Programming with Data Structures

CMPSCI 187
Spring 2016

- **Please find a seat**
  - **Try to sit close to the center (the room will be pretty full!)**
- **Turn off or silence your mobile phone**
- **Turn off your other internet-enabled devices**

Abstract and Array-Based Lists

- Comparing Objects
- The List ADT and Interfaces
- Iterators
- Array-Based Lists

Comparing Objects in Java

- We now begin our study of **lists** — general types of collections of objects.
- One basic operation for lists is testing whether a given element is already in a list (the `contains` method). Some of our lists will be **sorted**, which means we need to define what it means for one element to be equal to, smaller than, or larger than another.
- So how do we compare two objects?

Comparing Objects in Java

- We've learned that for primitive data types (such as `int`, `float`), the statement `(x==y)` is true if and only if `x` and `y` contain the same value.

- For objects, `(x==y)` is true if and only if they reference the same object (note that this does NOT contradict with the above because an object type variable holds the pointer to the object).

- If we want to compare the content of two objects, we need to define a custom `equals()` method.

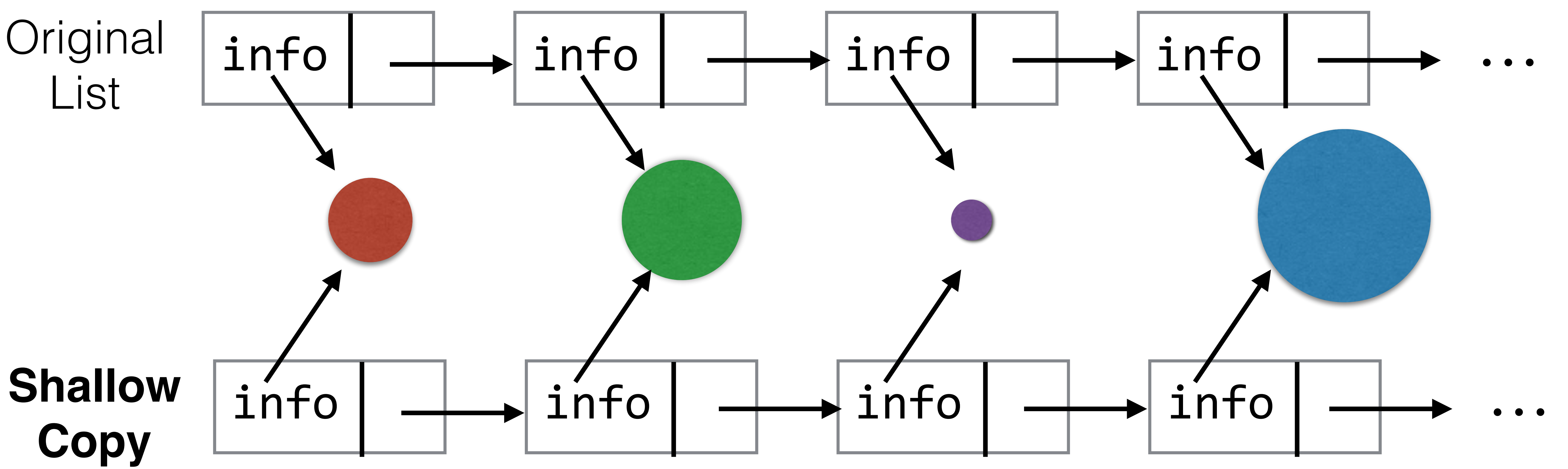
Comparing Objects in Java

- For example, Java's `String` class has a custom `equals` method which tests if two strings contain the same sequence of letters (this is intuitively what we mean by testing if two strings are equal).
- Another example:

```
public class Circle {
  protected float radius;
  public Circle(float r) {this.radius = r;}
  public boolean equals(Circle c) {
    return (this.radius == c.radius);
  }
}
```

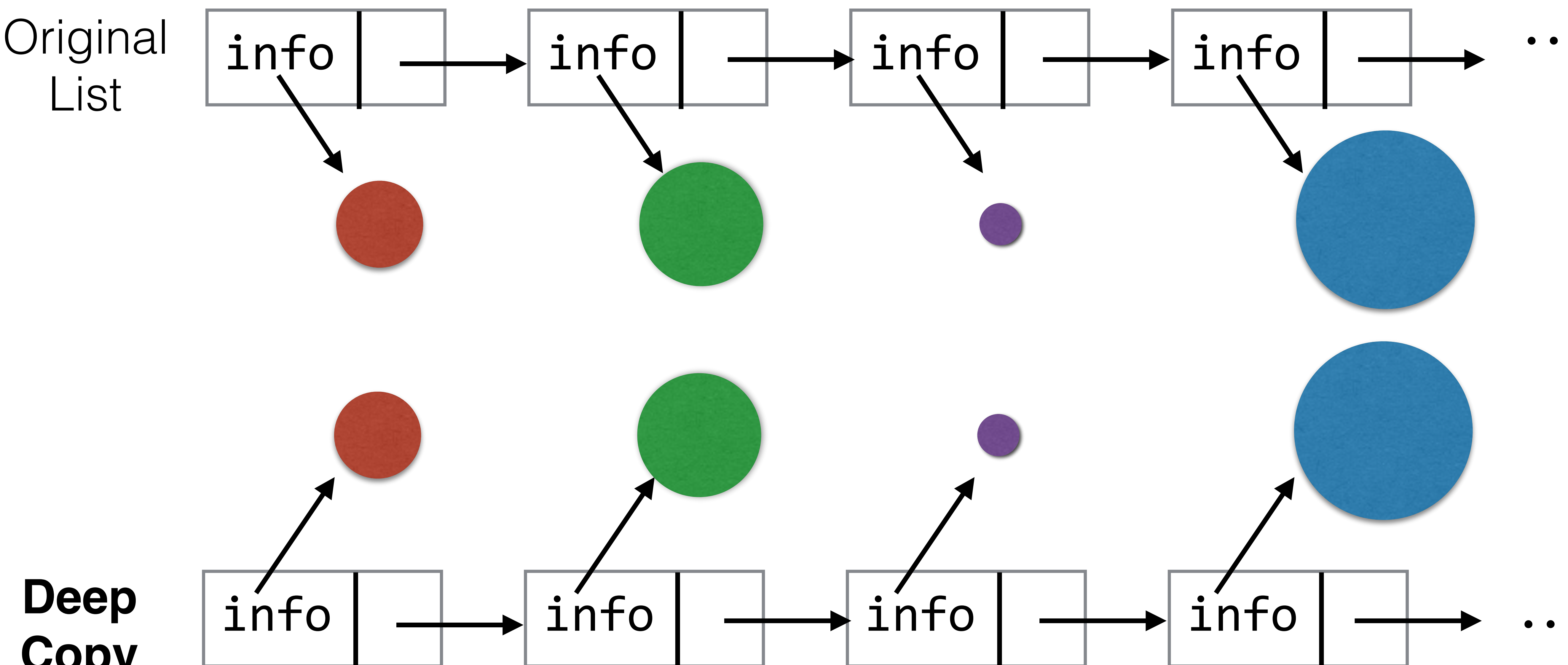
Shallow Copying

- Project 6 briefly describes the concept of Shallow Copying, which means when you copy a data element, you merely copy its reference, without cloning its content (i.e. without allocating new memory).



Deep Copying

- In contrast, deep copying means for each data element, you will need to make a clone (i.e. **allocate new memory**) and copy the content over.



Comparing Objects by Order

- In addition to check for equality, in order to sort objects, we also need to define what it means for one object to be smaller or larger than another.

- For example, to compare the following strings in alphabetic order:
  - `hat > cat`
  - `cats > cat`
  - `cats < hat`
  - `computation < computer`

Comparing Objects by Order

- Java provides a generic `Comparable<T>` interface that allows you to define custom comparison methods. Specifically, the interface requires that a class to implement

  ```
  public int compareTo (T other)
  ```

  which returns:

  - a negative number (e.g. `-1`) if ‘this’ object is smaller than the other object.
  - `0` if they are equal
  - a positive number (e.g. `1`) is ‘this’ object is larger.

Comparing Objects by Order

- Example of compareTo

```
Integer i = new Integer(10);
Integer j = new Integer(100);

System.out.println(i.compareTo(j));  // -1

System.out.println(j.compareTo(i));  //  1

System.out.println(i.compareTo(10)); //  0
```

Comparing Objects by Order

- Example of compareTo

```
public class Circle
             implements Comparable<Circle> {
  protected float radius;
  public Circle(float r) {this.radius = r;}
  public int compareTo(Circle c) {
    if (this.radius == c.radius) return 0;
    return (this.radius > c.radius) ? 1 : -1;
  }
}
```

Clicker question #1

```
public class Point implements Comparable<Point>{
  protected int x, y;
  public Point(int x, int y) {
    this.x = x; this.y = y;
  }
  public int compareTo(Point p) {
    return (this.x + this.y) - (p.x + p.y);
  }
}
```

What is the result of calling
`(new Point(3, 2)).compareTo(new Point(1, 5))`?

a) -1
b) 1
c) -2
d) 2
e) 0

answer on next slide

Clicker question #1

```
public class Point implements Comparable<Point>{
  protected int x, y;
  public Point(int x, int y) {
    this.x = x; this.y = y;
  }
  public int compareTo(Point p) {
    return (this.x + this.y) - (p.x + p.y);
  }
}
```

What is the result of calling
`(new Point(3, 2)).compareTo(new Point(1, 5))`?

a) -1
b) 1
c) -2
d) 2
e) 0

Clicker question #2

```
LLNode<String> n1 = new LLNode<String>();
n1.setInfo(new String("CS187"));
LLNode<String> n2 = new LLNode<String>();
n2.setInfo(n1.getInfo());
System.out.println(n1==n2);
System.out.println(n1.getInfo()==n2.getInfo());
System.out.println(n1.getInfo().equals(n2.getInfo()));
```

What's the output?

a) true true true
b) false true true
c) false false true
d) false false false

answer on next slide

Clicker question #2

```
LLNode<String> n1 = new LLNode<String>();
n1.setInfo(new String("CS187"));
LLNode<String> n2 = new LLNode<String>();
n2.setInfo(n1.getInfo());
System.out.println(n1==n2);
System.out.println(n1.getInfo()==n2.getInfo());
System.out.println(n1.getInfo().equals(n2.getInfo()));
```

What's the output?

a) `true  true  true`

b) `false true  true`

c) `false false true`

d) `false false false`

Lists: Unsorted, Sorted, Indexed

- A list is a **linear** data structure, where each element except the last has a **successor** and each element except the first has a **predecessor**. Every list also has a **size** — the number of elements in it.

- A list is **sorted** if the successor and predecessor properties are **consistent** with the `compareTo` method of the elements -- each element is “less than or equal to” its successor according to that method.

Lists: Unsorted, Sorted, Indexed

- A list without this property is **unsorted** -- it still has an order given by the successor and predecessor properties, but that order has no meaning in terms of the elements themselves.

- A list can also be **indexed**, meaning that we can access elements directly by their position in the list, or **index**. In an indexed list, we would have methods to "return the fourth element" in the order given by successor and predecessor.

Assumptions About Lists

- We make the following assumptions (design decisions):

- Lists are **unbounded** -- if implemented with arrays, the arrays can expand dynamically.

- **Duplicate elements** (where one equals the other) *are* allowed. Finding one equal element is as good as finding any other.

- We do not support `null` elements. For example, when calling the list's `add` and `remove` methods, you cannot pass a `null` value as argument.

Assumptions About Lists

- Operations generally report success or failure by returning a boolean value, not by throwing an exception on failure (except for a bad index in an indexed list).

- Sorted lists are in **increasing** (more precisely, **non-decreasing**) order. Indexed lists have indices ranging from `0` to the `size - 1`, with no gaps.

- The `equals` and `compareTo` methods are **consistent** with one another.

The List Interfaces

- While sorted and unsorted lists differ in some respects, the names of their operations are the same and thus the same interface may be used for both.

- We often want to **iterate** through a list, processing each element in turn. We have made the interface extend `Iterable` so any classes that implement this interface must provide `iterator()`.

- Note that we have changed the interface in DJW as it doesn't implement Iterable (bad).

The Parent List Interface

```
public interface ListInterface<T> extends Iterable<T>
{
    void add(T element);
    int size();
    boolean contains (T element);
    // returns true if this list contains an element e
    // such that e.equals(element) is true
    boolean remove (T element);
    // returns true if removed successfully
    T get (T element);
    // returns null if element does not exist
    String toString();
}
```

The Indexed List Interface

- In an indexed list, we have additional methods to **`add`**, **`get`**, **`set`**, or **`remove`** elements at a particular position (index). If the index is outside the valid range (which is `[0,size]` for `add` and `[0,size-1]` for the other methods), we throw an `IndexOutOfBoundException`.
- The **`add`** / **`remove`** method inserts / deletes an element at a particular position, and move all higher-indexed elements to the right / left by one spot to make room for the new element or to fill in the gap.
- **`set`** replaces the element at a position

The Indexed List Interface

```
public interface IndexedListInterface<T>
                 extends ListInterface<T> {
    void add (int index, T element);
    // higher elements move up
    T set (int index, T element);
    // returns former value
    T get (int index);
    // exception for bad index
    int indexOf (T element);
    // index of first one, -1 if none
    T remove (int index);
    // higher elements move down to fill the gap
    Iterator<T> iterator(); // required
}
```

Clicker Question #3

- Let **`AIL`** be a class implementing **`IndexedListInterface<Dog>`**. What value is returned at the end of the following code fragment? Assume the dog objects all exist and are unique.

```
AIL x = new AIL();
x.add(0, cardie);
x.add(0, duncan);
x.add(1, whistle);
x.add(0, whistle);
x.set(2, whistle);
x.remove(0);
x.add(1, cardie);
x.remove(2);
return x.indexOf(whistle);
```

(a) -1 (b) 0 (c) 1 (d) 2

Answer on next slide

Clicker Question #3

- Let **`AIL`** be a class implementing **`IndexedListInterface<Dog>`**. What value is returned at the end of the following code fragment? Assume the dog objects all exist and are unique.

```
AIL x = new AIL();
x.add(0, cardie);
x.add(0, duncan);
x.add(1, whistle);
x.add(0, whistle);
x.set(2, whistle);
x.remove(0);
x.add(1, cardie);
x.remove(2);
return x.indexOf(whistle);
```

$\boxed{\text{(a) -1}}$ (b) 0 (c) 1 (d) 2

Now lets implement it an unsorted list with an array…

```java
public class ArrayUnsortedList<T>
             implements ListInterface<T> {
  protected final static int DEFCAP = 100;
  protected int origCap;
  protected T[] list;
  protected int numElements=0, currentPos, location;
  public ArrayUnsortedList(int origCap) {
    list = (T[]) new Object[origCap];
  }
  public ArrayUnsortedList( ) {
    this(DEFCAP);
    origCap = DEFCAP;
  }
```

Note that this is different from DJW (location+find are not used)

ArrayUnsortedList

```
protected void enlarge( ) {
    T[] larger = (T[]) new Object[list.length+origCap];
    for (int i = 0; i < numElements; i++)
        larger[i] = list[i];
    list = larger;
}
```

- The textbook's approach is to enlarge (expand) the array capacity by **`origCap`** each time.

- How does Java know to release the old array?

Clicker Question #4

- Assume a list has a capacity of 100 to begin with. Each time it expands, we increase its capacity by 100 (i.e. add 100 more spots and copy the elements over). Starting from an empty list, we add one element at a time, until there are N elements. **How many assignment (=) instructions would have been executed?**

`(a) O(1) (b) O(N)  (c) O(log N) (d) O(N^2)`

Answer on next slide

Clicker Question #4

- Assume a list has a capacity of 100 to begin with. Each time it expands, we increase its capacity by 100 (i.e. add 100 more spots and copy the elements over). Starting from an empty list, we add one element at a time, until there are N elements. **How many assignment (=) instructions would have been executed?**

`(a) O(1) (b) O(N)  (c) O(log N)` $\boxed{\texttt{(d) O(N\^{}2)}}$

ArrayUnsortedList - The Typical Way

```
protected void enlarge( ) {
    T[] larger = (T[]) new Object[list.length*2];
    for (int i = 0; i < numElements; i++)
        larger[i] = list[i];
    list = larger;
}
```

- Here you double the capacity when enlarging it, this makes the array grow faster and it’s a better approach than the textbook.

Methods of AUL

```
protected int find (T target) {
    int location = 0;

    while (location < numElements) {
        if (list[location].equals(target)) {
            return location;
        } else location++;
    }
    return -1;  // Not found
}
```

Note that this is different from DJW, which uses an instance variable to pass state, bad, bad, bad…

Methods of AUL

```
public void add (T element) {
    if (numElements == list.length) enlarge( );
    list[numElements] = element;
    numElements++;
}
public boolean remove (T element) {
    int location = find(element);
    if (location != -1) {
        list[location] = list[numElements-1];
        list[numElements-1] = null;
        numElements--;
        return true;
    }
    return false;
}
```

Think about what these two lines are doing.

We can also have a sorted list by overriding add

ArraySortedList

```
public class ArraySortedList<T> extends ArrayUnsortedList<T>
                                implements ListInterface<T> {
  public void add (T element) {
    T listElement;
    int location = 0;
    if (numElements == list.length)  enlarge( );
    while (location < numElements) {
      listElement = list[location];
      if (((Comparable<T>) listElement).compareTo(element) < 0)
        location++;
      else break;
    }
    for (int index = numElements; index > location; index--)
      list[index] = list[index - 1];
    list[location] = element;
    numElements++;
  }
```

Other Methods of ASL

- The **`add`** method has a bunch of casts because the compiler doesn't know that the type **`T`** had better implement **`Comparable<T>`**.

- The **`find`**, **`contains`**, **`get`**, **`toString`**, **`iterator`** methods are all inherited from **`ArrayUnsortedList`**. They each run in O(N) time.

- The **`add`** and **`remove`** methods run in O(N) time on a list with N elements in the worst case, because of elements being moved to make room or fill a gap.