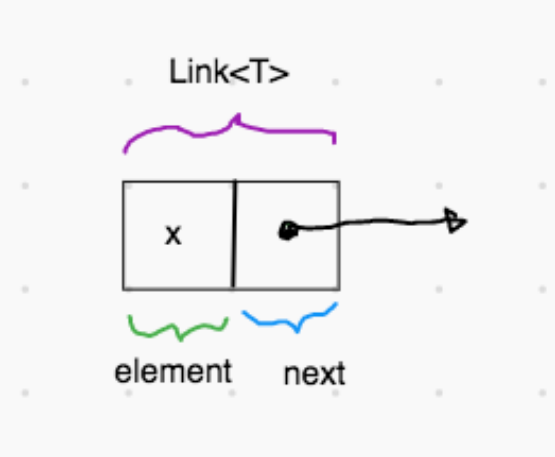
Links

```
class Link<T> {
    public T element;
    public Link<T> next;

    public Link() {
    }

    public Link(T element) {
        this.element = element;
    }
}
```



Draw memory for this:

```
Link<Integer> head = new Link();
head.element = 1;
head.next = new Link();
head.next.element = 2;
head.next.next = new Link();
head.next.next.element = 3;
head.next.next.next = new Link();
head.next.next.next.element = 4;
```

Creates a link ”chain” (aka a link list)

To access a ”random” element, we will NOT do something like this:

```
head.next.next.next.next.next.next.next.next.next.next.next.next.next.next.next.next.next.
```

- `head` is traditionally used to store the first link in the chain.

Example: write code to print the elements stored in a chain.

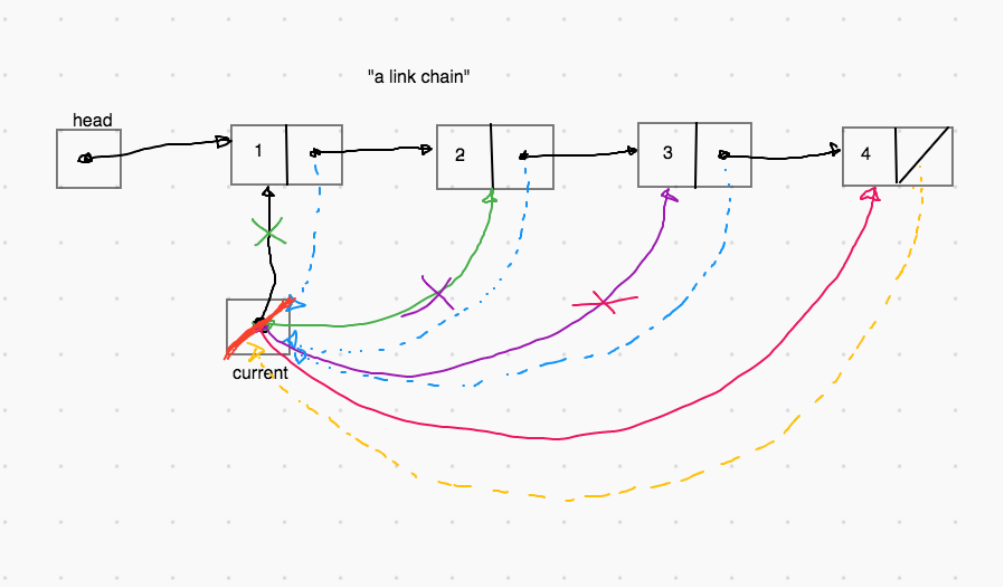
Assume that a variable `head` is referencing the first link in chain. Use a `while` loop. Version 1. Use `current !` null= as looping condition:

```
Link<T> current = head;
while(current != null) {
    sout(current.element);
    current = current.next;  // call this: step
}
```

Version 2. Use `current.next !` null= as looping condition:

```
if(head != null) {
    Link<T> current = head;
    while(current.next != null) {
        sout(current.element);
        current = current.next;  // call this: step
    }
    sout(current.element);
}
```

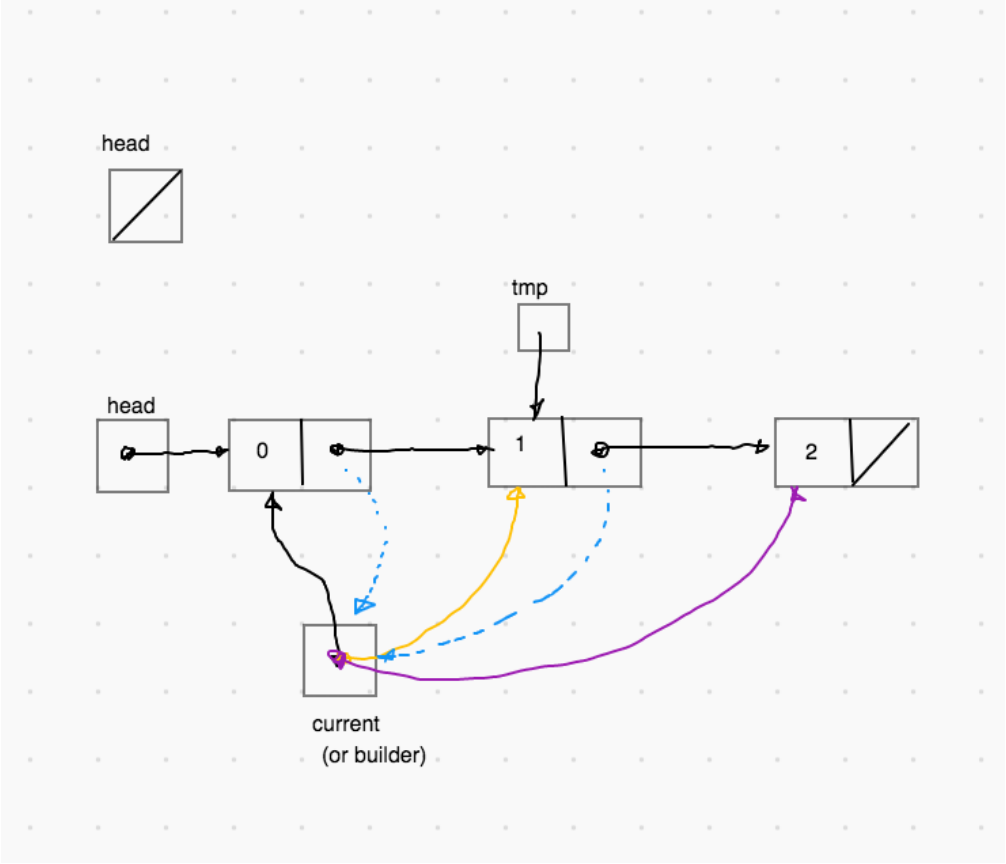
The second version is more complicated since the chain can have no links.



Why a second variable `current`? If we stepped `head` we would lose the links!

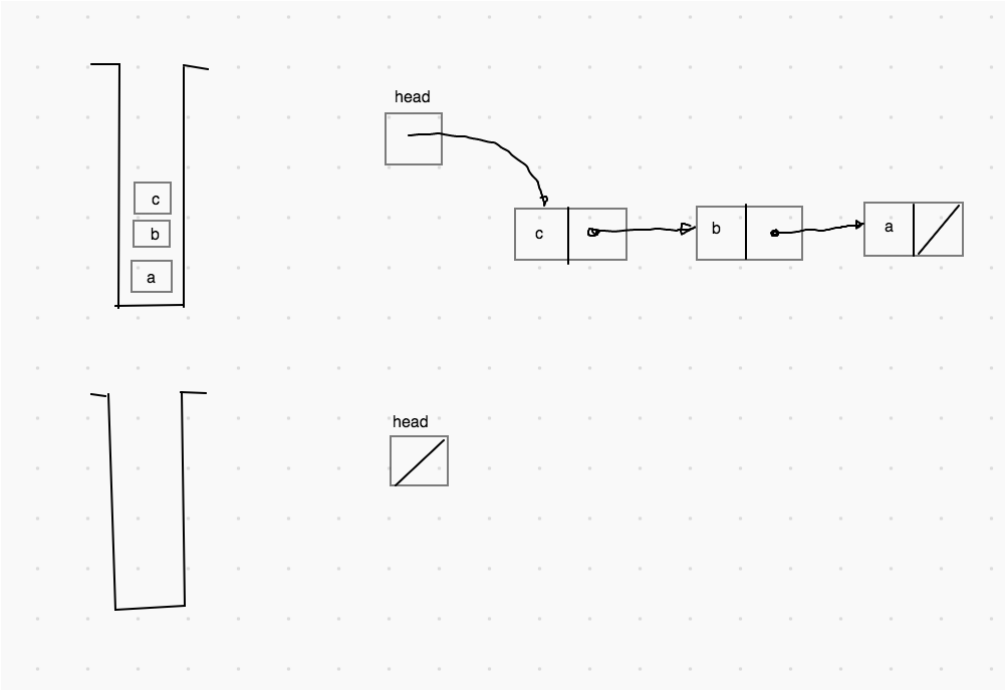
Exercise: code a method that creates a chain containing integer elements [0, n-1]

```
public Link<Integer> makeChain(int n) {
    if(n < 0) throw new IllegalArgumentException();
    if(n == 0)
        return null;

    Link<Integer> head = new Link<>(0);
    Link<Integer> builder = head;
    for(int i = 1; i < n; i++) {
        builder.next = new Link<>(i);
        builder = builder.next;
    }
    return head;
}
```



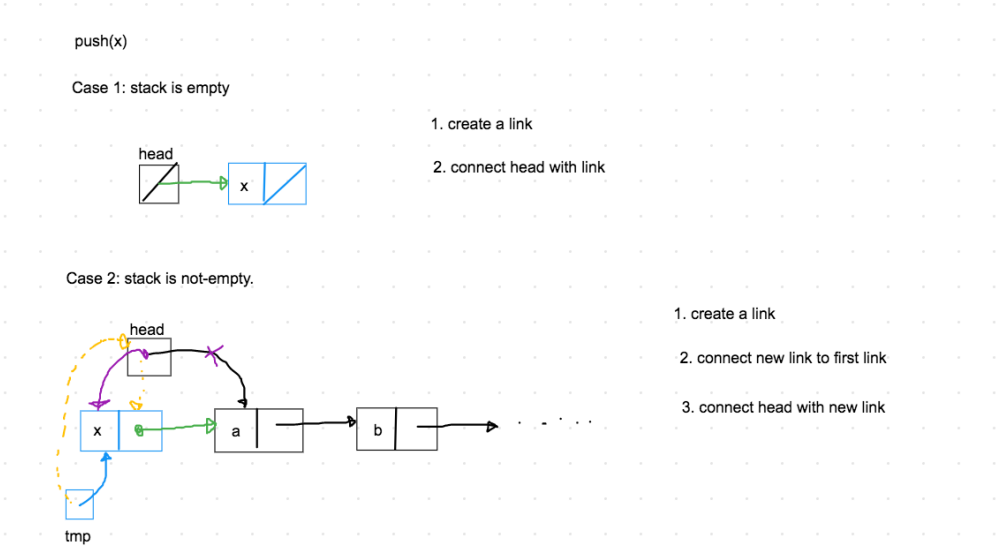
Stack API reimplemented with a link chain

We will re-implement the Stack with elements stored in a link chain:




```
public class LinkStack<T> {

    // Inner class is a class declared inside another class
    // why? scope, implementation detail
    private class Link<T> {
        public T element;
        public Link<T> next;

        public Link() {
        }

        public Link(T element) {
            this.element = element;
        }
    }

    // fields
    private Link<T> head;

    public LinkStack() {
        head = null;  // unnecessary initialization
    }
```

push(x)



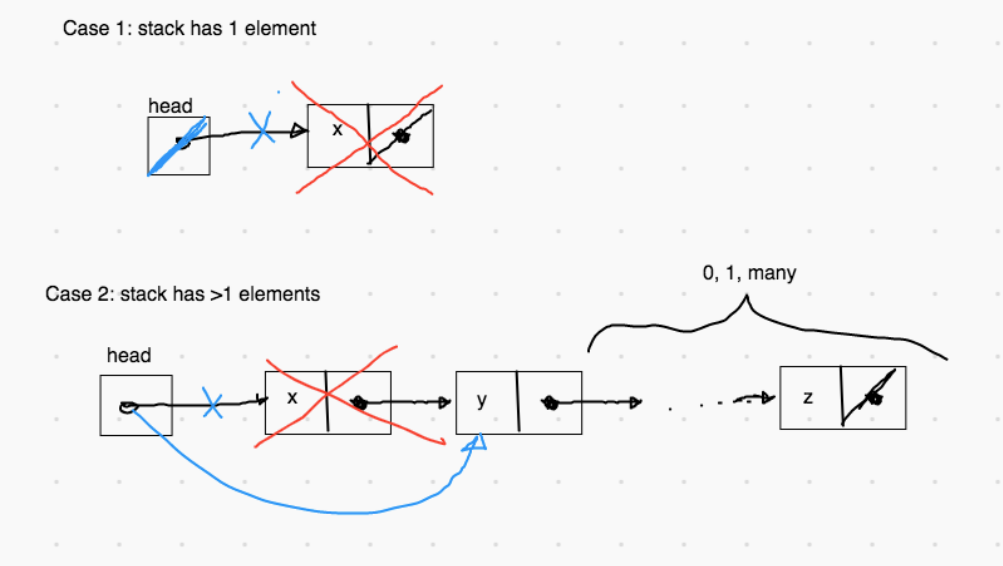

```
public void push(T element) {
   if(head == null)
       head = new Link<>(element);
   else { // not empty
       Link<Integer> tmp = new Link<>(element);
       tmp.next = head;
       head = tmp;
   }
}
```

We no longer need the check:

```
if(isFull())
    throw new StackOverflowException();
```

Since we can continue making links as needed, i.e.: there is no longer a capacity.

pop()



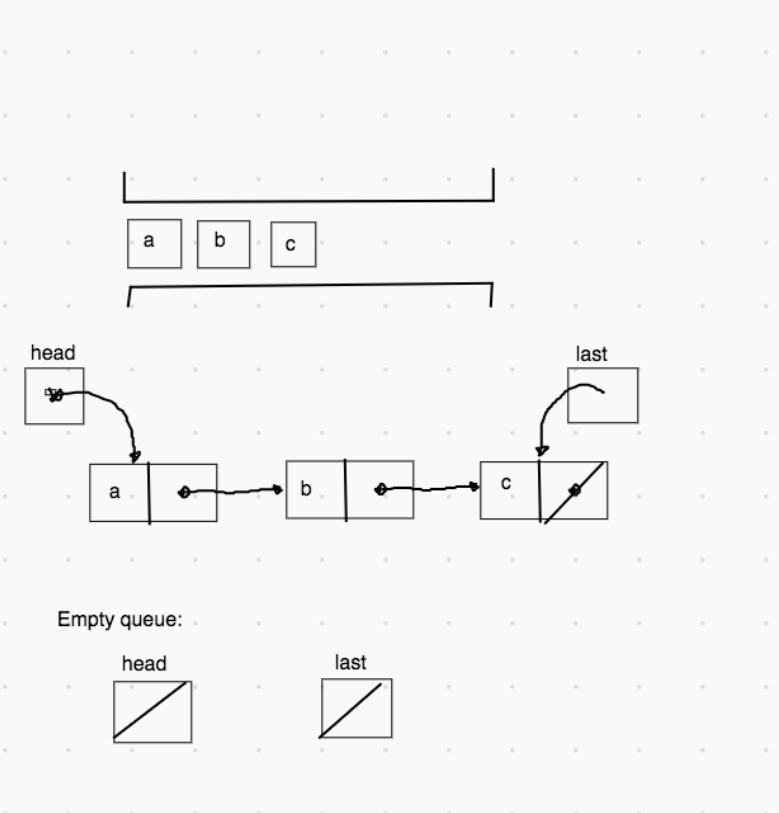

```
public T pop() {
    if(isEmpty())
        throw new StackUnderflowException();

    T tmp = head.element;
    head = head.next;
    return tmp;
}
```

isEmpty(), isFull() and size()

```
public boolean isEmpty() { return head == null; }

public boolean isFull() { return false;  }

public int size() {
    // We would have to loop over chain counting the links!
    // use a 'size' field instead
}
```

Queue API reimplemented with a link chain

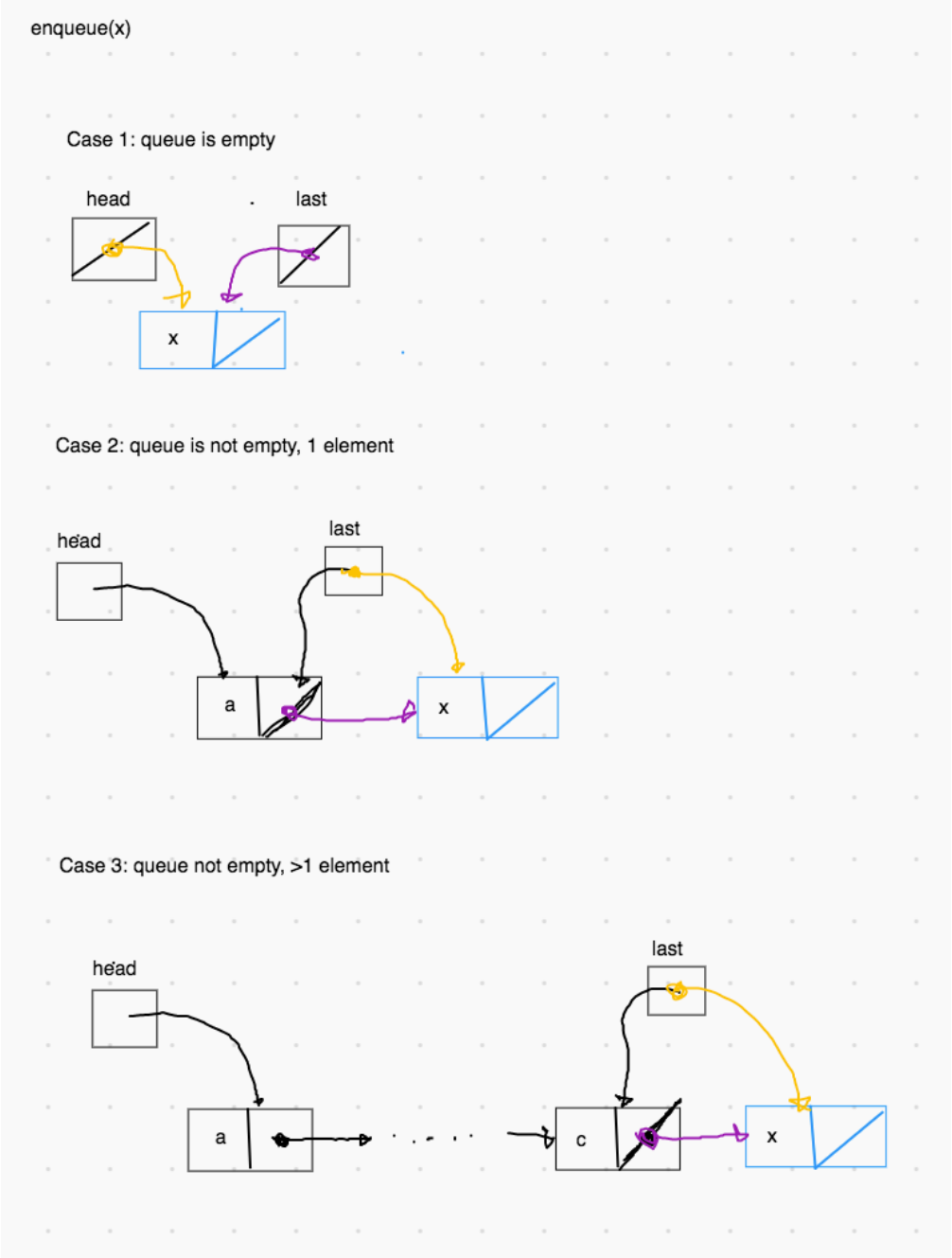
We will re-implement the Queue with elements stored in a link chain:




```
public class LinkQueue<T> {

    private class Link<T> {
        public T element;
        public Link<T> next;

        public Link() {
        }

        public Link(T element) {
            this.element = element;
        }
    }
```

```
// fields
private Link<T> head;
private Link<T> last;

public LinkQueue() {
    head = last = null;  // unnecessary
}
```

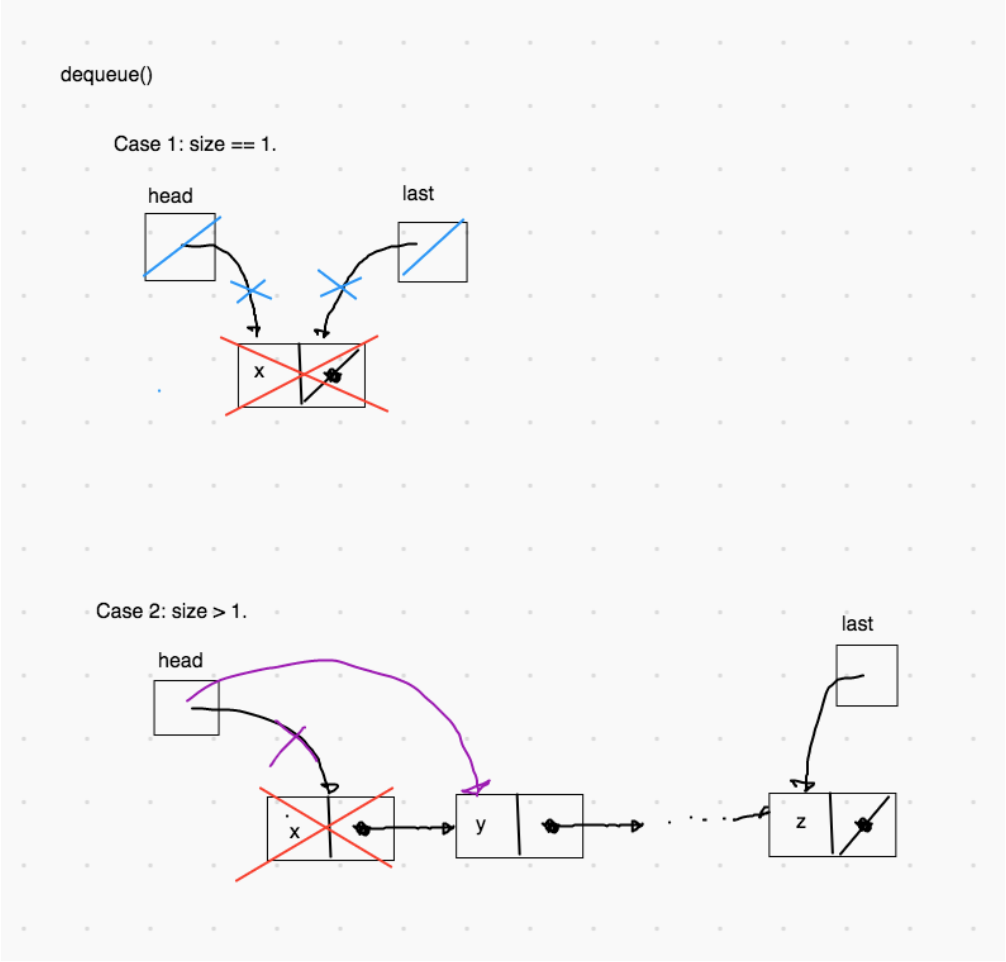
enqueue(x)




```
public void enqueue(T element) {
```

```
    Link<T> tmp = new Link<>(element);

    if(head == null) // removed unnecesary check: last == null
        head = last = tmp;
    else {
        last.next = tmp;
        last = last.next;
    }
}
```

dequeue()

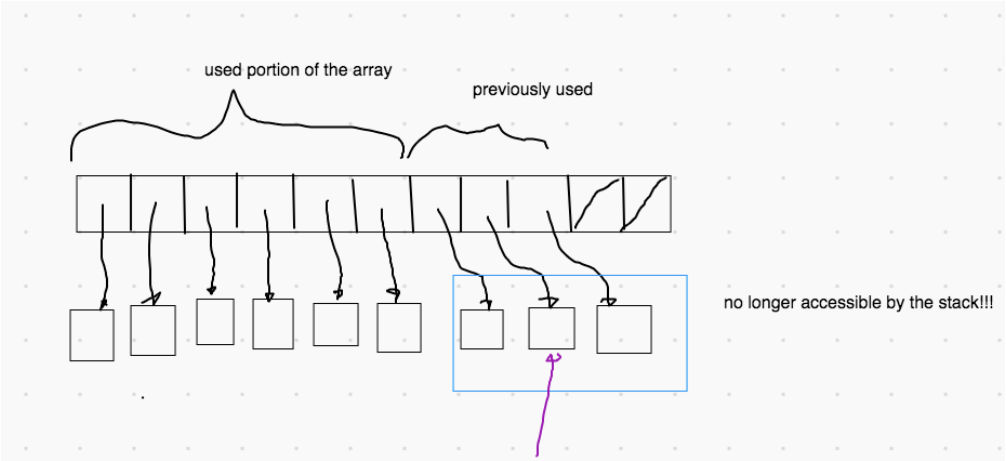



```
public T dequeue() {
```

```
    if(isEmpty())
        throw new QueueUnderflowException();

    T tmp = head.element;
    head = head.next;
    if(head == null)
        last = null;
    return tmp;

}
```

Do we need to nullify `last` in the dequeue? Yes, doing this does means that the link object can be reclaimed. Otherwise we are holding on the unused memory!

Stack (array version) pop() revisited.




```
public T pop() {
   if(isEmpty())
       throw new StackUnderflowException();
   T tmp = element[--top];
   element[top] = null;
   return tmp;
}
```

We should nullify the unused reference generated when popping, otherwise we are ”leaking” memory!