Abstract Lists

Clicker question #1

```
This is a good bit of code:

if (x == true){
  Do something
}
```

a) TRUE

b) FALSE

The List ADT and Comparisons

- We now begin our study of the **List ADT** — general types of collections of objects.
- To avoid confusion, we are discussing the List **ADT**, not linked list. The List ADT can be implemented using either array or linked list.
- One basic operation for lists is testing whether a given element exists 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 (i.e. they contain the same memory address).

- If we want to compare the **content of two objects**, we need to define a custom `equals()` method. For example, when comparing two Strings.

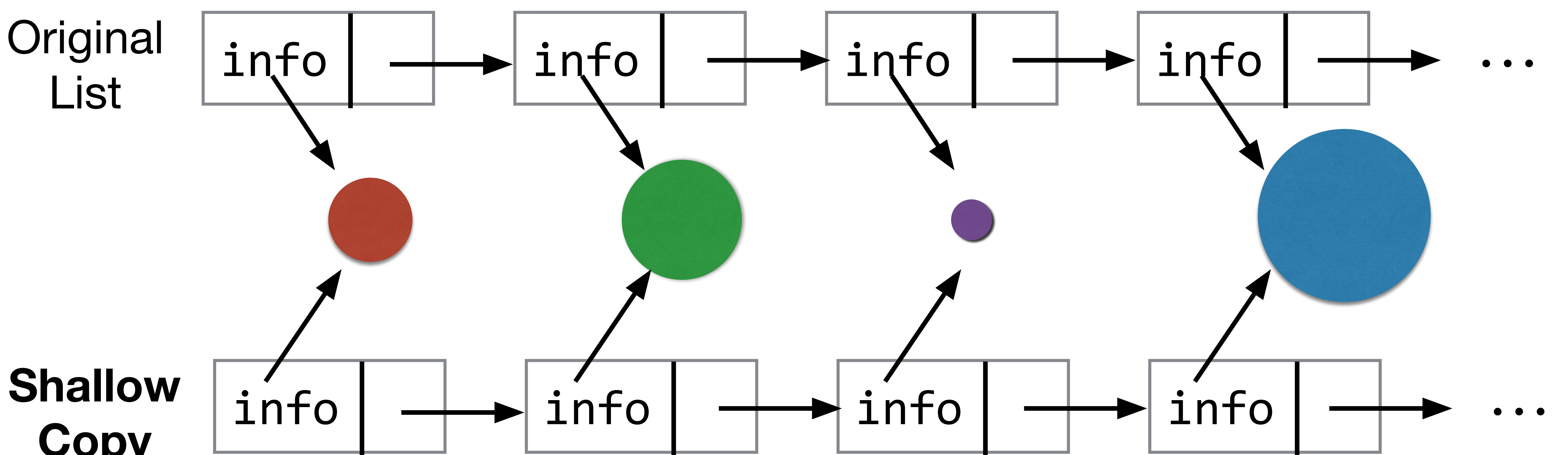
Comparing Objects in Java

- 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 (custom .equals method):

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

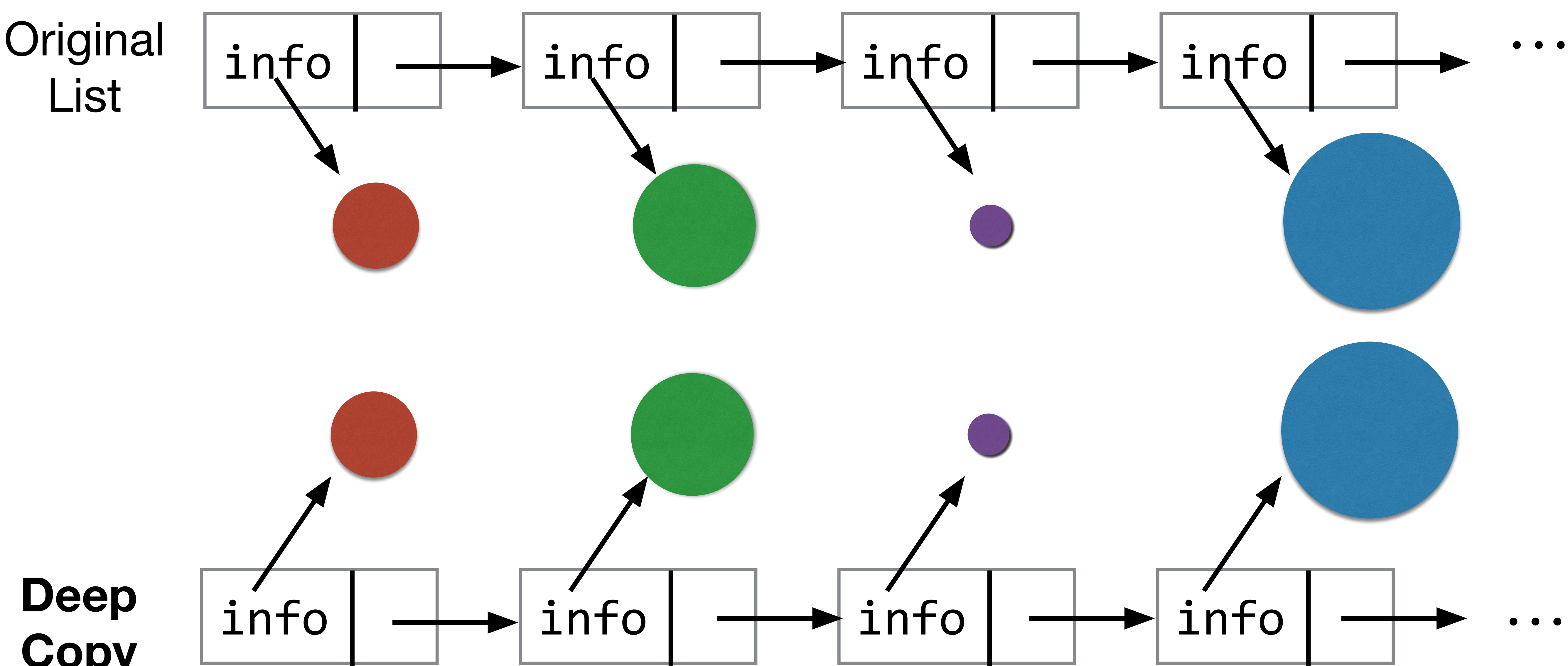
Shallow Copying

When you "copy" a data element, you can simply copy its reference, without cloning its content (i.e. no new memory is allocated). This is called **Shallow Copying**.



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 lexicographic (aka alphabetical) 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 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) if '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

- Custom compareTo for our Circle class:

```
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 #2

```
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 that) {
    return (this.x * this.x + this.y * this.y) -
(that.x * that.x + that.y * that.y);
  }
}
```

What is the result of calling
`(new Point(3, 4)).compareTo(new Point(4, 3))`?

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

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**.

- 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. A list without this property is **unsorted.**

- By default, our list is **indexed**, meaning that we can access elements using their positions in the list. In an indexed list, we would need a method to “return the k-th element” where k is the index (the first element has an index of 0).

Assumptions About Lists

- Lists are **unbounded** -- if implemented with arrays, the arrays must be able to expand dynamically.

- **Duplicate elements** (where one equals the other) **are allowed**. When searching for an element that has duplicates, generally our agreement is to return the first occurrence of the element.

- We do not support `null` elements. In other words, no element has a value of `null`. 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. This is often considered more graceful than throwing exceptions everywhere.

- Sorted lists are by default in **non-decreasing** order. Indexed lists have indices ranging from `0` to the `size-1`, with no gaps.

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

The List<T> Interface

```
public interface ListInterface<T> extends Iterable<T> {

    int size();
    // add an element.
    void add(T element);
    // return true if this list contains an element e
    // such that e.equals(element) is true
    boolean contains (T element);
    // remove an element from the list.
    // return true if successful, false if element doesn't exist
    boolean remove (T element);


    // continue to the next page
```

The `List<T>` Interface

```
    // insert an element at a given index. Higher-indexed
    // elements move up. Return true if successful
    boolean insert (int index, T element);
    // set the element at a given index
    // return true if successful
    boolean set (int index, T element);
    // get the element at a given index
    // return null if element does not exist
    T get (int index);
    // return the index of the first occurrence
    // of the element. -1 if element does not exist.
    int indexOf (T element);
    // remove the element at a specified index.
    T remove (int index);
}
```

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.insert(0, cardie);
x.insert(0, duncan);
x.insert(1, whistle);
x.insert(0, whistle);
x.set(2, whistle);
x.remove(0);
x.insert(1, cardie);
x.remove(2);
return x.indexOf(cardie);
```

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

Now let's implement an **unsorted** list using an array…

Class Variables and Constructor

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

size() and indexOf()

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

public int indexOf(T elem) {
?








}
```

size() and indexOf()

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

public int indexOf(T elem) {
    int location = 0;
    while (location < numElements) {
        if (list[location].equals(elem)) {
            return location;
        } else
            location++;
    }
    return -1;  // not found
}
```

contains, set, get

```
public boolean contains (T elem) {
  return (indexOf(elem) != -1);
}
public T get(int index) {
  if(index<0 || index>=numElements)
    return null; // index out of bounds
  return list[index];
}
public boolean set(int index, T elem) {
  if(index<0 || index>=numElements)
    return false; // index out of bounds
  list[index] = elem;
  return true;
}
```

Dynamic Resizing

To add elements, we need the list to be unbounded, so we have to implement an `enlarge` method to dynamically expand the array's capacity.

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

- **Q: When does the old array get recycled?**

add, remove

```
public void add (T elem) {
    if (numElements == list.length) enlarge();
    list[numElements++] = elem;
}

public boolean remove (T elem) {
```

add, remove

```
public void add (T elem) {
    if (numElements == list.length) enlarge();
    list[numElements++] = elem;
}

public boolean remove (T elem) {
    int location = indexOf(elem);
    if (location != -1) {
        for(int i=location; i<numElements-1; i++)
            list[i]=list[i+1];
        numElements--;
        return true;
    }
    return false;
}
```

- **Does this still work if the element to be removed is the last element?**

insert at a given index

```
public boolean insert (int index, T elem) {
    if(index<0 || index>numElements) return false;
    if(numElements == list.length) enlarge();

    // elements with higher indices move up
    // to make space for the new element



}
```

insert at a given index

```
public boolean insert (int index, T elem) {
    if(index<0 || index>numElements) return false;
    if(numElements == list.length) enlarge();

    for(int i=numElements; i>index; i--)
      list[i] = list[i-1];

    list[index] = elem;
    numElements++;
}
```

- **Does this still work if the element is to be inserted at index `numElements`?**

Clicker Question #4

In the insertion code from the previous slide, what would happen if we change the loop to the following:

```
for(int i=index; i<numElements; i++)
  list[i+1] = list[i];
```

(a) nothing would change (it still shifts all elements correctly)

(b) all elements from `index` to the end would be the same

(c) all elements from `index` to the end would be over-shifted by one position to the right.

(d) all elements from `index` to the end would be over-shifted by one position to the left.

(e) nothing would change except when `index` is equal to `numElements`

A Sorted Version of ArrayList

- In some cases we want the elements in the list to be **sorted** at all times. This accelerates certain operations such as search (which you will see in the future).
- By default, elements are sorted in **non-decreasing** order.
- Most methods in the unsorted version are still valid, so we can have `ArraySortedList` **extend** `ArrayUnsortedList`.
- One major difference is the **add** method: it needs to ensure the new element is added to the correct position to **maintain the order**.
- Inserting and setting an element at a specified index do not apply to sorted list, so we ignore them for now.

ArraySortedList

```
public class ArraySortedList<T> extends ArrayUnsortedList<T> {
  public void add (T elem) { // must preserve sorted order
    int location = 0;
    if (numElements == list.length)  enlarge();
```

Find the location where the new element should be inserted

Shift higher indexed elements up and insert the new element

```
    numElements++;
  }
```

ArraySortedList

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

This cast is necessary. Without it, the compiler will complain that it doesn't know T is a class that implements Comparable<T>