CS 151

Linked Lists, Recursively Implemented
Recall that a **linked list** is a data structure that represents a sequence of elements that are stored non-contiguously in memory.

We can equivalently define a LL recursively, as follows: A linked list is either:
- an empty list, or
- an item of type T followed by a linked list.

Both definitions represent the same structure (a linked list), but they imply different implementations. The non-recursive definition used a Node class to represent each element and their pointers in memory.

The recursive definition will use inheritance to define the recursive relationships. Our methods will thus be implemented differently as well, using recursion.

Disclaimer: Java is not the best language in which to implement LLs this way, but we’re going to push through it anyway so that you can see it.
Recall that our old implementation started out like this:

```java
class LinkedList<T> {
    class Node<T> { // points to the front of the list
        private T data // the number of elements in the list
        private Node<T> next

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

        public T getData() return data
        public Node<T> getNext() return next
        public void setData( T data ) this.data = data
        public void setNext( Node<T> next ) this.next = next
    }

    Node<T> front
    int size

    // Linked List class methods go here
}
```
Recursive Linked List Implementation

Our new implementation will have 3 separate public classes:

/* This abstract class represents a Linked List. It defines the methods that every linked list must have, but doesn’t implement them. We will have different implementations depending on whether our list is empty, or is a “ConsList”, i.e. an item followed by a Linked List. */

public abstract class LinkedList<T> {
    // method headers for LL methods go here
}

public class EmptyList<T> extends LinkedList<T> {
    // method implementations for empty lists go here
}

public class ConsList<T> extends LinkedList<T> {
    T               data    // the data of the front element
    LinkedList<T>   rest    // the LL that follows the front elt

    // method implementations for non-empty lists go here
}
How to Construct Recursively Implemented LLs

A user of a LinkedList knows there are 2 actual (non-abstract) classes. They can store lists of either type in a variable of type LinkedList.

For example, to construct an empty list, you could do this:

```java
LinkedList<Integer> e = new Empty<Integer>();
```

To make the list with one element, say [3], you would do this:

```java
LinkedList<Integer> c = new ConsList<Integer>( 3, e );
```

To make the list [5,3], you would do:

```java
new ConsList<Integer>( 3, new ConsList<Integer>(3,e) );
```

```java
public abstract class LinkedList<T> {...}
public class EmptyList<T> extends LinkedList<T> {...}
public class ConsList<T> extends LinkedList<T> {
    T data    // the data of the front element
    LinkedList<T> rest    // the LL that follows the front elt
}
```
How to Implement LL Methods

To add a method to our Linked List class, we must do three things:

• add the method as an abstract method header in the LinkedList class
• implement the method in the Empty class
• implement the method in the Cons class

For example, let’s start with the isEmpty() method.

We first add the method header to the LinkedList class, to declare that any class that implements LinkedList must have the isEmpty method.

```java
public abstract class LinkedList<T> {
    public abstract boolean isEmpty();
}
```

In the EmptyList class, we implement the method. Is an EmptyList empty?

```java
public class EmptyList<T> {
    public boolean isEmpty()        { return true; } // always.
}
```

In the ConsList class, we implement the method. Is an ConsList empty?

```java
public class ConsList<T> {
    public boolean isEmpty()        { return false; } // always.
}
```
Now the makeEmpty() method. Note the change in return type.

We add the method header to the abstract LinkedList class:

```java
public abstract class LinkedList<T> {
    public abstract LinkedList<T> makeEmpty();
}
```

We implement the method in the Empty class.

```java
public class EmptyList<T> {
    public LinkedList<T> makeEmpty() { return new EmptyList<T>(); }
}
```

We implement the method in the Cons class.

```java
public class ConsList<T> {
    T data
    LinkedList<T> rest

    public LinkedList<T> makeEmpty() {
        return new EmptyList<T>();
    }
}
```
How to Implement LL Methods

Now let’s try the size() method. We use recursion on the *structure*...

We add the method header to the abstract LinkedList class:

```java
public abstract class LinkedList<T> {
    public abstract int size();
}
```

We implement the method in the Empty class. What is the size of an empty list?

```java
public class EmptyList<T> {
    public int size() { return 0; } // “base case”
}
```

We implement size in the Cons class. What is the size of this cons list?

```java
public class ConsList<T> {
    T             data
    LinkedList<T> rest

    public int size() {
        return 1 + rest.size() // because of our recursive definition
    }
```
Now let’s try the get(int index) method.

We add the method header to the abstract LinkedList class:

```java
public abstract class LinkedList<T> {
    public abstract T get(int index);
}
```

We implement the method in the Empty class:

```java
public class EmptyList<T> {
    public T get(int index) { throw IndexOutOfBoundsException }
}
```

We implement the method in the Cons class:

```java
public class ConsList<T> {
    T             data
    LinkedList<T> rest

    public T get(int index) {
        if( index < 0 )  throw IndexOutOfBoundsException
        if( index == 0 ) return data
        else             return rest.get(index-1)
    }
}
```
How to Implement LL Methods

Now let’s try the add(T item, int index) method. Note change in return type.

```java
public abstract class LinkedList<T> {  
    public abstract LinkedList<T> add(T item, int index);

    public class EmptyList<T> {  
        public LinkedList<T> add(T item, int index) {  
            if( index != 0 ) throw IndexOutOfBoundsException  
            return new ConsList<T>( item, new EmptyList<T>() )  
        }

        public class ConsList<T> {  
            T             data  
            LinkedList<T> rest

            public LinkedList<T> add(T item, int index) {  
                if( index < 0 ) throw IndexOutOfBoundsException  
                if( index == 0 ) return new ConsList<T>( item, this )  
                else    return new ConsList<T>(data,rest.add(item, index-1))
            }
```
Now the `remove(T item, int index)` method. Note change in return type.

```java
public abstract class LinkedList<T> {
    public abstract LinkedList<T> remove(int index);

    public class EmptyList<T> {
        public LinkedList<T> remove(int index) {
            throw NoSuchElementException
        }
    }

    public class ConsList<T> {
        T             data
        LinkedList<T> rest

        public LinkedList<T> remove(int index) {
            if( index < 0 )  throw IndexOutOfBoundsException
            if( index == 0 ) return rest
            else       return new ConsList<T>(data, rest.remove(index-1))
        }
    }
}
```
Let’s try to implement a new method: reverse() that reverses the list:

```java
public abstract class LinkedList<T> {
    public abstract LinkedList<T> reverse();

    public class EmptyList<T> {
        public LinkedList<T> reverse() {
            return this;
        }
    }

    public class ConsList<T> {
        T data
        LinkedList<T> rest

        public LinkedList<T> reverse() {
            return rest.reverse().add(data, size()-1);
        }
    }
}
```
Another new method: `find(T item)` that returns whether item is in the list:

```java
public abstract class LinkedList<T> {
    public abstract boolean find(T item);
}

public class EmptyList<T> {
    public boolean find(T item) {
        return false // item is never in an empty list
    }
}

public class ConsList<T> {
    T data
    LinkedList<T> rest

    public boolean find(T item) {
        if(item.equals(data)) return true
        else return rest.find(item)
    }
}
How to Implement LL Methods

One more: append( LinkedList<T> list ) that appends “list” onto end of our list

```java
public abstract class LinkedList<T> {
    public abstract LinkedList<T> append( LinkedList<T> list );

    public class EmptyList<T> {
        public LinkedList<T> append( LinkedList<T> list ) {
            return list
        }
    }

    public class ConsList<T> {
        T             data
        LinkedList<T> rest

        public LinkedList<T> append( LinkedList<T> list ) {
            return new ConsList<T>( data, rest.append( list ) )
        }
    }
}
```