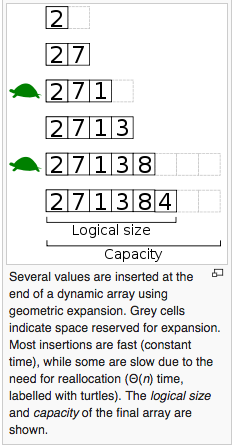
1. ***Recap on ArrayList:*** 

The ArrayList class implements the List interface. ArrayList supports dynamic arrays that can grow as needed.

Standard Java arrays are of a fixed length. After arrays are created, they cannot grow or shrink, which means that you must know in advance how many elements an array will hold.

Array lists are created with an initial size. When this size is exceeded, the collection is automatically enlarged. When objects are removed, the array may be shrunk.

***constructors:***

1. **ArrayList()**

Builds an array list with default capacity.

1. **ArrayList(int capacity)**

This constructor builds an array list has the specified initial capacity. The capacity is the size of the underlying array that is used to store the elements. The capacity grows automatically as elements are added to an array list.

***Some Methods:***

1. **void add(int index, Object element)**

Inserts the specified element at the specified position index in this list. Throws IndexOutOfBoundsException if the specified index is out of range (index < 0 || index > size()).

**2. void add(Object o)**

Appends the specified element to the end of this list.

**3. void clear()**

Removes all of the elements from this list.

**4. T replace(int position, Object o)**

Replace an object in a specific position.

**5. boolean contains(Object o)**

Returns true if this list contains the specified element. More formally, returns true if and only if this list contains at least one element **e** such that (o==null ? e==null : o.equals(e)).

**6. Object get(int index)**

Returns the element at the specified position in this list. Throws IndexOutOfBoundsException if the specified index is out of range (index < 0 || index >= size()).

**7. int indexOf(Object o)**

Returns the index in this list of the first occurrence of the specified element, or -1 if the List does not contain this element.

**8. Object remove(int index)**

Removes the element at the specified position in this list. Throws IndexOutOfBoundsException if the index out is of range (index < 0 || index >= size()).

**9. int getSize()**

Returns the number of elements in this list.

**10. Object[] toArray()**

Returns an array containing all of the elements in this list in the correct order. Throws NullPointerException if the specified array is null.

**11. Void reverse()**

Reverse all the elements in the list.

But in ArrayList.java, the implementation uses O(n) additional memory. You should reimplement later in the exercise session.

Example:

public class ArrayListDemo {  
  
 public static void main(String args[]) {  
 // create an array list  
 ArrayList al = new ArrayList(); //size is 10, using code in this recitation  
 System.out.println("Initial size of al: " + al.size());  
  
 // add elements to the array list  
 al.add("C");  
 al.add("A");  
 al.add("E");  
 al.add("B");  
 al.add("D");  
 al.add("F");  
 al.add(1, "A2");  
 System.out.println("Size of al after additions: " + al.size());  
  
 // display the array list  
 System.out.println("Contents of al: " + al);  
  
 // Remove elements from the array list  
 al.remove("F");  
 al.remove(2);  
 System.out.println("Size of al after deletions: " + al.size());  
 System.out.println("Contents of al: " + al);  
 }  
}

***II recap on LinkedList***

In [computer science](https://en.wikipedia.org/wiki/Computer_science), a **linked list** is a linear collection of data elements, called nodes, each pointing to the next node by means of a [pointer](https://en.wikipedia.org/wiki/Pointer_(computer_programming)). It is a [data structure](https://en.wikipedia.org/wiki/Data_structure) consisting of a group of [nodes](https://en.wikipedia.org/wiki/Node_(computer_science)) which together represent a [sequence](https://en.wikipedia.org/wiki/Sequence). Under the simplest form, each node is composed of data and a [reference](https://en.wikipedia.org/wiki/Reference_(computer_science)) (in other words, a *link*) to the next node in the sequence. This structure allows for efficient insertion or removal of elements from any position in the sequence during iteration. More complex variants add additional links, allowing efficient insertion or removal from arbitrary element references.

Screen Shot 2016-11-15 at 11.28.05 AM.png

***constructors:***

1. **LinkedList( )**

This constructor builds an empty linked list.

**Some methods:**

1. **boolean add(Object o)**

Appends the specified element to the end of this list.

1. **void add(int index, Object element)**

Inserts the specified element at the specified position index in this list. Throws IndexOutOfBoundsException if the specified index is out of range (index < 0 || index > size()).

1. **Object remove(int index)**

Removes the element at the specified position in this list. Throws NoSuchElementException if this list is empty.

1. **Object replace(int position, Object O)**

replace the element at the specified position in this list. Throws IndexOutOfBoundsException if this list is empty.

1. **Node getNodeAt(int position)**

This returns the node which points to the next object in the list.

1. **void reverse()**

In LinkedList.java, the implementation uses getNodeAt() at each step, and is therefore O(n^2). You should reimplement this method in O(n) time, without using getNodeAt() at each step (Hint: build backwards).

Exercise:

1. In ArrayList.java, The reverse() method implementation uses O(n) additional memory. You should reimplement this method in-place: without using O(n) extra memory (Hint: swaps). Be sure to test with both even and odd lengths!

2. In LinkedList.java, the provided code for reverse() is O(n^2) because it uses getNodeAt() for each iteration. Reimplement a version that is O(n). (Hint: build backwards).

A ReverseTest.java has been provided for them to demonstrate the behavior.