PA1 → tonight @ 10pm
wed → late / resubmits
→ due June 3 @ 10pm
→ form

PA2 → tomorrow

```
public interface StringList {
  /* Add an element at the end of the list */
  void add(String s);

  /* Get the element at the given index */
  String get(int index);

  /* Get the number of elements in the list */
  int size();

  /* Add an element at the specified index */
  void insert(int index, String s);

  /* Remove the element at the specified index */
  void remove(int index);
}
```

During the pre-lecture recording, why were the insert and remove methods commented out?

we didn't want to write the methods (yet)

interface → requires bodies for all methods when implemented

```
public class ArrayStringList implements StringList {

  String[] elements;
  int size;

  ...

  private void expandCapacity() {
    int currentCapacity = this.elements.length;
    if(this.size < currentCapacity) { return; }

    String[] expanded = new String[currentCapacity * 2];

    for(int i = 0; i < this.size; i += 1) {
      expanded[i] = this.elements[i];
    }

    this.elements = expanded;
  }

  public void foo(){
    String[] tmp = elements;
    add("a"); add("b"); add("c");
    expandCapacity();
    System.out.println(tmp == elements);
  }
}
```

What's the point of having *size* as a field (member variable) as the array *elements* already has size?

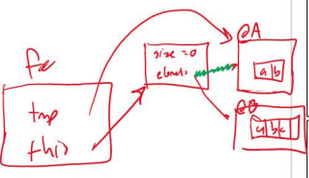
length of array → capacity

size → # of elements added to the data structure

When do we need to call this *expandCapacity* function?

add()

insert()



size = 0 1 2 3

"a" "b"

"a" "b" "c" "bm"

If this *foo* method is called, what will be printed out? Assume that the array starts empty and has a capacity of 2.

false

@A @B

```
public class TestStringList {
  @Test
  public void testAdd() {
    StringList slist = new ArrayStringList();
    slist.add("banana");
    slist.add("apple");

    assertEquals("banana", slist.get(0));
    assertEquals("apple", slist.get(1));
  }
}
```

Can we write the tester as?

```
assertEquals(slist.get(0), "banana");
assertEquals(slist.get(1), "apple");
```

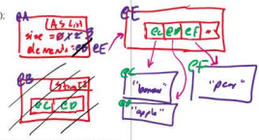
yes, but switches expected & actual values in jUnit

expected, actual

```
public class ArrayStringList implements StringList {
   ...

  /* Add an element at the end of the list */
  public void add(String s); {
    expandCapacity();
    this.elements[this.size] = s;
    this.size += 1;
  }

  /* Add an element at the specified index */
  public void insert(int index, String s) {
    expandCapacity();
    for (i = size -1; i >= index; i--) {
      this.elements[i+1] = this.elements[i];
    }
    this.elements[index] = s;
  }   this.size += 1;

  /* Remove the element at the specified index */
  public void remove(int index) {
    for (int i = index; i < size-1; i++) {
      this.elements[i] = this.elements[i+1];
    }
    this.elements[size-1] = null;
  }   this.size -= 1;
}
```

Assuming a fully implemented ArrayStringList class, draw the memory diagram (and array contents) for the following method calls:

```
add("banana");
add("apple");
add("pear");
```



Assuming a fully implemented ArrayStringList class, draw the memory diagram (and array contents) for the following method calls:

```
insert(0, "a");
insert(0, "b");
insert(0, "c");
insert(1, "z");
```

prepend

size = 0 1 2 3 4

4 "c" "b" "z" "a"

Write a test case for the ArrayList insert method:

```
ArrayStringList s = new ...
s.insert(0, "a");
assertEquals("a", s.elements[0]);
s.insert(0,"b");
assertEquals("b", s.elements[0]);
assertEquals("a", s.elements[1])
```

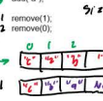
Implement the ArrayList insert method.

Use old Array Equals

Assuming a fully implemented ArrayStringList class, draw the memory diagram (and array contents) for the following method calls:

```
add("c");
add("z");
add("b");
add("a");

remove(1);
remove(0);
```

size = 4 3 2



Write a test case for the ArrayList remove method:

```
s.add("a")
s.add("b")
s.remove(0)
assertEquals("b", s.elements[0])
```

Implement the ArrayList remove method.

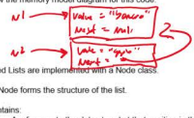
What is a Linked List?

A Linked List is a data structure that implements a List ADT, where elements in the list may appear anywhere in memory, but are "linked" together in a particular order using references or pointers.

```
class Node {
  String value;
  Node next;
  public Node(String value, Node next) {
    this.value = value;
    this.next = next;
  }
}

// Somewhere else in the code... still inside Node class (can access next)
Node n1 = new Node("banana", null);
Node n2 = new Node("apple", null);
n2.next = n1;
```

value | next

Draw the memory model diagram for this code.



Linked Lists are implemented with a Node class.

The Node forms the structure of the list.

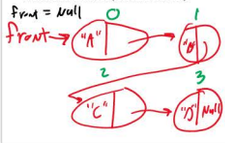
It contains:

- A reference to the data stored at that position in the list
- A reference to the next node in the list
- Optionally (for a doubly linked list) a reference to the previous node in the list.

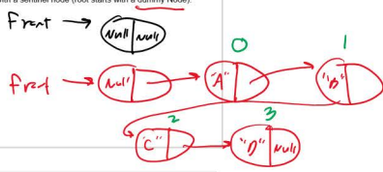
The Linked List itself usually contains only a reference to the first node in the list (head), and sometimes a reference to the last node (tail). It also might store the list's size.

Assume that a linked list contains the following elements in this order: "A", "B", "C", "D". Draw the memory model of the linked list. Label the index of each element.

Without a sentinel node (root starts with null).



With a sentinel node (root starts with a dummy Node):




```
public class LinkedStringList implements StringList {
  Node front;
  int size;

  public LinkedStringList() {
    this.front = new Node(null, null);
  }

  /* Add an element at the beginning of the list */
  public void prepend(String s) {
    Node newFront = new Node(s, this.front.next);
    this.front.next = newFront;
    this.size += 1;
  }

  /* Add an element at the end of the list */
  public void add(String s) {









  }

  /* Add an element at the specified index */
  public void insert(int index, String s) {









  }

  /* Remove the element at the specified index */
  public void remove(int index) {









  }
}
```

Draw the linked list for the following method calls:

```
prepend("banana");
prepend("apple");
prepend("pear");
```

Draw the linked list for the following method calls:

```
add("banana");
add("apple");
add("pear");
```

Draw the linked list for the following method calls:

```
insert(0, "a");
insert(0, "b");
insert(0, "c");
insert(1, "z");
```

Draw the linked list for the following method calls:

```
add("c");
add("z");
add("b");
add("a");

remove(1);
remove(0);
```