Implementing Linked Lists

- L&C's Single-Link LinkedList Class
- The remove Operation for LinkedList
- Doubly Linked Lists and the New remove Using find
- The LinkedList Class
- Linked Ordered Lists
- Reviewing the DLDeque From Discussion #6
L&C’s Single-Link LinkedList Class

• We can use our existing LinearNode class to implement a list, whether ordered or unordered. The Java library uses doubly-linked nodes, as we will see soon, but let’s find out what happens with singly-linked nodes.

• As we saw when looking at deques, we can add or remove from the front, or add to the rear, or peek at either, in O(1) time. But the ListADT interface also needs us to remove an object from the list, including removing from the rear, and to create an iterator object.

```java
class LinkedList<T> implements ListADT<T>, Iterable<T> {  
    protected int count;  
    protected LinearNode<T> head, tail;  
    public LinkedList( ) {  
        count = 0;  
        head = tail = null;  
    }
```
The **remove** Operation for **LinkedList**

- We can’t break out the **find** part of **remove**, because we need the node *before* the one we are looking for.

```java
public T remove (T target ) throws EmptyCollectionException,
                    ElementNotFoundException {
    if (isEmpty( )) throw new EmptyCollectionException ("List");
    boolean found = false;
    LinearNode<T> previous = null;
    LinearNode<T> current = head;
    while (current != null && !found) {
        if (target.equals (current.getElement( )) found = true;
            else {previous = current; current = current.getNext( );}}
    if (!found) throw new ElementNotFoundException ("List");
    if (size( ) = 1) head = tail = null;
    else if (current == head) head = current.getNext( );
    else if (current == tail)) {
        tail = previous; tail.setNext(null);
    else previous.setNet(current.getNext( ));
    count--; return current.getElement( );}
```
Doubly Linked Lists

• It’s easy to adapt LinearNode to make a doubly linked node. We don’t extend LinearNode because we need the next field to be different and there are really no LinearNode methods we would make much use of.

• Though they don’t mention it in the text, L&C declare a class DoubleList, whose fields and constructors are like those of LinkedList except that they use DoubleNode objects, and say front and rear, not head and tail.

• With free access to the predecessor of any node, things become simpler.

```java
public class DoubleNode<T> {
    private DoubleNode<T> next;
    private T element;
    private DoubleNode<T> previous;
    // get and set methods, zero- and one-parameter constructors
```
The **find** Method

- Remember that in the array implementation we separated out finding an element in the list from removing it. The `find` method was also useful for implementing `contains`.

- We make `find` a private method because its result is a node, an inherently implementation-dependent thing, rather than something for the user.

```java
private DoubleNode<T> find (T target) {
    boolean found = false;
    DoubleNode<T> traverse = front;
    DoubleNode<T> result = null;
    if (!isEmpty( ))
        while (!found && traverse != null)
            if (target.equals(traverse.getElement( ))) found = true;
            else traverse = traverse.getNext( );
    if (found) result = traverse;
    return result;}
```
Removing From a Doubly-Linked List

• Once we find the element, our two commands to remove it are symmetrical.

• We handle the cases of the element being the front or rear one separately, since we already have methods for those two operations.

```java
public T remove (T element) throws ElementNotFoundException {
  T result;
  DoubleNode<T> node = find (element);
  if (node == null) throw new ElementNotFoundException("List");
  result = node.getElement();
  if (node == front) result = this.removeFirst();
  else if (node == rear) result = this.removeLast();
  else {
    node.getNext().setPrevious(node.getPrevious());
    node.getPrevious().setNext(node.getNext());
    count--;
  }
  return result;
}
```
The **LinkedIterator** Class

- This is for singly-linked lists -- there is a similar class for doubly-linked.

- The **Iterator** interface includes a `remove` method, to remove the element just returned by `next`. If we don’t write it, we have to include a stub.

```java
public class LinkedIterator<T> implements Iterator<T> {
    private int count;
    private LinearNode<T> current;
    public LinkedIterator (LinearNode<T> collection, int size) {
        current = collection; count = size;
    }
    public boolean hasNext( ) {return (current != null);}
    public T next( ) throws NoSuchElementException {
        if (!hasNext( )) throw new NoSuchElementException( );
        T result = current.getElement( );
        current = current.next( );
        return result;
    }
    public void remove( ) throws UnsupportedOperationException {
        throw new UnsupportedOperationException( );
    }
}
```
Linked Ordered Lists

- If our base type $T$ implements `Comparable<T>`, we can maintain the natural order on the elements of a list, and implement L&C's `OrderedListADT` interface. The essential new operation is `add`, which puts its parameter element in the right place in the list.

```java
public void add (T elem) {
    LinearNode<T> newNode = new LinearNode<T>(elem);
    if (isEmpty() ) {head = tail = newNode; count++; return;}
    if (head.getElement().compareTo(elem) >= 0) {
        newNode.setNext (head.getNext( ));
        head = newNode; count++; return;}
    LinearNode<T> before = head;
    while ((before.getNext( ) != null) &&
        (before.getNext( ).getElement( ).compareTo(elem) < 0))
        before = before.getNext( );
    newNode.setNext (before.getNext( ));
    before.setNext(newNode);
    if (newNode.getNext( ) == null) tail = newNode;
    count++; return;}
```
Reviewing the DLDeque From Discussion #6

- We need to make sure both links are right for every node, plus front and rear.

```java
public void addToRear (T element) {
    DoubleNode<T> newNode = new DoubleNode<T> (element);
    newNode.setPrevious (rear);
    newNode.setNext (null); // redundant
    if (rear != null) rear.setNext (newNode);
    rear = newNode;
    if (size == 0) front = newNode;
    size++;
}

public T removeRear ( ) throws EmptyCollectionException {
    if (size == 0) throw new EmptyCollectionException ( );
    if (size == 1) front = null;
    T ret = rear.getElement ( );
    rear = rear.getPrevious ( );
    rear.setNext (null);
    size--; return ret;
}
```