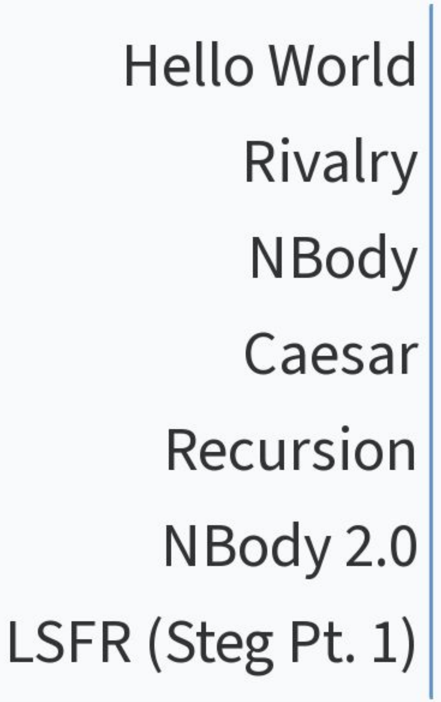
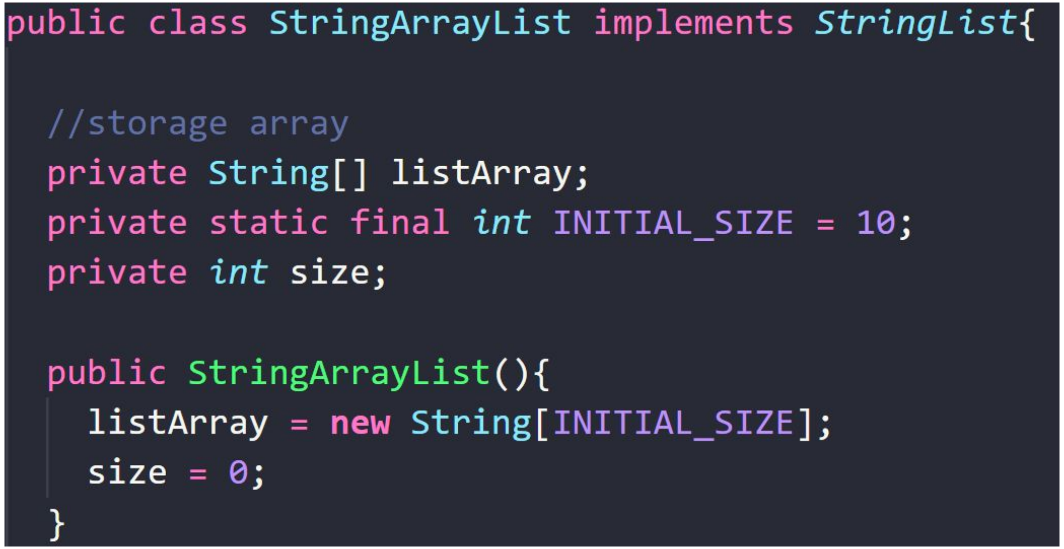
Implementing an Array List

Once more into the breach

What's the most interesting thing you've learned about this semester (in any class)?

Favorite CIS 110 HW so far?

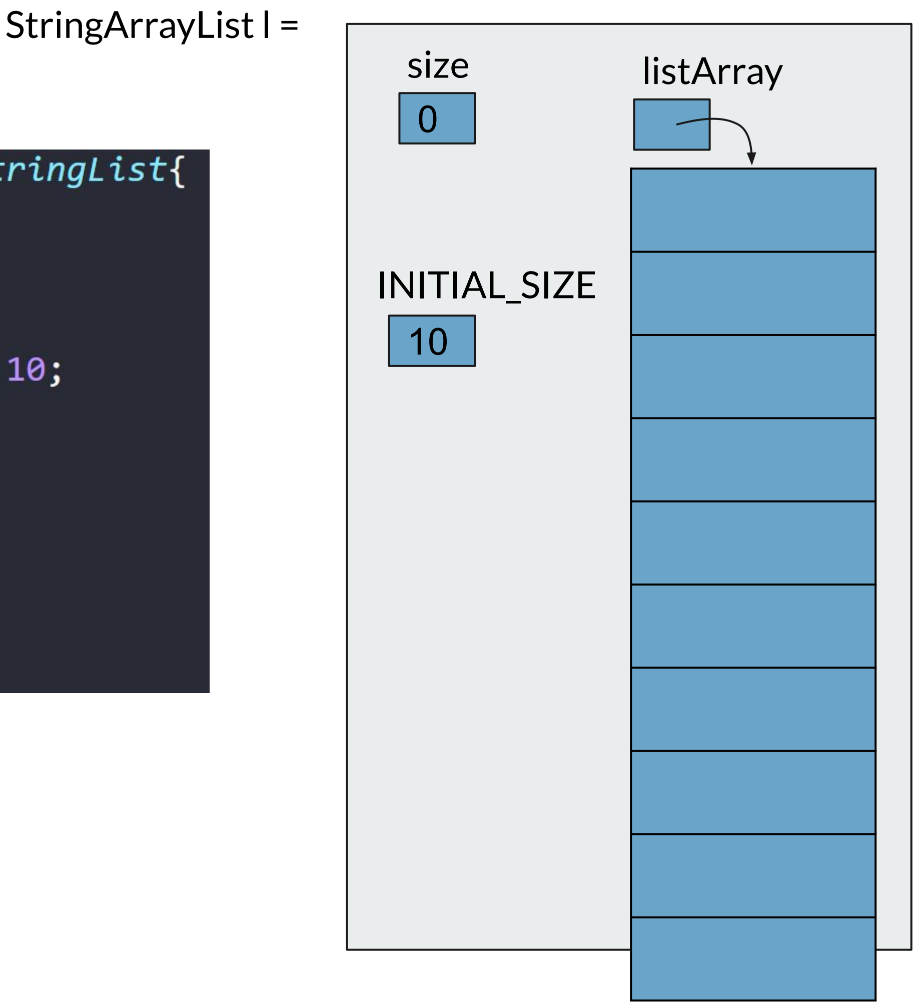




```
public class StringArrayList implements StringList{

  //storage array
  private String[] listArray;
  private static final int INITIAL_SIZE = 10;
  private int size;

  public StringArrayList(){
    listArray = new String[INITIAL_SIZE];
    size = 0;
  }
```



The ArrayList stub

Insert

Cases to Handle

Bad index

- Check that index is positive and fits inside of list; if not, throw exception
- Check that list isn’t full; if full, return false and do nothing

Good index

- Start at position *size*
- Copy over the element to the left into the current position and move to the left
- Keep going until all elements after the target *index* have been copied one position to the right
- Insert the element at *index* and increment the *size*

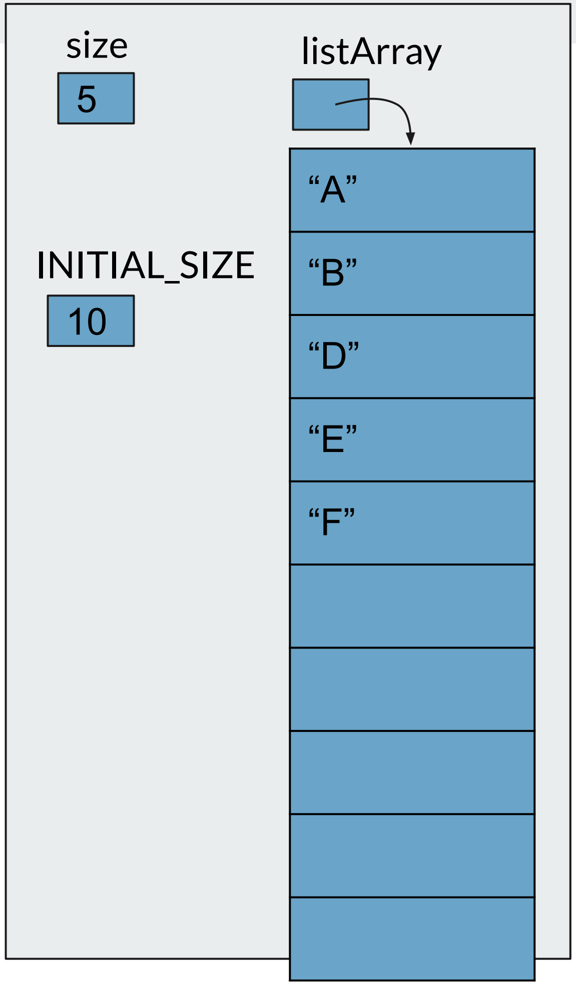
Inserting

Keep in mind: listArray is {“A”, “B”, “D”, “E”, “F”, null, null, null, null, null}

- Start at position *size*
- Copy over the element to the left into the current position and move to the left
- Keep going until all elements after the target *index* have been copied one position to the right
- Insert the element at *index* and increment the *size*

```
l.insert(2, “C”)
```

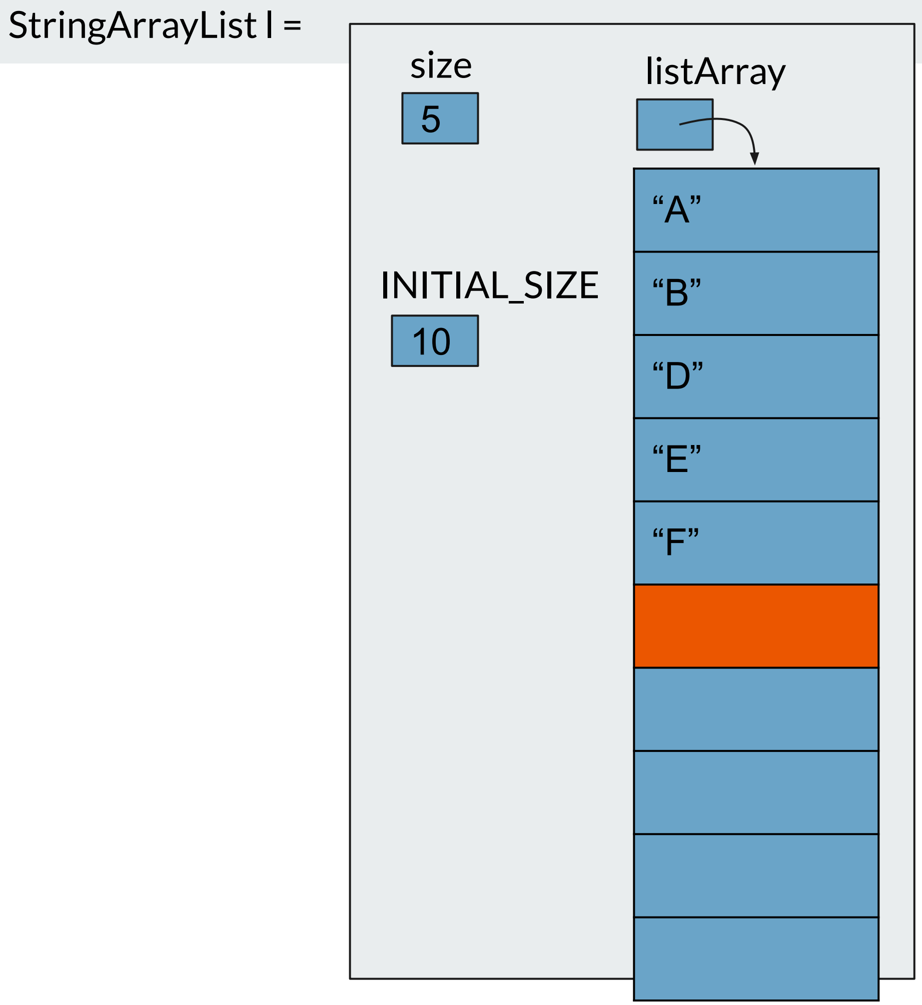
StringArrayList l =



Inserting

- **Start at position *size***
- Copy over the element to the left into the current position and move to the left
- Keep going until all elements after the target *index* have been copied one position to the right
- Insert the element at *index* and increment the *size*

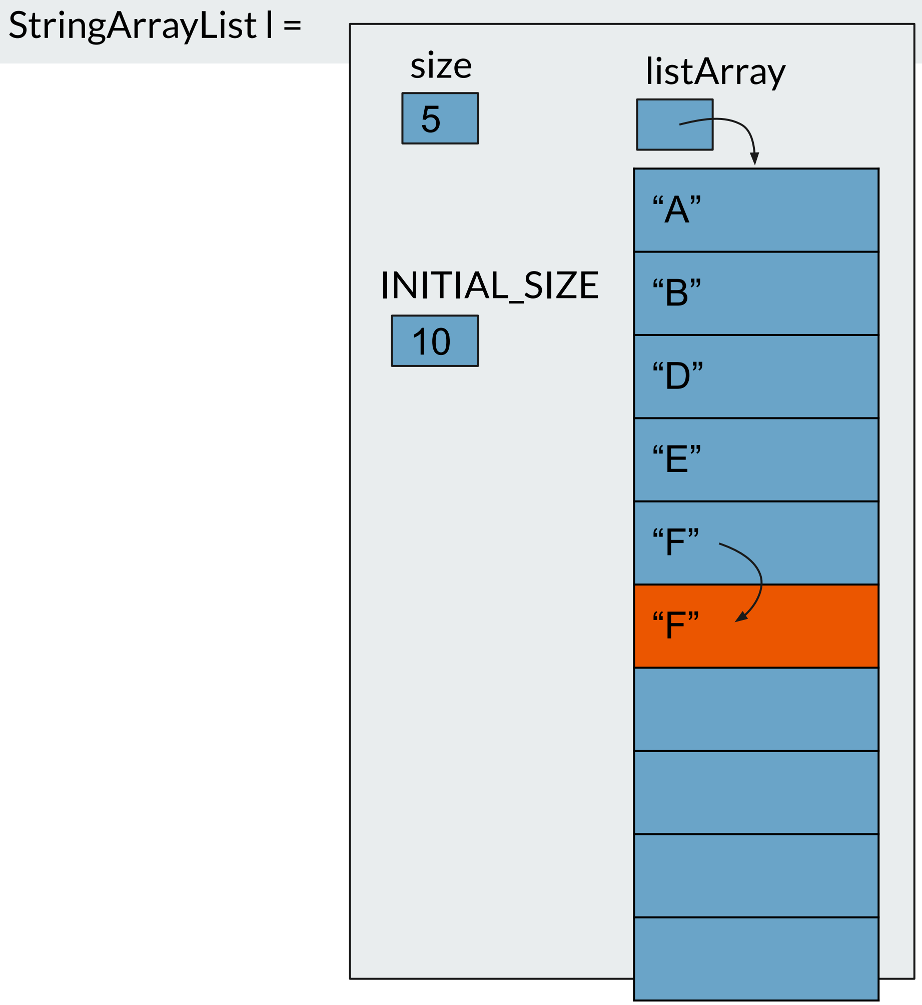
```
l.insert(2, “C”)
```



Inserting

- Start at position *size*
- **Copy over the element to the left into the current position and move to the left**
- Keep going until all elements after the target *index* have been copied one position to the right
- Insert the element at *index* and increment the *size*

```
l.insert(2, “C”)
```



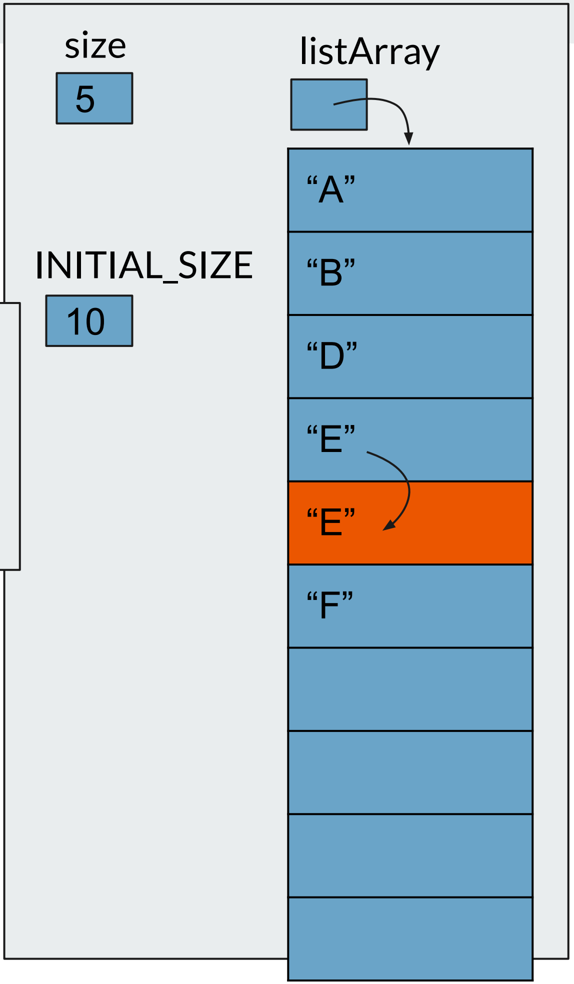
Inserting

- Start at position *size*
- **Copy over the element to the**
  current position and move to
- **Keep going until all elements**
  target *index* have been copied one position to the right
- Insert the element at *index* and increment the *size*

```
listArray[i] = listArray[i - 1];
```

```
l.insert(2, "C")
```

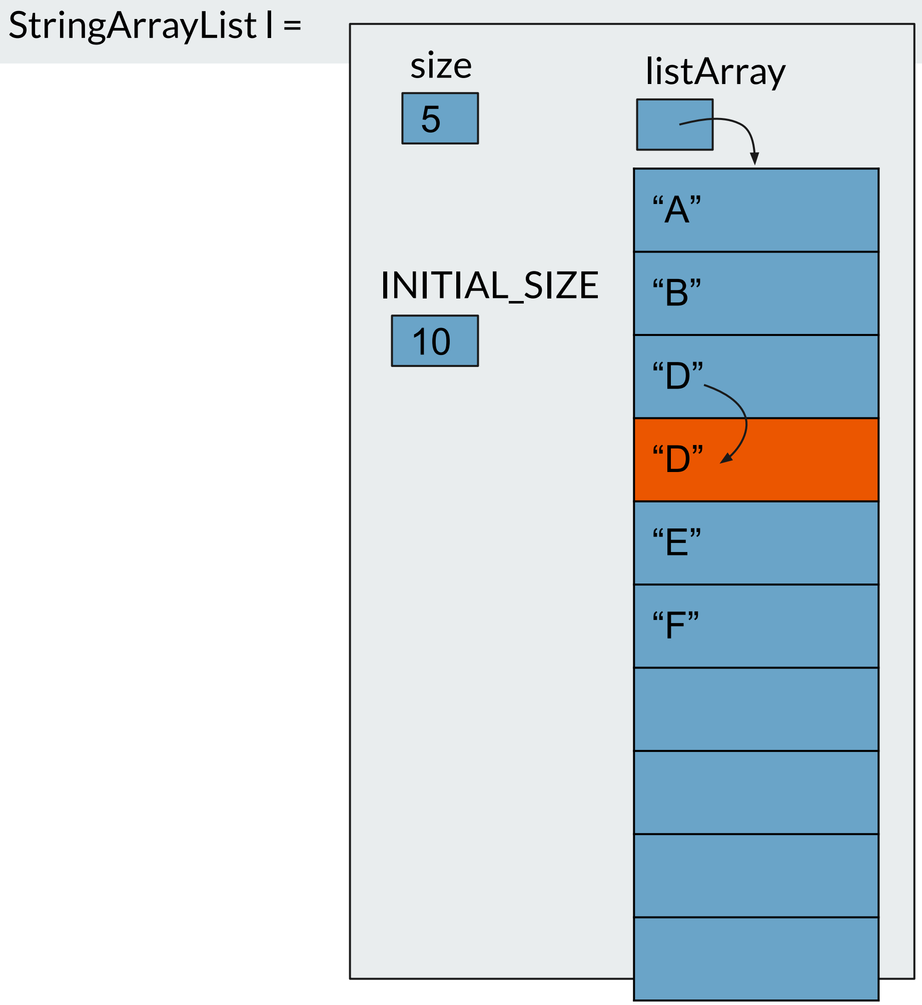
StringArrayList l =



Inserting

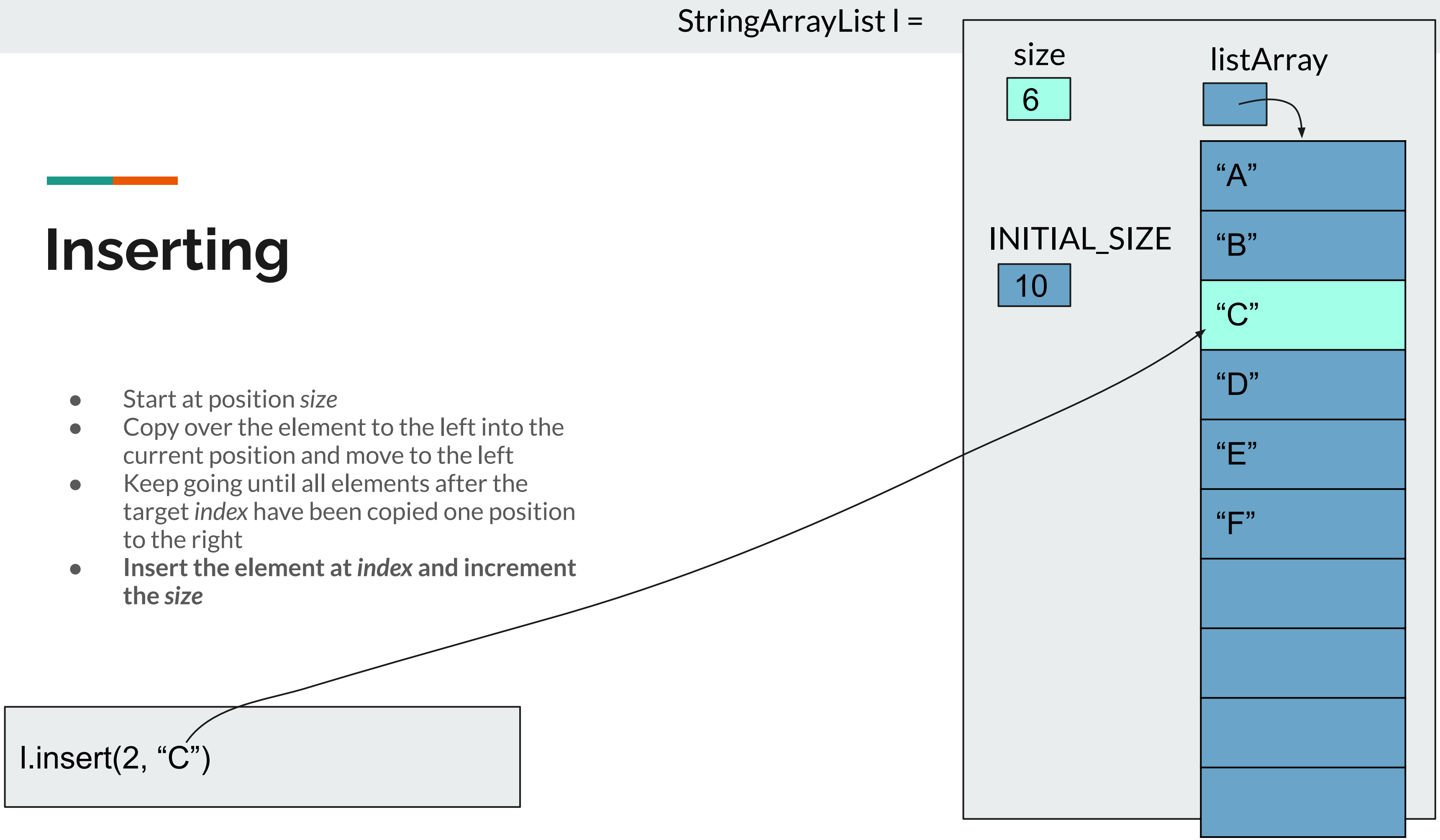
- Start at position *size*
- **Copy over the element to the left into the current position and move to the left**
- **Keep going until all elements after the target *index* have been copied one position to the right**
- Insert the element at *index* and increment the *size*

```
l.insert(2, “C”)
```



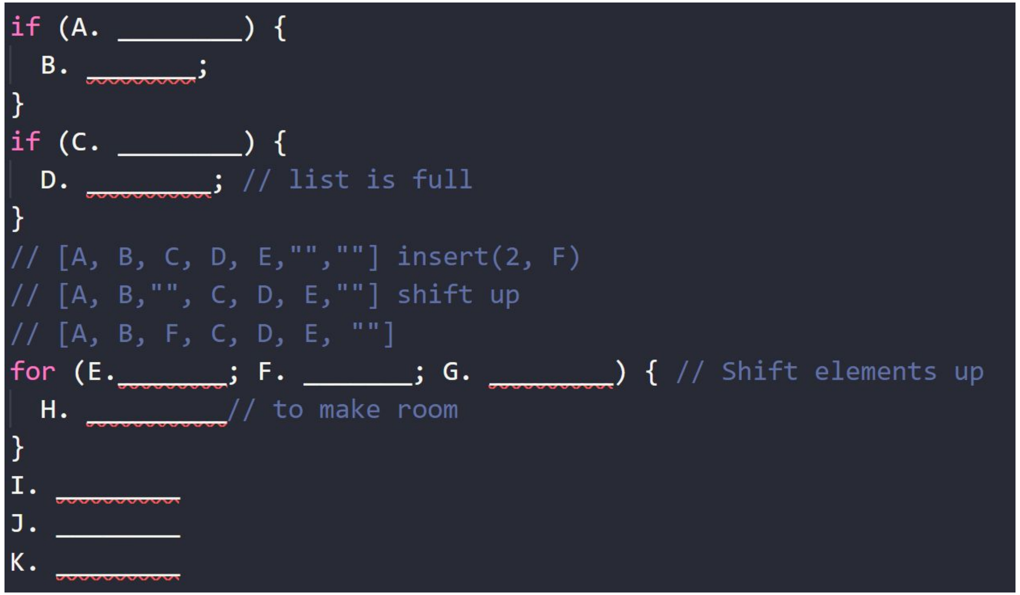
Inserting

- Start at position *size*
- Copy over the element to the left into the current position and move to the left
- Keep going until all elements after the target *index* have been copied one position to the right
- **Insert the element at *index* and increment the *size***

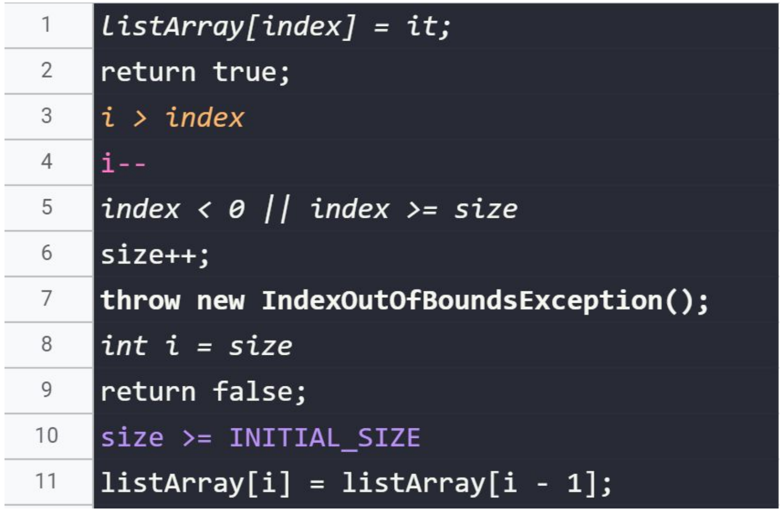


```
l.insert(2, "C")
```

Unscramble!

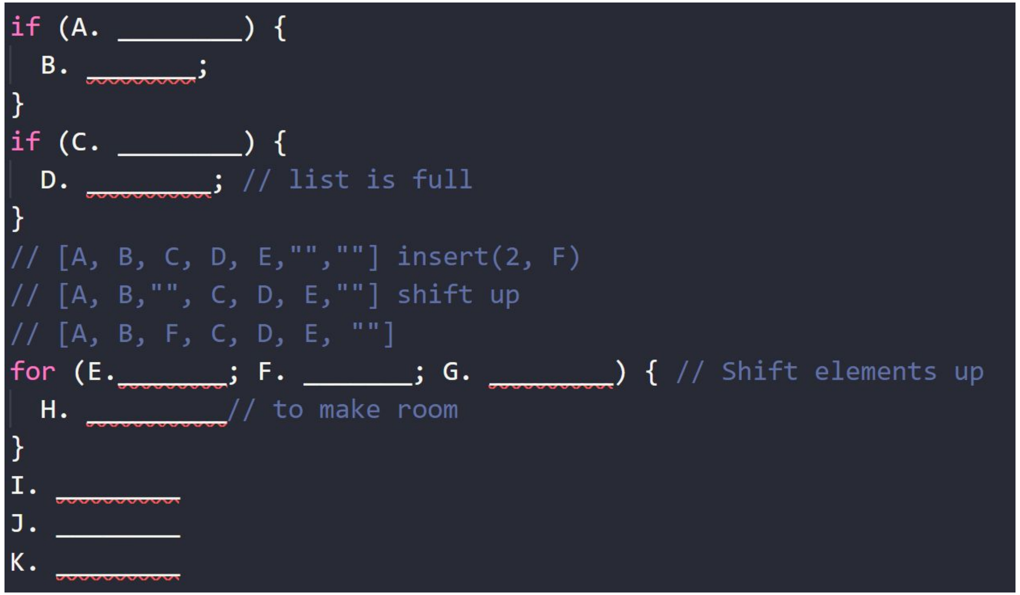


```
if (A. ________) {
  B. _______;
}
if (C. ________) {
  D. ________; // list is full
}
// [A, B, C, D, E,"",""] insert(2, F)
// [A, B,"", C, D, E,""] shift up
// [A, B, F, C, D, E, ""]
for (E._______; F. _______; G. ________) { // Shift elements up
  H. _________// to make room
}
I. ________
J. ________
K. ________
```

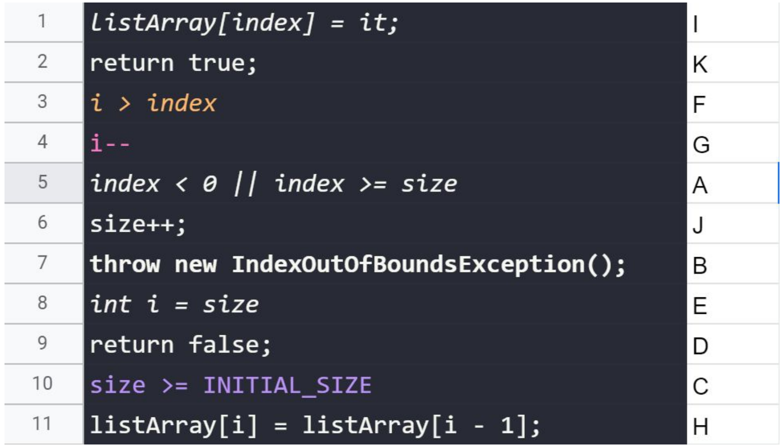


1	`listArray[index] = it;`
2	`return true;`
3	`i > index`
4	`i--`
5	`index < 0 || index >= size`
6	`size++;`
7	`throw new IndexOutOfBoundsException();`
8	`int i = size`
9	`return false;`
10	`size >= INITIAL_SIZE`
11	`listArray[i] = listArray[i - 1];`

Unscramble!



```
if (A. ________) {
  B. _______;
}
if (C. ________) {
  D. ________; // list is full
}
// [A, B, C, D, E,"",""] insert(2, F)
// [A, B,"", C, D, E,""] shift up
// [A, B, F, C, D, E, ""]
for (E._______; F. _______; G. ________) { // Shift elements up
  H. _________// to make room
}
I. ________
J. ________
K. ________
```

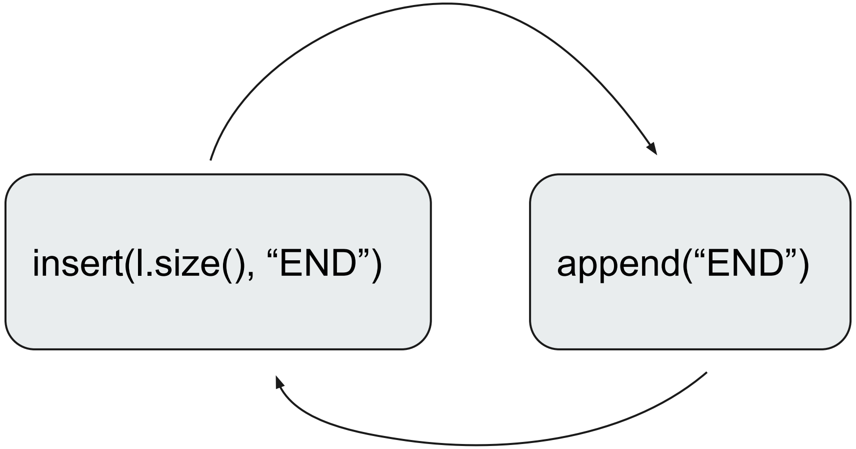


1	listArray[index] = it;	I
2	return true;	K
3	i > index	F
4	i--	G
5	index < 0 || index >= size	A
6	size++;	J
7	throw new IndexOutOfBoundsException();	B
8	int i = size	E
9	return false;	D
10	size >= INITIAL_SIZE	C
11	listArray[i] = listArray[i - 1];	H

Append

Like with the LinkedList, appending is just inserting at the end.

Let’s just use that instead.

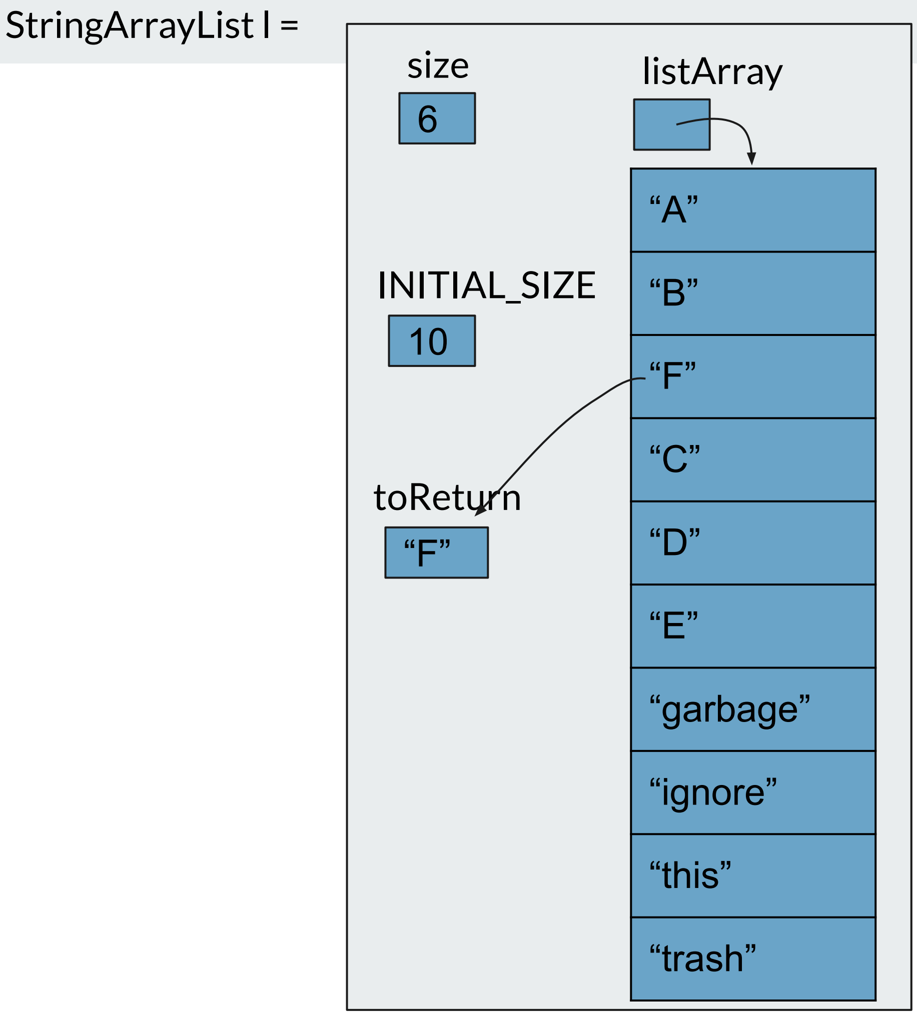


Remove

Removing

- **Do error checking for valid index; throw exception if *index* is invalid**
- **Copy the element at the index to be removed**
- Start at the removal *index*
- Copy the element at *index + 1* into the position *index*.
- Increment *index* and repeat until we've copied the last element in the List (lives at size - 1)
- Decrement *size* and return the removed element

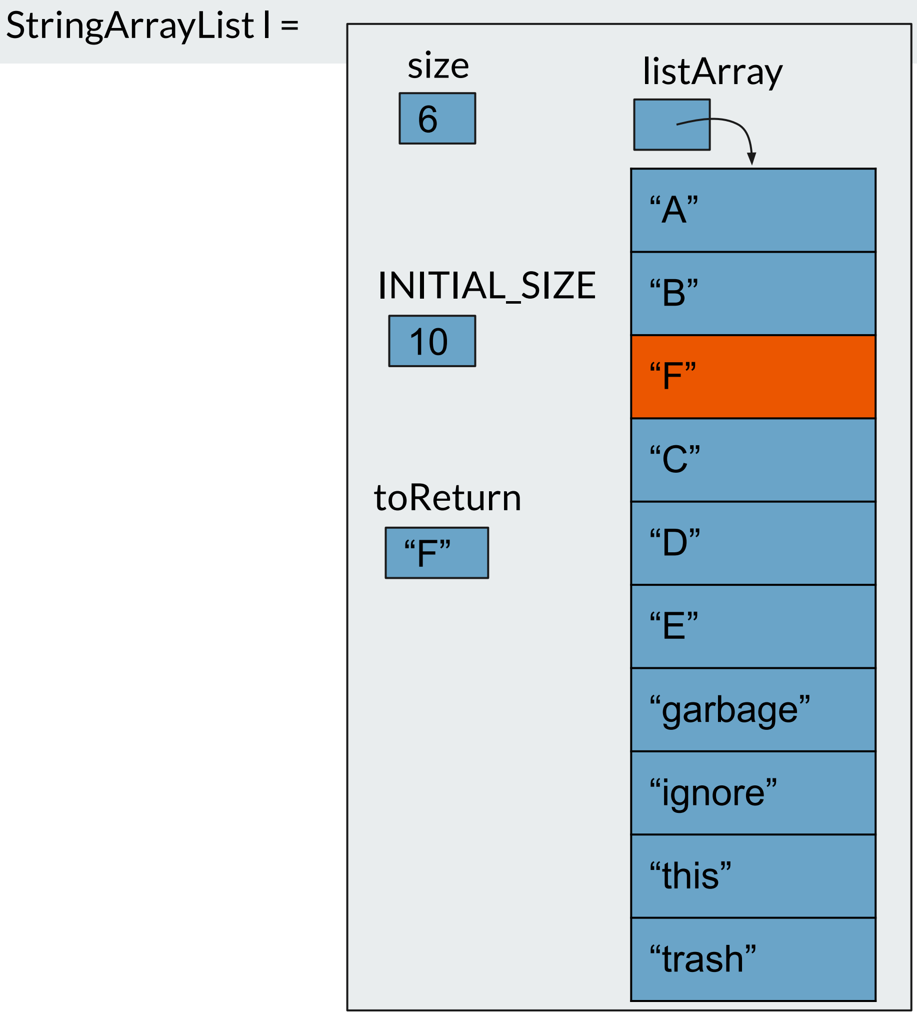
```
l.remove(2)
```



Removing

- Do error checking for valid index; throw exception if *index* is invalid
- Copy the element at the index to be removed
- **Start at the removal *index***
- Copy the element at *index + 1* into the position *index*.
- Increment *index* and repeat until we've copied the last element in the List (lives at size - 1)
- Decrement *size* and return the removed element

```
l.remove(2)
```

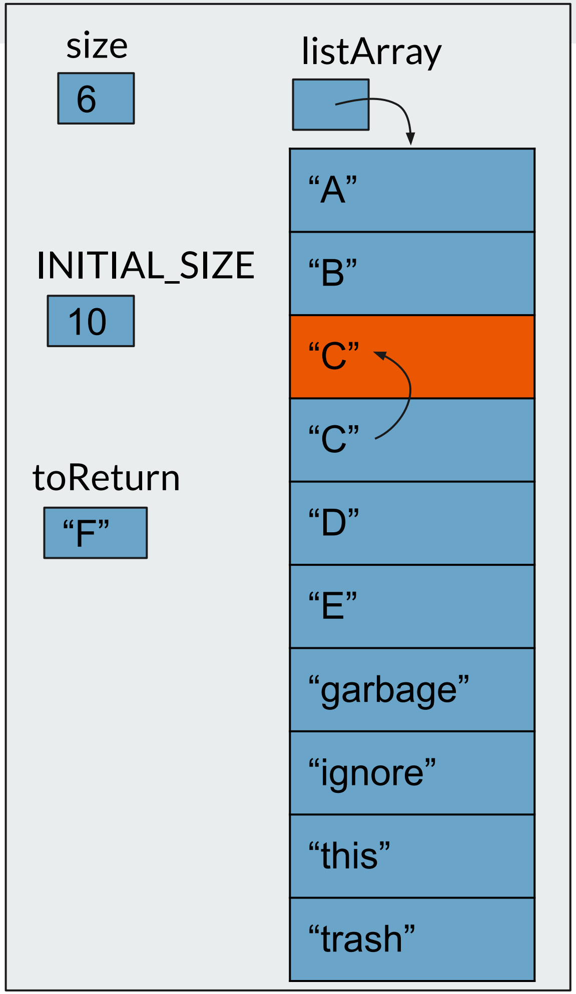


Removing

- Do error checking for valid index; throw exception if *index* is invalid
- Copy the element at the index to be removed
- **Start at the removal *index***
- **Copy the element at *index + 1* into the position *index*.**
- Increment *index* and repeat until we've copied the last element in the List (lives at size - 1)
- Decrement *size* and return the removed element

```
l.remove(2)
```

StringArrayList l =

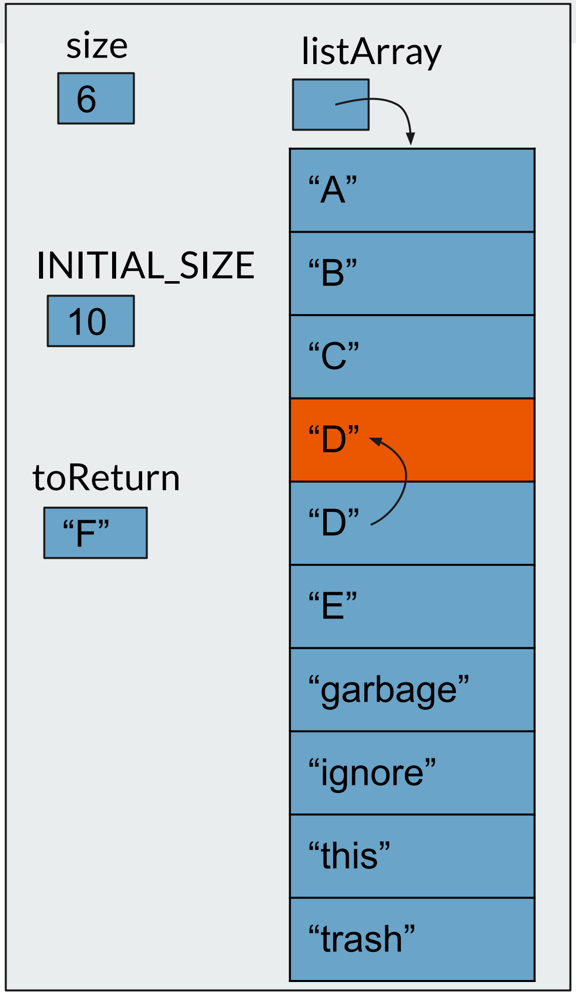


Removing

- Do error checking for valid index; throw exception if *index* is invalid
- Copy the element at the index to be removed
- **Start at the removal *index***
- **Copy the element at *index + 1* into the position *index*.**
- **Increment *index* and repeat until we've copied the last element in the List (lives at size - 1)**
- Decrement *size* and return the removed element

```
l.remove(2)
```

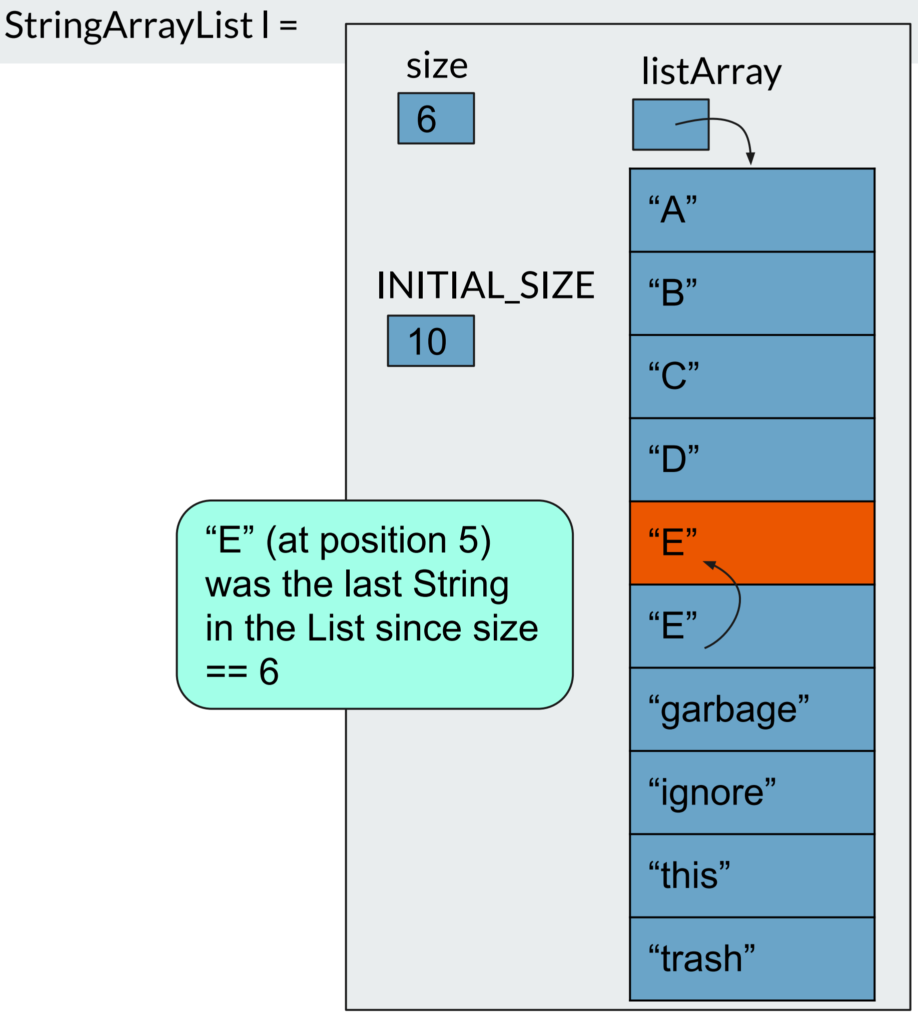
StringArrayList l =



Removing

- Do error checking for valid index; throw exception if *index* is invalid
- Copy the element at the index to be removed
- **Start at the removal *index***
- **Copy the element at *index + 1* into the position *index*.**
- **Increment *index* and repeat until we've copied the last element in the List (lives at size - 1)**
- Decrement *size* and return the removed element

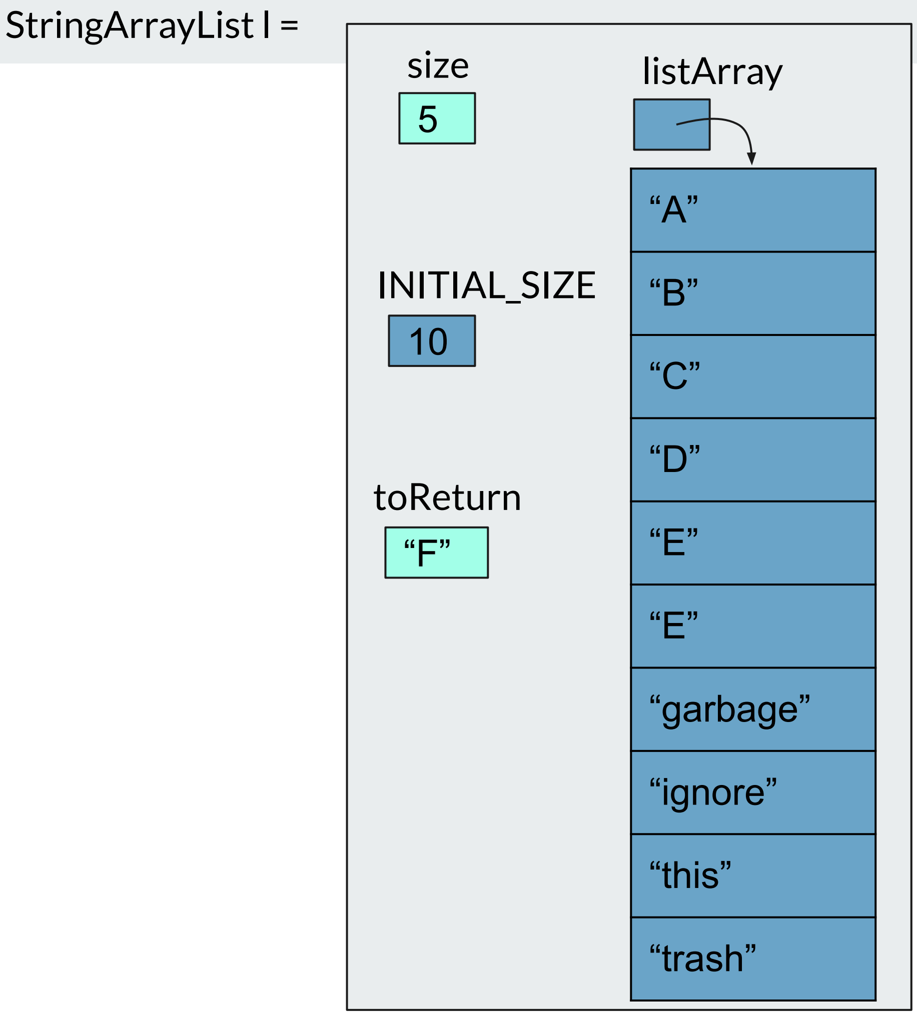
l.remove(2)



Removing

- Do error checking for valid index; throw exception if *index* is invalid
- Copy the element at the index to be removed
- Start at the removal *index*
- Copy the element at *index + 1* into the position *index*.
- Increment *index* and repeat until we've copied the last element in the List (lives at size - 1)
- **Decrement *size* and return the removed element**

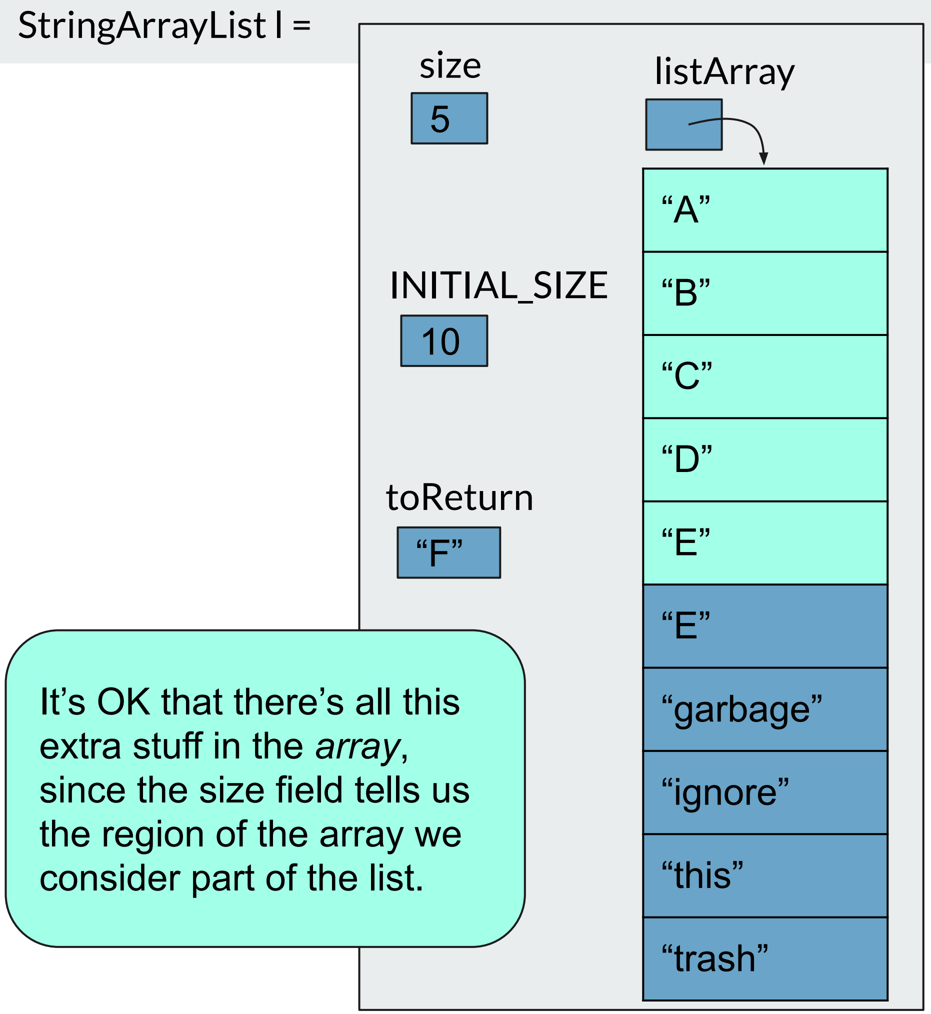
```
l.remove(2)
```



Removing

- Do error checking for valid index; throw exception if *index* is invalid
- Copy the element at the index to be removed
- Start at the removal *index*
- Copy the element at *index + 1* into the position *index*.
- Increment *index* and repeat until we've copied the last element in the List (lives at size - 1)
- **Decrement *size* and return the removed element**

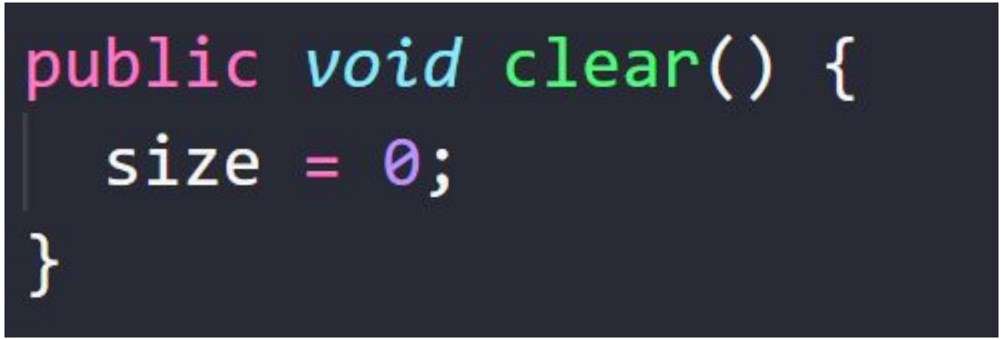
```
l.remove(2)
```



It's OK that there's all this extra stuff in the *array*, since the size field tells us the region of the array we consider part of the list.

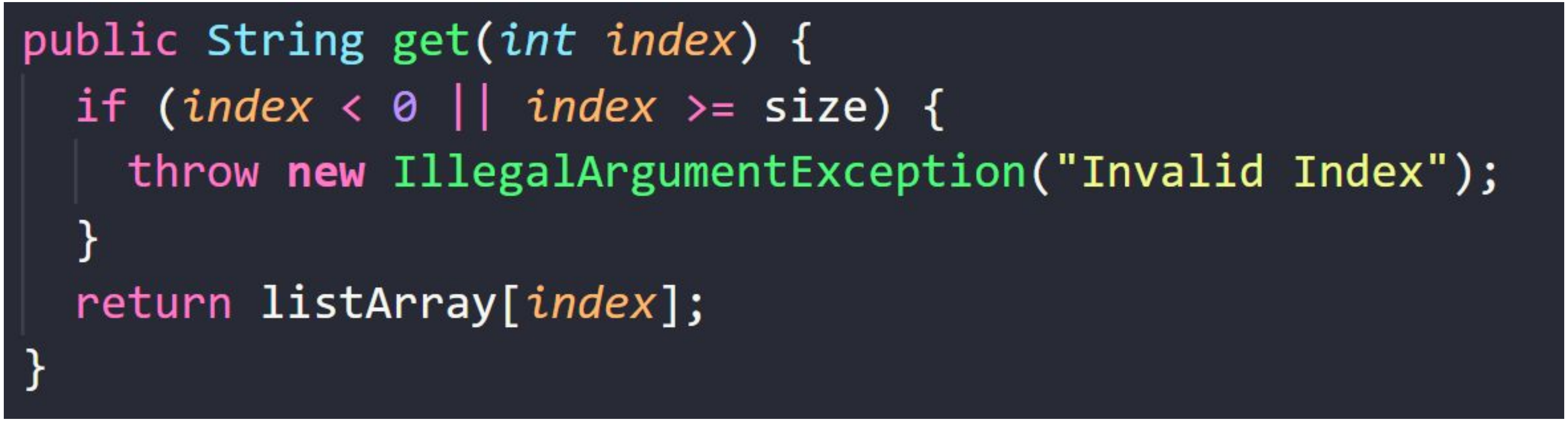
The Rest

We've done the hard part. The rest is easy because of the underlying array.



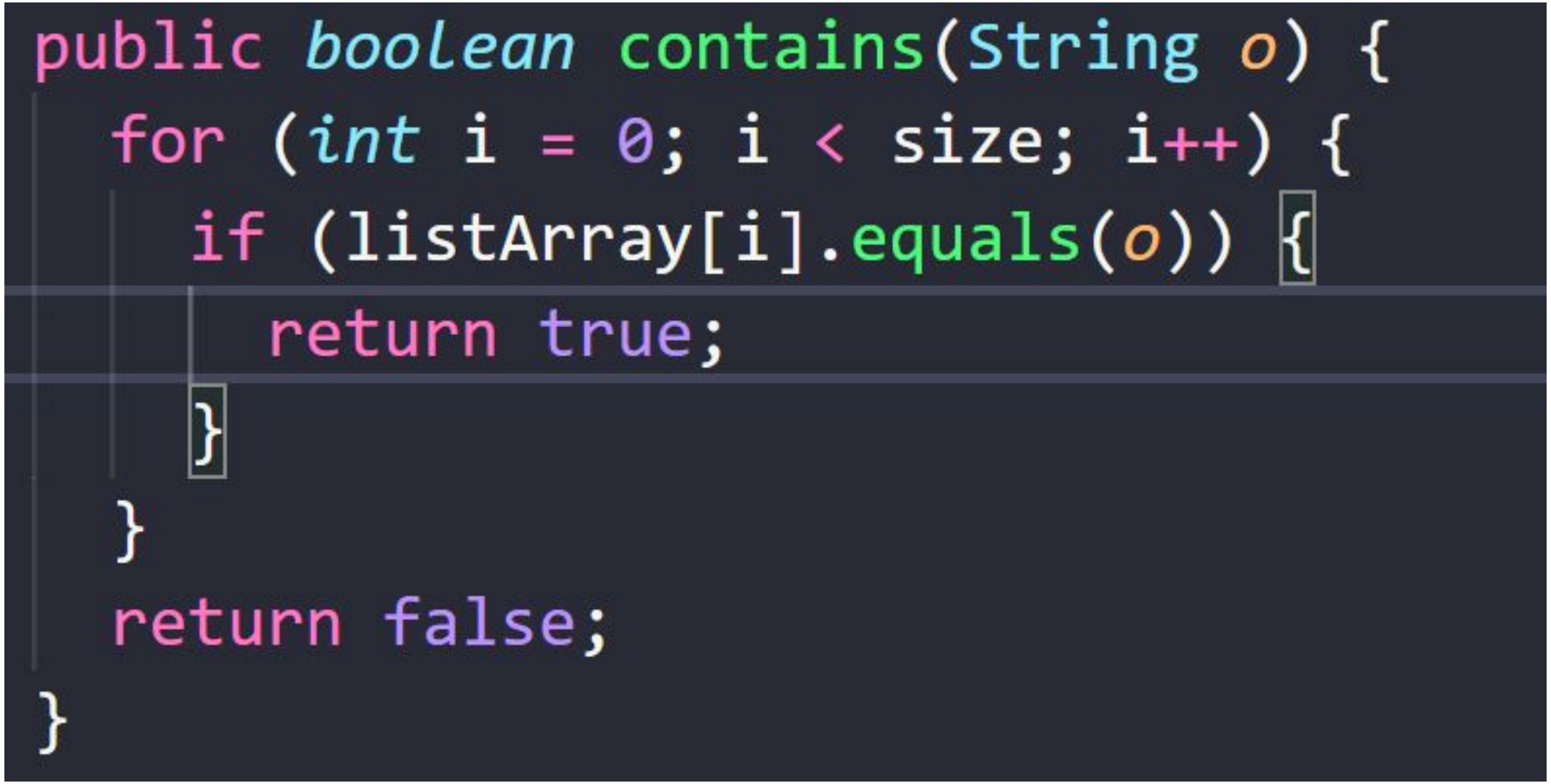
```java
public void clear() {
  size = 0;
}
```

clear

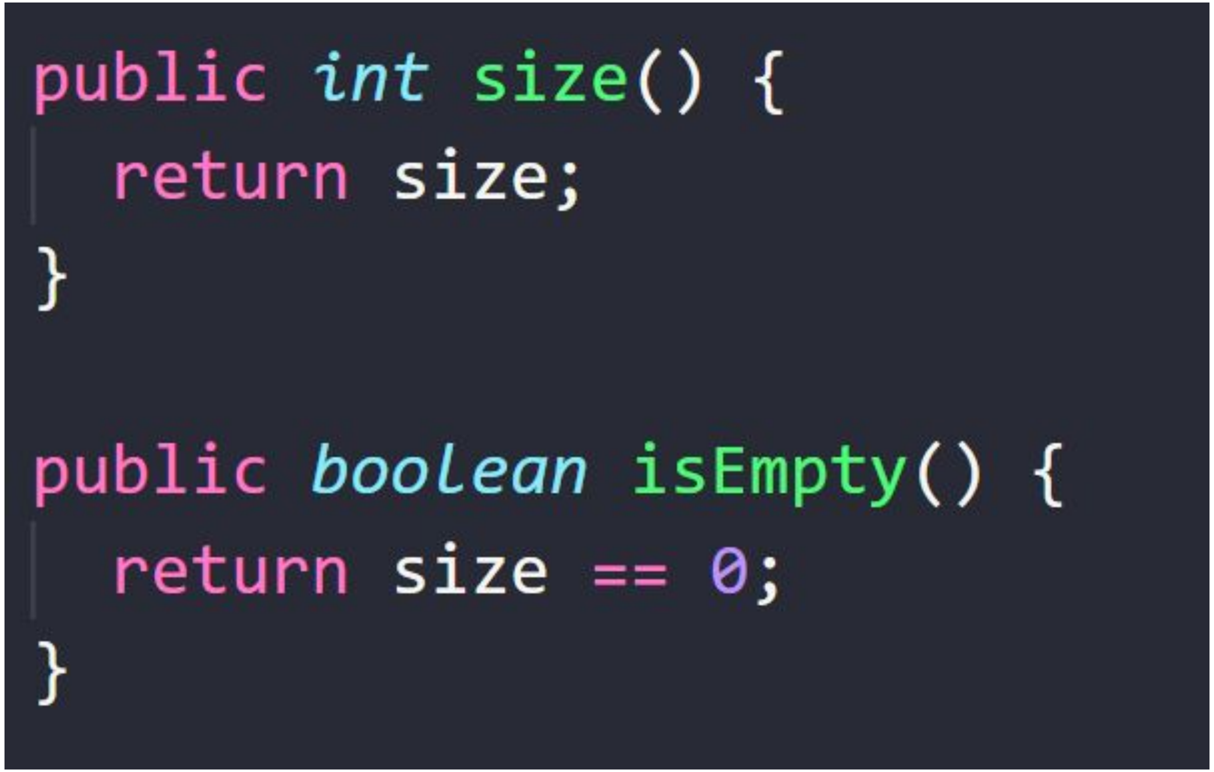


```java
public String get(int index) {
  if (index < 0 || index >= size) {
    throw new IllegalArgumentException("Invalid Index");
  }
  return listArray[index];
}
```

get


```java
public boolean contains(String o) {
    for (int i = 0; i < size; i++) {
        if (listArray[i].equals(o)) {
            return true;
        }
    }
    return false;
}
```

contains



```java
public int size() {
  return size;
}

public boolean isEmpty() {
  return size == 0;
}
```

size & isEmpty