Linked Lists

- Arrays are limited in the sense they have a fixed size
  - Resizing as needed must be done carefully and is not efficient
- A linked structure is the primary alternative to an array-based implementation of a collection
- Initial implementation idea: a class Person as part of a community

```java
public class Person {
    private String name;
    private String address;
    private Person next;
    // and whatever else
}
```

Elements with Links

- Using only this one class, a linked structure of “nodes” is created
  - One Person object contains a link to another Person object
  - This second object contains a reference to a third Person, etc.
  - This type of object is sometimes called self-referential
  - This kind of relationship forms the basis of a linked list
- Has capacity limited only by memory in the computer
  - Is a dynamic structure because its size grows and shrinks as needed to accommodate the number of elements stored

Doubly Linked Lists

- A simple linked list is only one kind of linked structure
- In a doubly linked list, each node in the list stores both a reference to the next element and a reference to the previous one
- Java’s library implementation uses doubly linked lists

Nonlinear Linked Structures

- Using multiple links you can create non-linear structures
Managing Linked Lists

- A node may be inserted or deleted at any location
  - at the front of the list,
  - among the interior nodes, or
  - at the end of the list
- There are a few basic techniques when managing nodes on the list, no matter what the list is used to store
- Special care must be taken when dealing with the first or last node in the list so that the reference to the entire list is maintained appropriately

Inserting a node at the front

Inserting a node in the middle

Deleting the first node in the list
Deleting an interior node

Better: Elements Without Links

A flaw in the Person class is that the self-referential Person class must be designed so that it “knows” it may become a node in a linked list.

This violates the goal of separating the implementation details from the parts of the system that use the collection.
A Reference-Based Stack Implementation

- **LinkedStack**
  - Implements Stack Interface
  - `top` is a reference to the head of a linked list of LinearNodes

---

14.12 - The LinkedStack Class

```java
package javafoundations;
import javafoundations.exceptions.*;
public class LinkedStack<T> implements Stack<T> {
    private int count;
    private LinearNode<T> top;

    // Creates an empty stack using the default capacity.
    public LinkedStack() {
        count = 0;
        top = null;
    }

    // Removes the element at the top of this stack and returns a reference to it. Throws an EmptyCollectionException if the stack contains no elements.
    public T pop() throws EmptyCollectionException {
        if (count == 0)
            throw new EmptyCollectionException("Pop operation failed. " + "The stack is empty.");
        T result = top.getElement();
        top = top.getNext();
        count--;
        return result;
    }

    // The following methods are left as Programming Projects.
    // public void push () { }
    // public T peek () throws EmptyCollectionException { }
    // public boolean isEmpty () { }
    // public int size() { }
    // public String toString() { }
}
```

Java's LinkedList<E> "feels" like an array:

Even though Java uses doubly linked list, we will use the simplified picture because it is simple and clear about accessibility:

To use it you need to start your code by:
```
import java.util.*;
```
To get a new list:
```
LinkedList<E> L = new LinkedList<E>();
```
Stack Implementation using Java’s LinkedList <E>

- The ADT LinkedList can be used to represent the items in a stack

```java
public E element() // same as peek
public void push(E item)
public E pop() throws NoSuchElementException

LinkedList<String> L1 = new LinkedList<String>();
L1.add(0, "eggs");
L1.addFirst("milk");
L1.addLast("bread");
L1.add(2, "chicken");
System.out.println("contents of L1: "+ L1);
```

Methods for Adding LinkedList Nodes

```java
public void add(int index, E element)
indices change automatically!
if index > size() throws IndexOutOfBoundsException
public void addFirst(E element)
public void addLast(E element)
aka: public void add(E element)

LinkedList<String> L1 = new LinkedList<String>();
L1.add(0, "eggs");
L1.addFirst("milk");
L1.addLast("bread");
L1.add(2, "chicken");
System.out.println("contents of L1: "+ L1);
```

Methods for Setting and Removing Items

```java
public E set(int index, E element)
public E remove(int index)
public E removeFirst()
public E removeLast()

L1.remove(1);
L1.set(2, "beer");
L1.set(1, L1.removeFirst());
L1.addFirst(L1.getLast());
L1.add(1, "butter");
System.out.println("new contents of L1: "+ L1);  
```

(what happened to the beer?)
Append and Share

```java
/** appends the list L2 onto the end of L1 */
public static void append (LinkedList<E> L1, LinkedList<E> L2) {
    for (int i = 0; i < L2.size(); i++)
        L1.addLast(L2.get(i));
    // The nodes of L2 are now shared with L1!
}
```

Append and Do Not Share

```java
/** appends a copy of the nodes of L2 to the end of L1 */
public static void copyAppend (LinkedList<E> L1, LinkedList<E> L2) {
    for (int i = 0; i < L2.size(); i++)
        L1.addLast(clone(L2.get(i)));
}
```

Insert a list inside another list

```java
/** inserts nodes of L2 into the list L1, *
 * starting at the input index */
public static void insertList (LinkedList<E> L1, LinkedList<E> L2, int index) {
    for (int i = 0; i < L2.size(); i++)
        L1.add(index + i, L2.get(i));
}
```