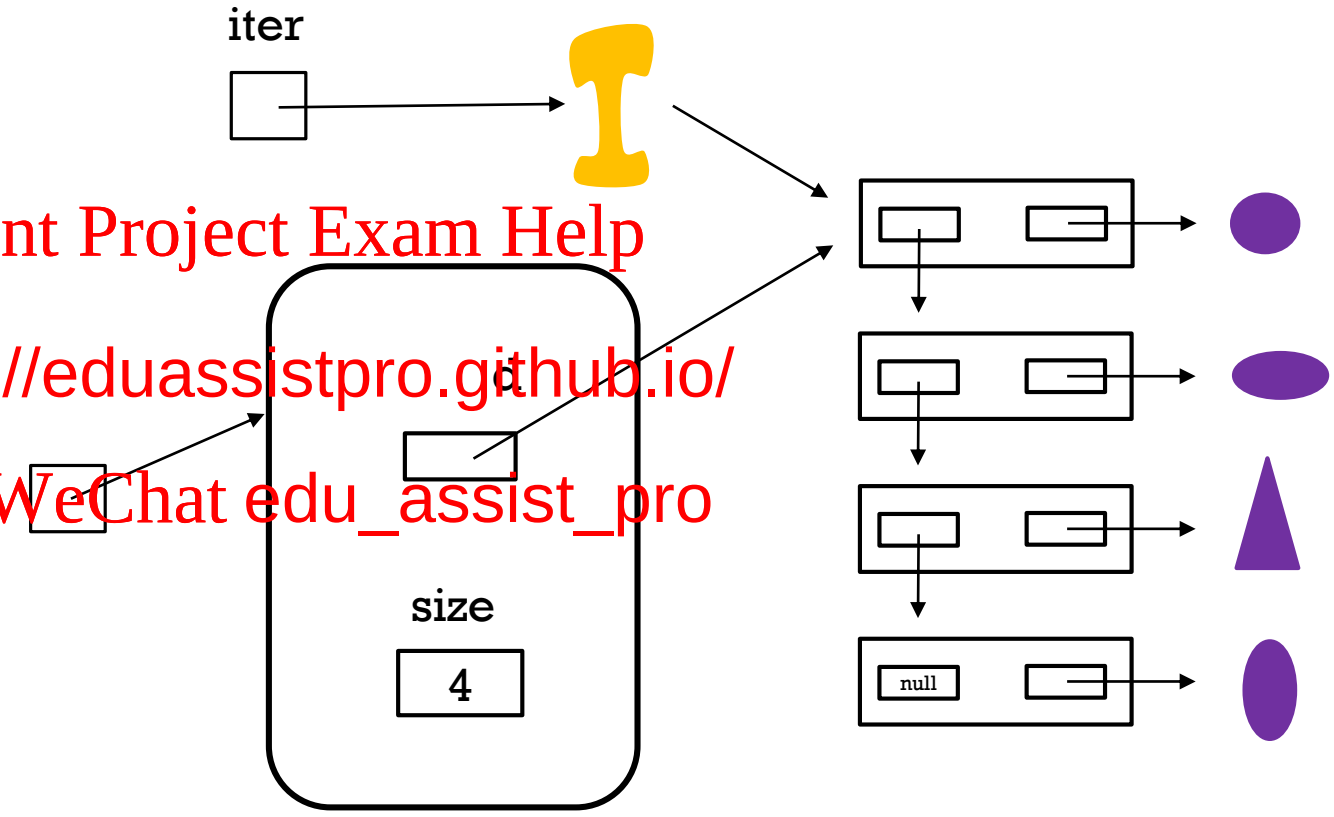
EXAMPLE

- Suppose we have a SLinkedList of Shapes:

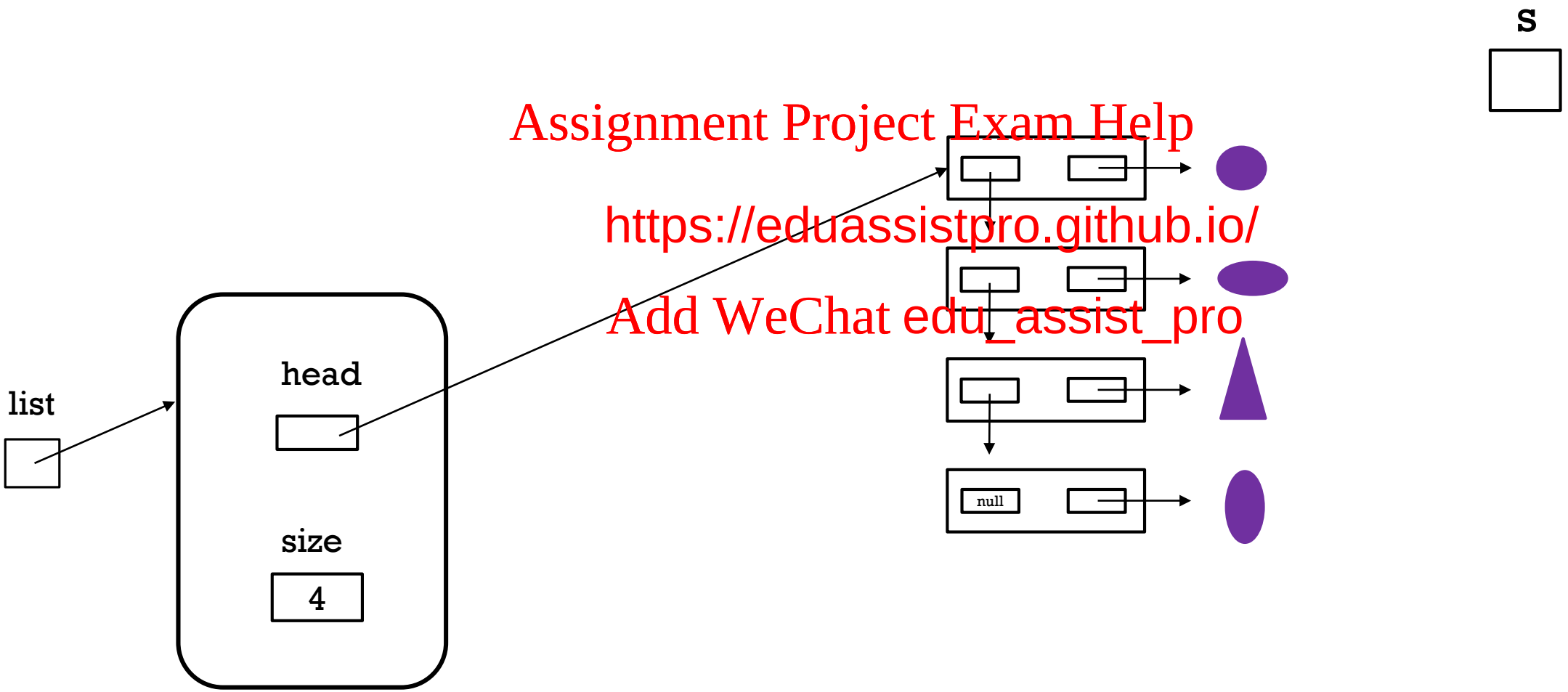
```
SLinkedList<Shape> list = …
```

- Then by calling `iterator()` we an object of type Iterator that points to the head of the list.

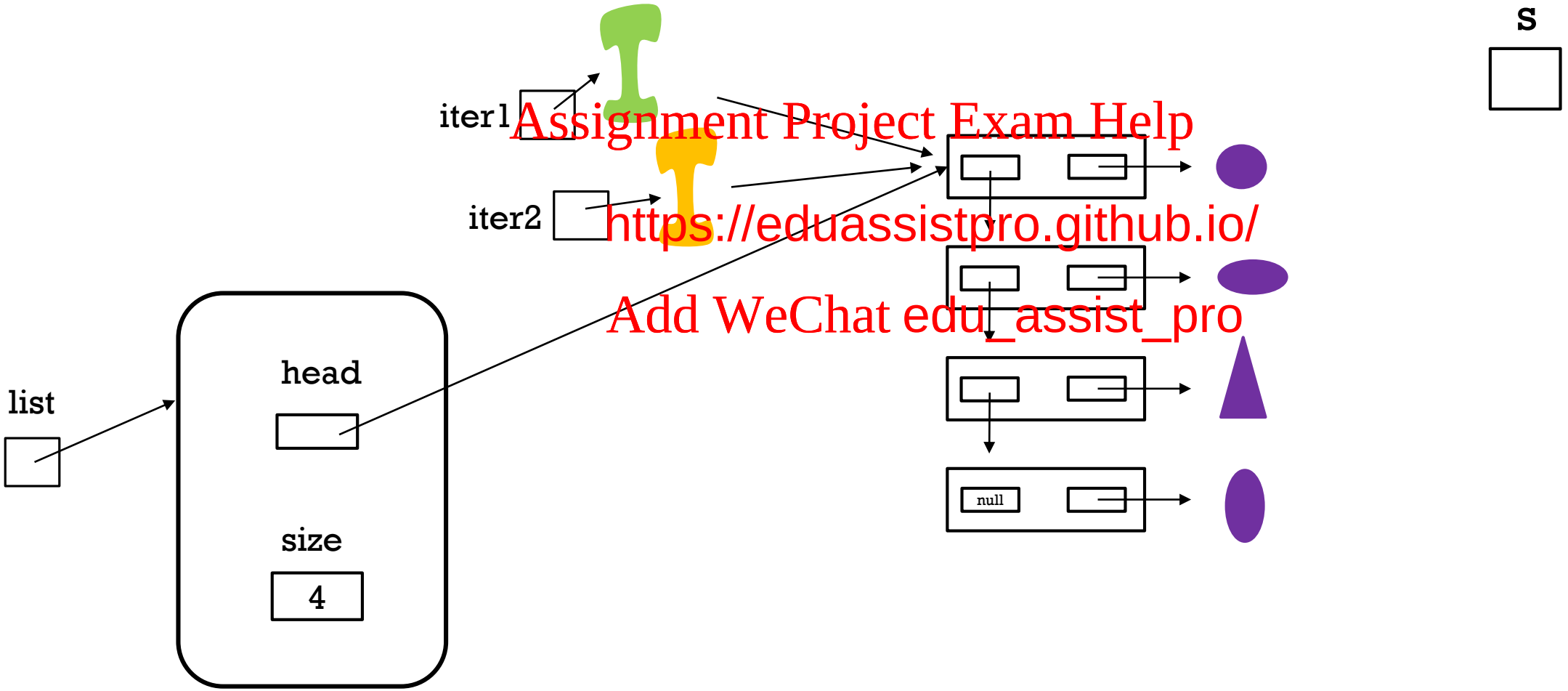
```
Iterator iter = list.iterator();
```



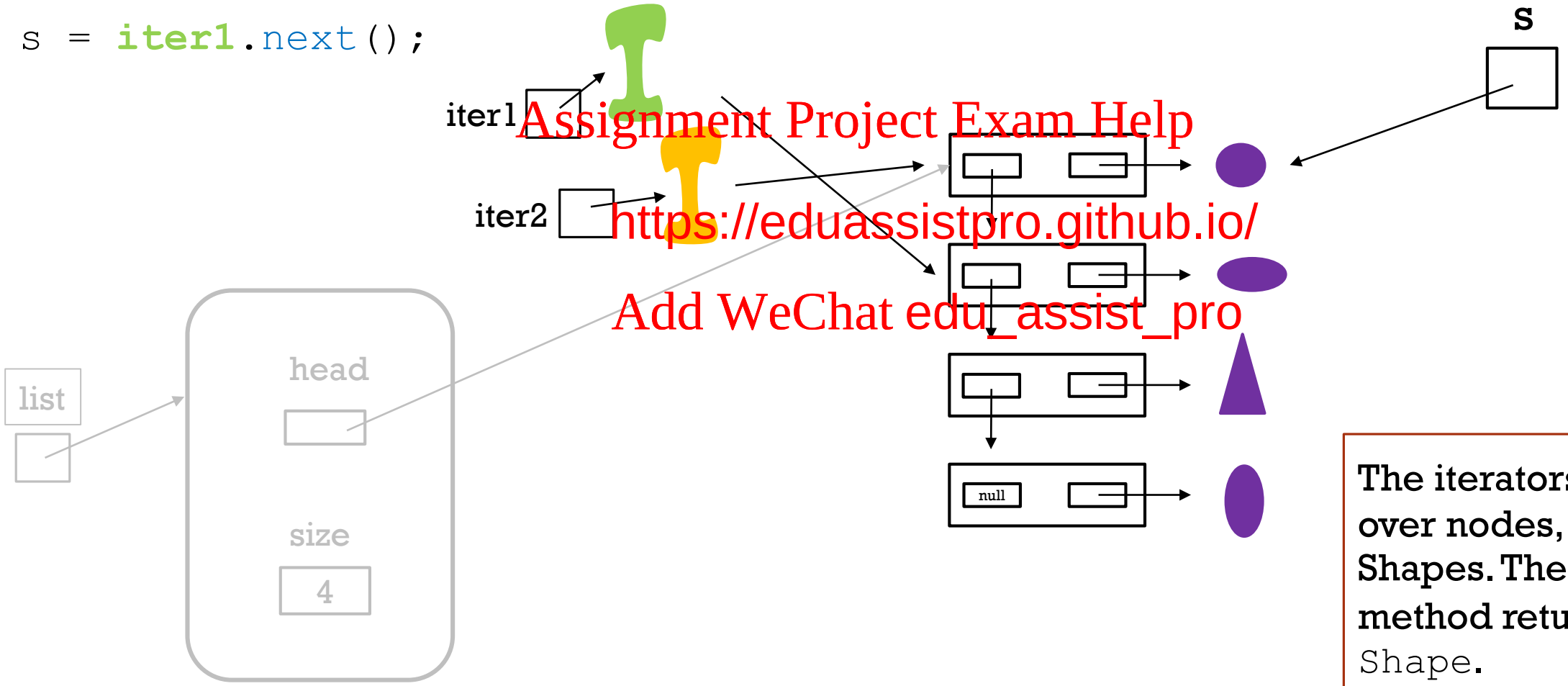
```
SLinkedList<Shape> list = … ;
Shape s;
```



```
SLinkedList<Shape> list = … ;
Shape s;
Iterator<Shape> iter1 = list.iterator();
Iterator<Shape> iter2 = list.iterator();
```

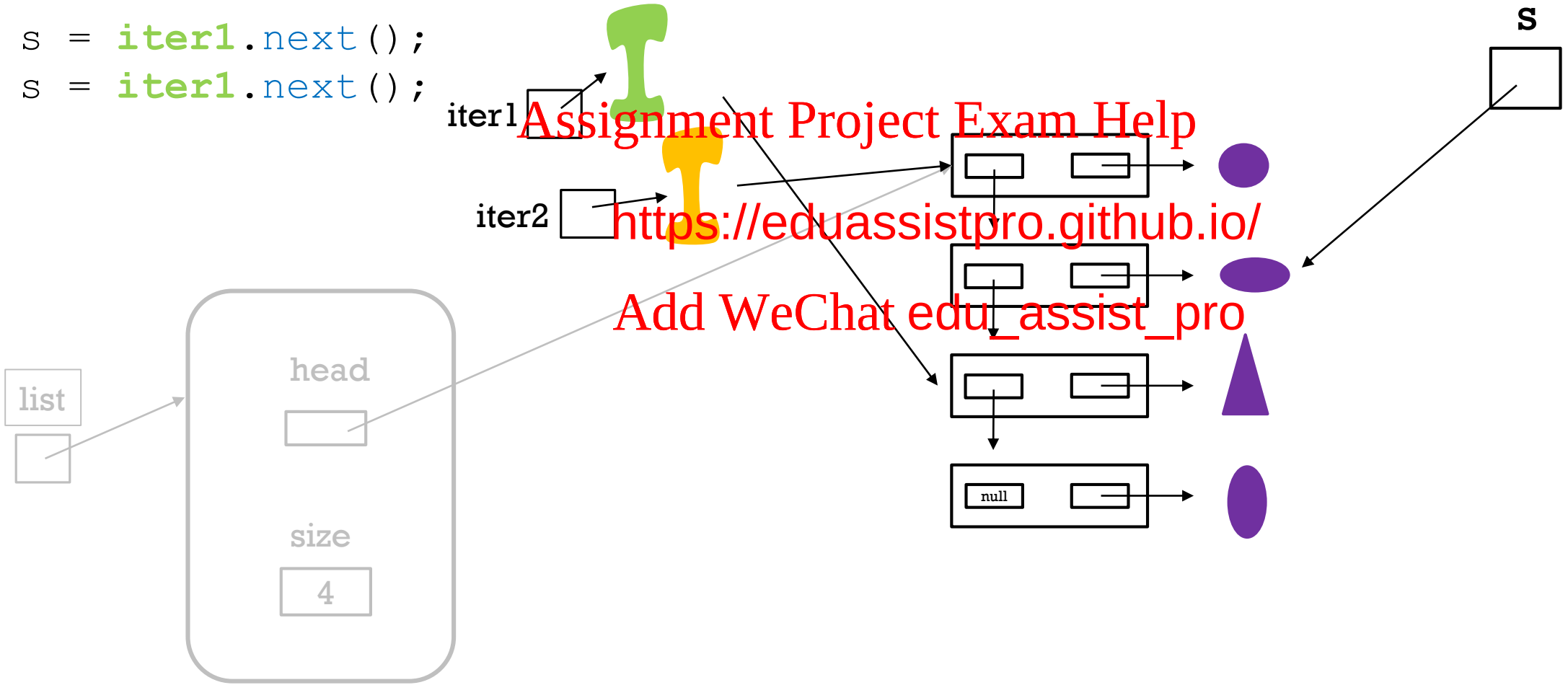


```
SLinkedList<Shape> list = … ;
Shape s;
Iterator<Shape> iter1 = list.iterator();
Iterator<Shape> iter2 = list.iterator();

s = iter1.next();
```

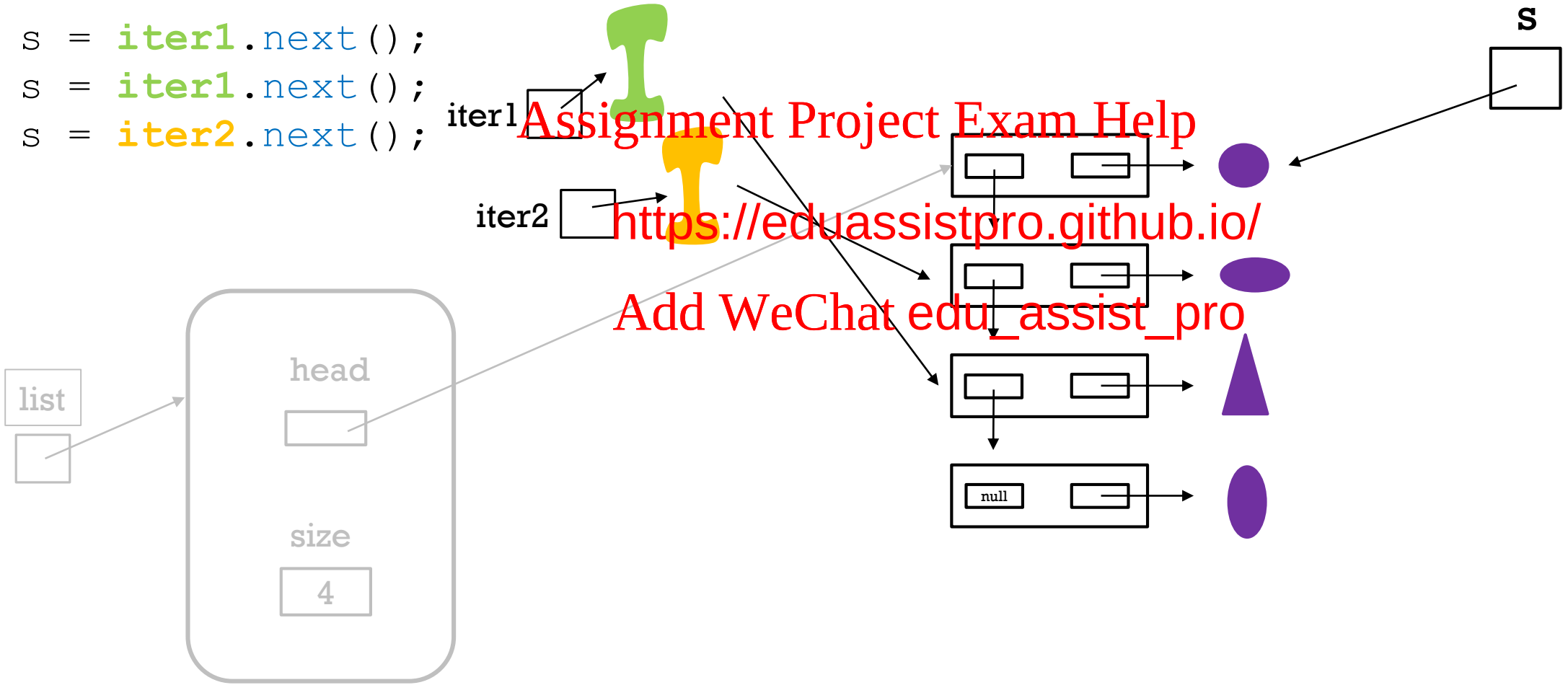


The iterators iterate over nodes, not Shapes. The `next()` method returns a `Shape`.

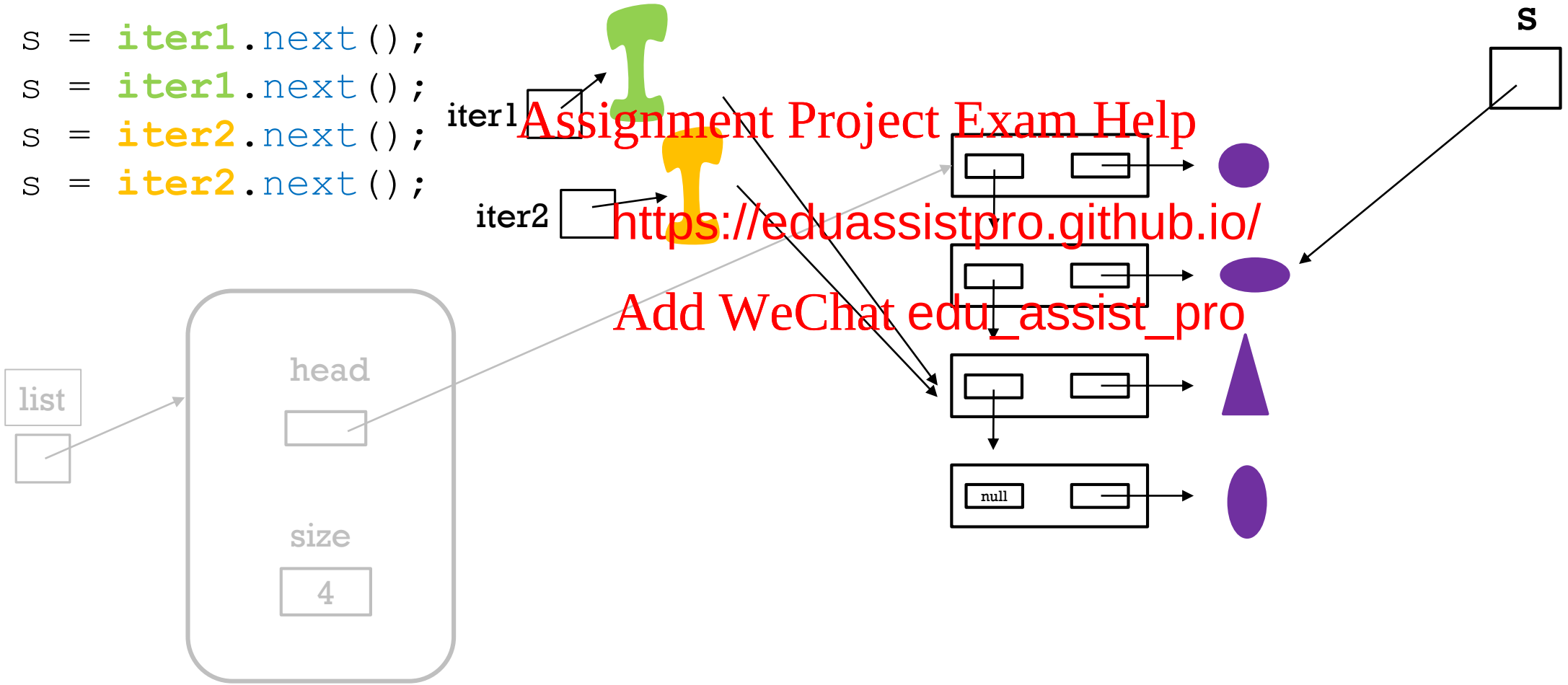
```
SLinkedList<Shape> list = … ;
Shape s;
Iterator<Shape> iter1 = list.iterator();
Iterator<Shape> iter2 = list.iterator();

s = iter1.next();
s = iter1.next();
```



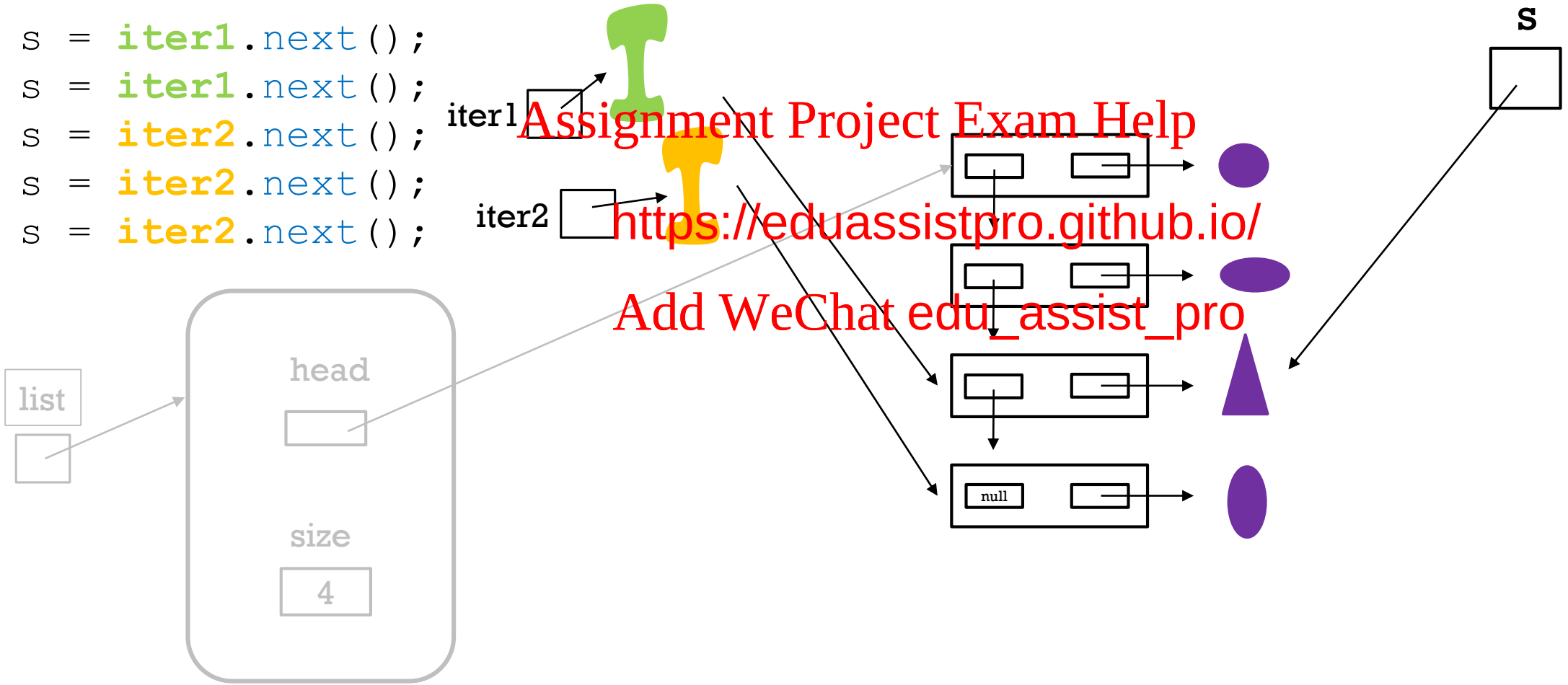
```
SLinkedList<Shape> list = … ;
Shape s;
Iterator<Shape> iter1 = list.iterator();
Iterator<Shape> iter2 = list.iterator();

s = iter1.next();
s = iter1.next();
s = iter2.next();
```



```
SLinkedList<Shape> list = … ;
Shape s;
Iterator<Shape> iter1 = list.iterator();
Iterator<Shape> iter2 = list.iterator();

s = iter1.next();
s = iter1.next();
s = iter2.next();
s = iter2.next();
```



```
SLinkedList<Shape> list = … ;
Shape s;
Iterator<Shape> iter1 = list.iterator();
Iterator<Shape> iter2 = list.iterator();

s = iter1.next();
s = iter1.next();
s = iter2.next();
s = iter2.next();
s = iter2.next();
```

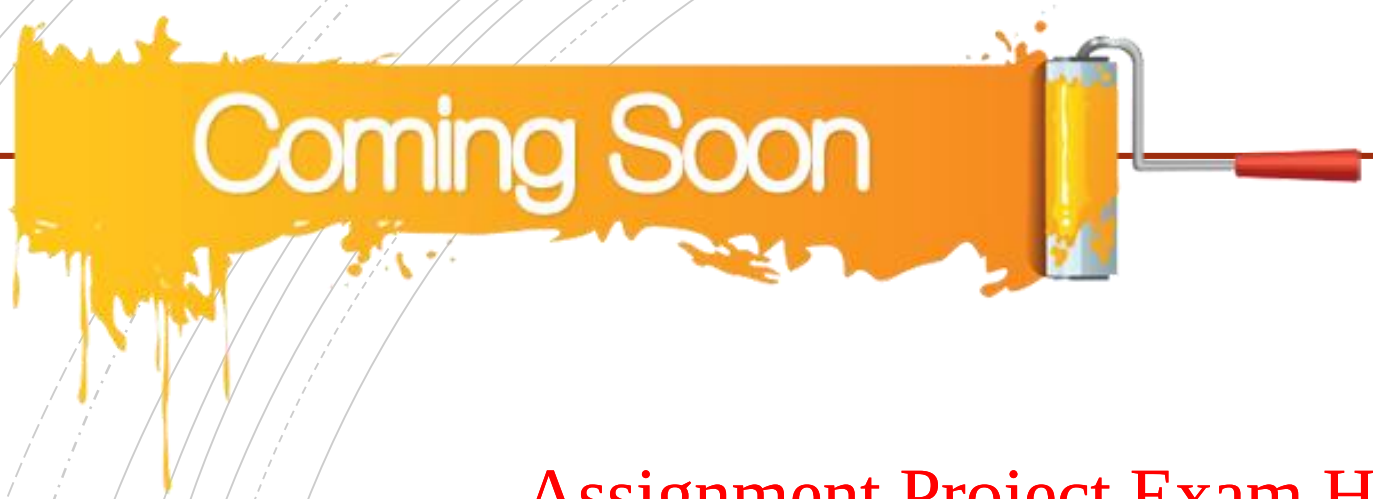


ITERATING THROUGH ELEMENTS IN A LINKED LIST

- What is the time complexity of the following two snippet of code? (suppose the size of the list is N)


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

```
for (E element : list)
    System.out.println(e);
```



In the next

- Inductio