INTRODUCTION
-----------------
- We have started to use ADTs, like lists, maps, stacks, sets, seen that they're useful.
- Now, we're going "under the hood": how to implement them, how to do it efficiently
- why should you care?
- a) to use effectively, it helps to know how they work -- particularly, what can be done efficiently.
- b) different kind of programming, providing a service/abstraction, messy details. design for anticipated use.
- c) a lot of programming in real world is middleware. you will probably never be asked to implement a linked list again, but if you go to work at Google, they may ask you to help support a distributed key-value store

- Today we look at list implementations, and maybe start the conversation about maps
- Efficiency is a big part of conversation, so tomorrow talk about how we measure work, etc.
- Sorting, a good forum for talking about algorithm design and efficiency, and flexing our recursive "muscles"
- Then we get intro trees, both a kind of ADT, but also key to one approach to building an efficient map and queue impls

ARRAY IMPLEMENTATION
-----------------------
A natural way to implement a list is to store items in an array. This is covered pretty well in the book so I don't plan to spend much class time going over it.

- Note the use of generics in class definition. This says that this class is defined in terms of a type that is to-be-specified at *runtime*

- SimpleArrayList implement the List interface... actually, I have only implemented a few of the methods, the others throw an UnsupportedOperationException. Most of these methods aren't very interesting once you get the basic methods in place. We're focusing on the most essential features today.

- Inside our list class is an array
  - Why is it an array of Objects?
  - Short answer is that we cannot create an array of T's b/c T is not known until runtime.
  - The solution is to create an Object array, b/c whatever T is, it must be a subtype of Object
  - Then cast appropriately. While down casting is generally to be avoided, since we control the

- To get an item at index, just retrieve item at that position in array
- To set item at index, just set item in the array.

- To add item at end, we can put in last available spot in array.
  - Things start to get a little interesting: what if array is full?
  - System.arraycopy: arrays are contiguous blocks of memory, this system method calls the underlying OS to copy the memory, which is much faster than a for loop running in Java.

- To add at a specific index, we have to shift everything over.
- Similarly for remove

- Efficiency:
  - we'll talk a bit more formally about efficiency tomorrow, but let's talk informally about efficiency here
  - get, set
    -- very fast, random access
    -- references as sitting packed in an array
  - adding at the end
    -- if full, must expand
    -- copying the entire array can be expensive, but we only have to do it when full.
    -- our rule is to double array when full, so how often do I have to expand?
    -- suppose initial capacity is 1, and I add 1000 item in a row, how many expansions?
  - adding/removing the middle
    -- copy everything, system call is fast, but for a big list, this can be expensive.
    -- e.g., suppose when we added 1M items, we added them at the beginning of the list rather than the end?
    -- then we would make 1M copies, of increasingly larger sizes

LINKED IMPLEMENTATION
-----------------------
- So, arraylists are fast for random access, but really slow for adding, such as adding to front of the list. The reading introduced a completely different way of implementing a list: the linked list. We'll see that it behaves differently than an arraylist.

- A good analogy:
  -- an array list is like a CD
  -- a linked list is like a cassette tape.
- We'll see that linkedlists can be faster for some operations.

- THE MAIN REASON for studying linked lists is to work on your mental model of references and objects to prepare you for our later work with trees.

- So, on to linked data:

- Instead of storing refs packed into an array, imagine scattered throughout memory.

[5, 4, 3, 7]


- To maintain the list, we need to keep track of them somehow, linking them together in the order in which they belong. We'll put each item into a container, then link the containers together, forming a chain.

NODES
------
That container is called a Node. We can define it as Java class, as follows. For the moment, let's just consider a list of ints. Later we'll change our implementation so it uses generics to hold any type of Object.

```java
class Node {
    int data;
    Node next;
}
```

Notice that this the Node class has a recursive definition. The definition of a Node is thing that has some data of type int and a reference to a Node. We will see other examples of recursively defined data structures.

Also, the fields are not explicitly private, so we can access them. This is mostly for convenience. Later, We'll see why this violation of encapsulation is not such a problem.

We can add constructors:

```java
class Node {
    ...
    public Node(int data, Node next) {
        this.data = data;
        this.next = next;
    }
    public Node(int data) {
        this(data, null);
    }
}
```

Before we start using Nodes to build a list, let's play around with them for a bit. We can create new nodes and chain them together like this. On the board, I'll draw our objects and variables in the usual way.

Node list; reference to null
new Node(5) creates an Node object
list = new Node(5) assigns reference to this object.

We can create another Node object. How do we link it to this one? That is, what variable should we update so that it has a reference to this object?

```java
list.next = new Node(4);
```

There's nothing mysterious going on here. We can also do something like this: create a new Node object, and create a variable x that holds a reference to this new Node. How would would like link this object, o2, to the new object, o3?

```java
x = new Node(3);
list.next.next = x;
```

List is a reference object o1. list.next refers to this variable inside o1, which we can see holds
reference to $o_2$. So, think of (list.next) a handle for this object $o_2$. (list.next).next is the variable called "next" inside the $o_2$ object.

ALTERNATIVE REPRESENTATION

Alternative way to draw them that will be a little more convenient. Represent a reference to an object/primitive as a box, and draw an arrow connecting that box to the object it points to.

A Node object has two things it: a data item and a ref to the next Node. We can draw them with a box like this. We can draw this whole linked structure. This slash is meant to denote a pointer to null.

This is a shorthand, and we'll see it's convenient, but it should be clear that it's similar to the way we've been drawing objects and refs so far. If this starts to get confusing, you may want to revisit the old approach.

So we want to ultimately create a List implementation, but this idea of linked data can get a little tricky and it's worth getting comfortable with this next pointers before diving into the list implementation.

Here's an example to think about: Suppose we have this situation:

```
+------+------+      +------+------+
| data | next |      | data | next |
+---+      | 2     |      | 4     | /
+---+      +------+------+
+------+------+

+------+------+      +------+------+
| data | next |
+---+      +------+------+
| 3     | 9     | /
+---+      +------+------+
+------+------+
```

And we want to get to this situation:

```
+------+------+      +------+------+      +------+------+
| data | next |      | data | next |      | data | next |
+---+      +------+------+
| 2     | 4     |      | 3     | /
+---+      +------+------+
+------+------+

- Let's number them.
- Which ones have to change to get us where we want to be? 3, 4, 5
- What sequence of java statements can we execute to achieve the desired change?

- First of all: we see we want to change this variable, here. how do we even refer to this one?
  - p.next
  - p.next.next
- Now we can make assignment statements. it's a little tricky b/c order matters!
- Individually, take a minute and write down a sequence of statements that will update the variables so we
  have a picture like this.

  ```java
  p.next.next = q;
  q = q.next;
  p.next.next.next = null;
  ```

This is tricky. It's easy to draw the boxes and just reconnect the arrows. Why is it harder to write it
as Java? Well, because when we look at the drawing, it's easy to "grab" an arrow and redirect it. But
in Java we can also access the data by traversing these arrows, these next references. So we have to
think carefully before manipulating them so that we always maintain a reference to the data.

**CHANGE TO TYPE T**

--------------

Now, we were working with Nodes that held an int as data. We can easily change this to an arbitrary type
T. Can think of this now as a pointer off to some T object. Or perhaps we can just be a little lazy and
think of the T object as sitting inside the Node. Our focus should be on the link structure, and so let's
not worry too much about how we draw the reference to the data.

Change Java code to use generics.

```java
class Node<T> {
    T data;
    Node next;

    public Node(T data, Node<T> next) {
        this.data = data;
        this.next = next;
    }
    public Node(T data) {
        this(data, null);
    }
}
```

**LIST IMPL**

----------

- List class
- Member data of a list object is very simple: has a reference to a list node.
public class SimpleLinkedList<T> {

    Node list;

    public SimpleLinkedList() {
        list = null;
    }
}

private class Node<T> {
    T data;
    Node next;

    Node(T data, Node next) {
        this.data = data;
        this.next = next;
    }

    Node(T data) {
        this(data, null);
    }
}

- We will move Node to be an inner class and make it private
  - Why?  Explained a bit in the book
  - Basically, inaccessible to outsiders (which makes sense here, we want to encapsulate)
  - List can access private data of inner class (no compelling reason to encapsulate)
  - But want it to be a separate class b/c we want to make new Node objects.

public class SimpleLinkedList<T> {

    Node list;

    public SimpleLinkedList() {
        list = null;
    }
}

private class Node<T> {
    T data;
    Node next;

    Node(T data, Node next) {
        this.data = data;
        this.next = next;
    }

    Node(T data) {
        this(data, null);
    }
}

- Once it becomes an inner class, we can drop the generic in the Node definition, this T is whatever
generic type the list is made up of.  This ensures consistency between the type held in the List and the
type held in the node.

    private class Node {
        // drop the <T> and use the <T> from SimpleLinkedList,
    }

- Let's implement the add method.  Let's add elements at a specific at a given position.

    public void add(int i, T item) {


- 6/8 -
- So imagine we have a list and it looks something like this. Let's start with a special case: what if we want to add the item to the beginning of the list? I.e., i = 0?

- Well, first thing we need to do is put the new data into a Node object. So make a new Node.

        Node n = new Node(item);

- Now, drawing it as a picture, where are we trying to get to. We have the list. And we have n pointing to a new node object that's not in the list yet. We want to update the references to get this. What Java statements will get us there?

        if (i == 0) {
            n.next = list;
            list = n;
        }

- Okay, that was the special case of i = 0. What about arbitrary i? Let's draw the picture. What references need to change?

- So it's clear that we need to have a reference to the node before the spot where we want to insert the new node. So, we want the node at position i-1. Let's assume we have a method that will get us the node at position i-1.

        Node before = getNodeAt(i-1);

- So now we have a handle on the node before the place we want to add and we have a handle on the new node. (Draw pointers for before and n.)

- Take a moment and think about what sequence of statements will give us our desired update.

        else {
            Node before = getNodeAt(i-1);
            Node current = before.next;
            n.next = current;
            before.next = n;
        }

- Implementing getNodeAt is pretty straightforward. Simply start at the beginning of the list and call next the right number of times to land you at i-1.

- Now, let's think about remove.

        /** Removes item at position i and returns it to caller */
        public T remove(int i) {

        }

- Quite similar to add. Working in small groups 2-3. Implement remove.
- Class finished up while students were working on remove.