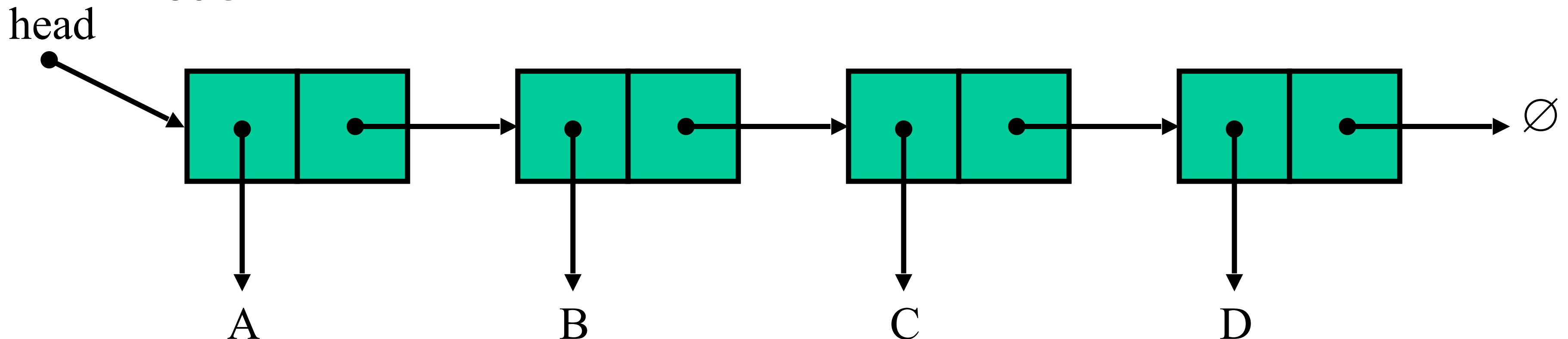
Linked Lists^2

Part 2

removeLast()

- Problem
  - How do you remove the last
  - Can we use the lastNode utility function?
    - Not exactly, because to remove D we need to do things to C
    - Cannot go backwards!!
- So, need to search forward in list to find the node before the last node



Linked Lists and Stacks

```
public class StackLL<S> implements StackIntf<S> {
    private SingleLinkedList<S> underlyingLL;
    public StackLL() { underlyingLL = new SingleLinkedList<S>(); }

    public boolean isEmpty() {  return underlyingLL.isEmpty();  }

    public S push(S e) {
        underlyingLL.addFirst(e);
        return e;
    }

    public S peek() {  return underlyingLL.first(); }

    public S pop() { return underlyingLL.removeFirst();  }

    public int size() {  return underlyingLL.size();  }

    public void clear() { underlyingLL = new SingleLinkedList<S>() }
}
```

Linked Lists and Queues

```
public class QueueLL<Q> implements QueueInterface<Q>{
    private SingleLinkedList<Q> underlyingLL;
    public QueueLL() {
        underlyingLL = new SingleLinkedList<Q>();
    }
    public int size() { return underlyingLL.size();  }

    public boolean isEmpty() { return underlyingLL.isEmpty(); }

    public Q peek() { return underlyingLL.first(); }

    public boolean enqueue(Q e) { underlyingLL.addLast(e); return true; }

    public Q dequeue() { return underlyingLL.removeFirst(); }

    public void clear() { underlyingLL == new SingleLinkedList<Q>(); }

}
```

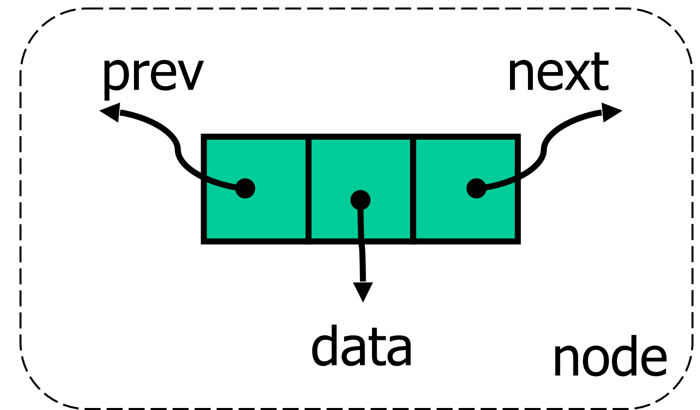
Musings on singly linked lists

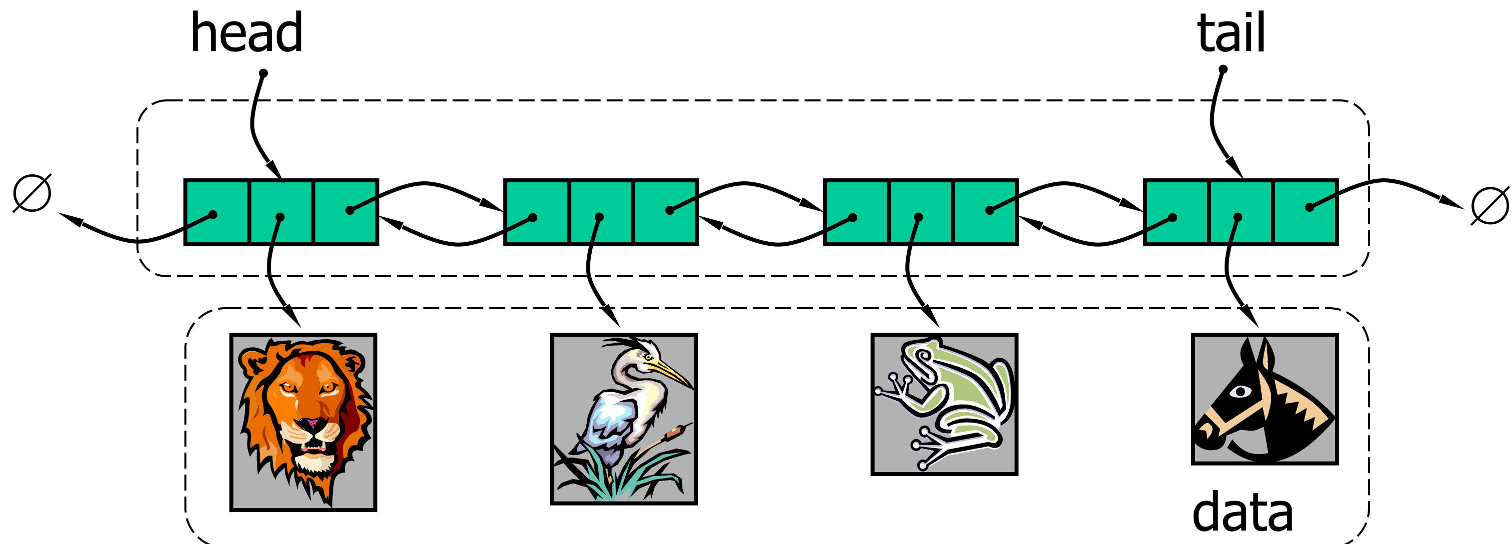
- The whole remove last method is a pain
  - and it is slow O(n)
- Not being able to go backward is a pain
  - Linked lists are a pain

- Can't do anything about linked lists be ing a pain

Doubly Linked List

- Can be traversed forward and backward
- Nodes store an extra reference





Double Linked List interface

```
public interface LinkedListInterfaceComp<E extends Comparable<E>> {
    int size();
    boolean isEmpty();
    E first();
    E last();
    void addLast(E c);
    void addFirst(E c);
    E removeFirst();
    E removeLast();
    E remove(E r);
    boolean contains(E iD);
}
```

This could also be applied to a single linked list (or an array list) or …

Node & DLL start

```
public class DoubleLinkedList<T extends Comparable<T>> implements
LinkedListInterfaceComp<T> {
    protected class Node<V extends Comparable<V>> {
        public V data;
        public Node<V> next;
        public Node<V> prev;
        public Node(V data) {
            this.data = data;
            this.next = null;
            this.prev = null;
        }
    }
    private Node<T> head = null;
    private Node<T> tail = null;
    private int size = 0;
```

Basics

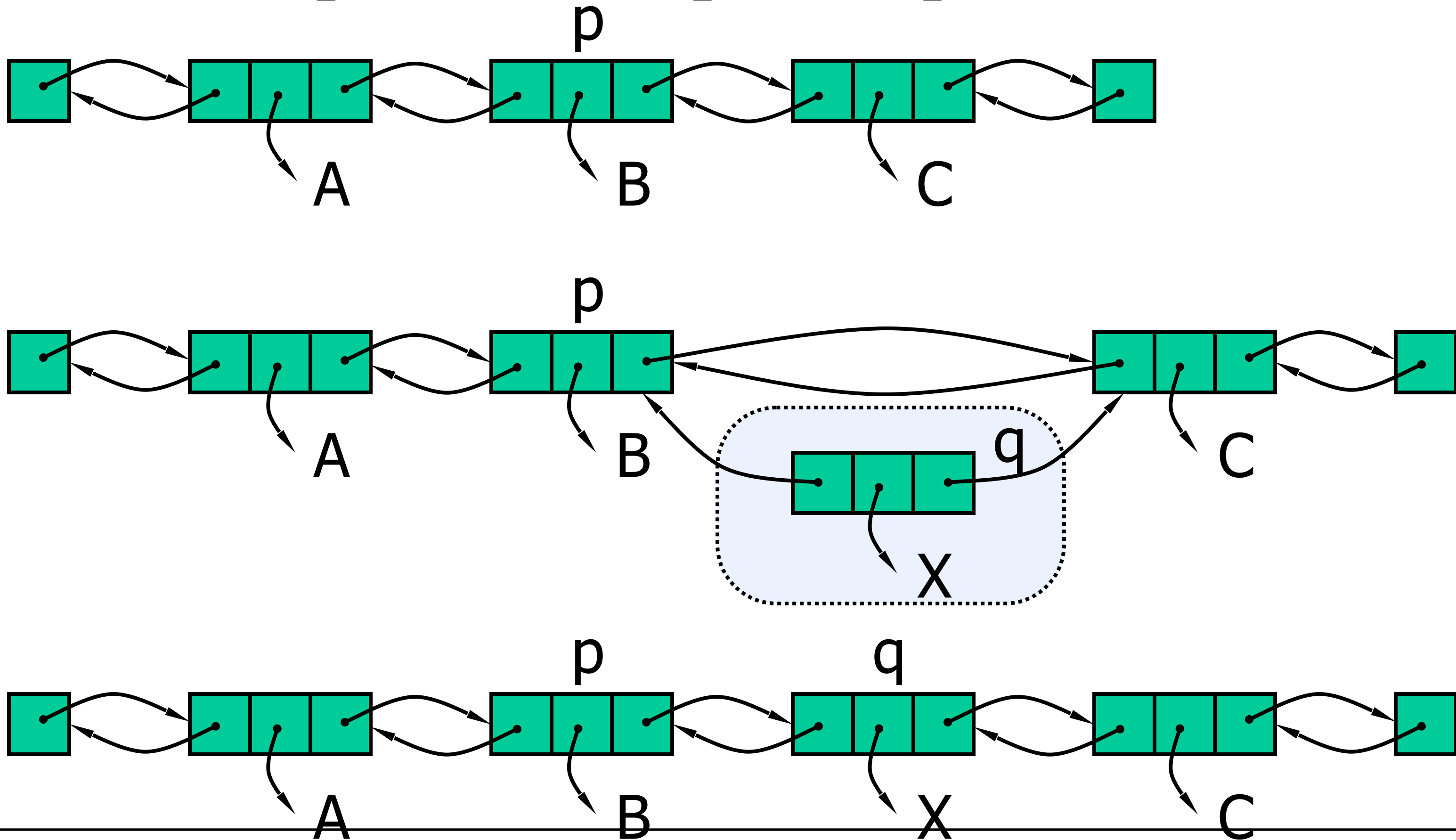
```
@Override
public int size() {
     return size;
 }
@Override
 public boolean isEmpty() {
     return size == 0;
 }

@Override
    public T first() {
     if (head == null)
         return null;
     return head.data;
 }
@Override
 public T last() {
     if (head == null)
         return null;
     return tail.data;
 }
```

Insertion: AddFirst, AddLast

Add Between

- Insert `q` between `p` and `p.next`



Add Between

```
protected void addBtw(T c, Node prev, Node next) {
    Node newest = new Node<>(c);
    prev.next = newest;
    next.prev = newest;
    newest.prev = prev;
    newest.next = next;
    size++;
}
```

Why not public?? Is this enough?

Deletion — last element

```
public T removeLast() {
        if (head == null)
            return null;
        T rtn = tail.data;
        if (head == tail) {
            head = null;
            size = 0;
            tail = null;
            return rtn;
        }
        tail = tail.prev;
        tail.next = null;
        size--;
        return rtn;
    }
```

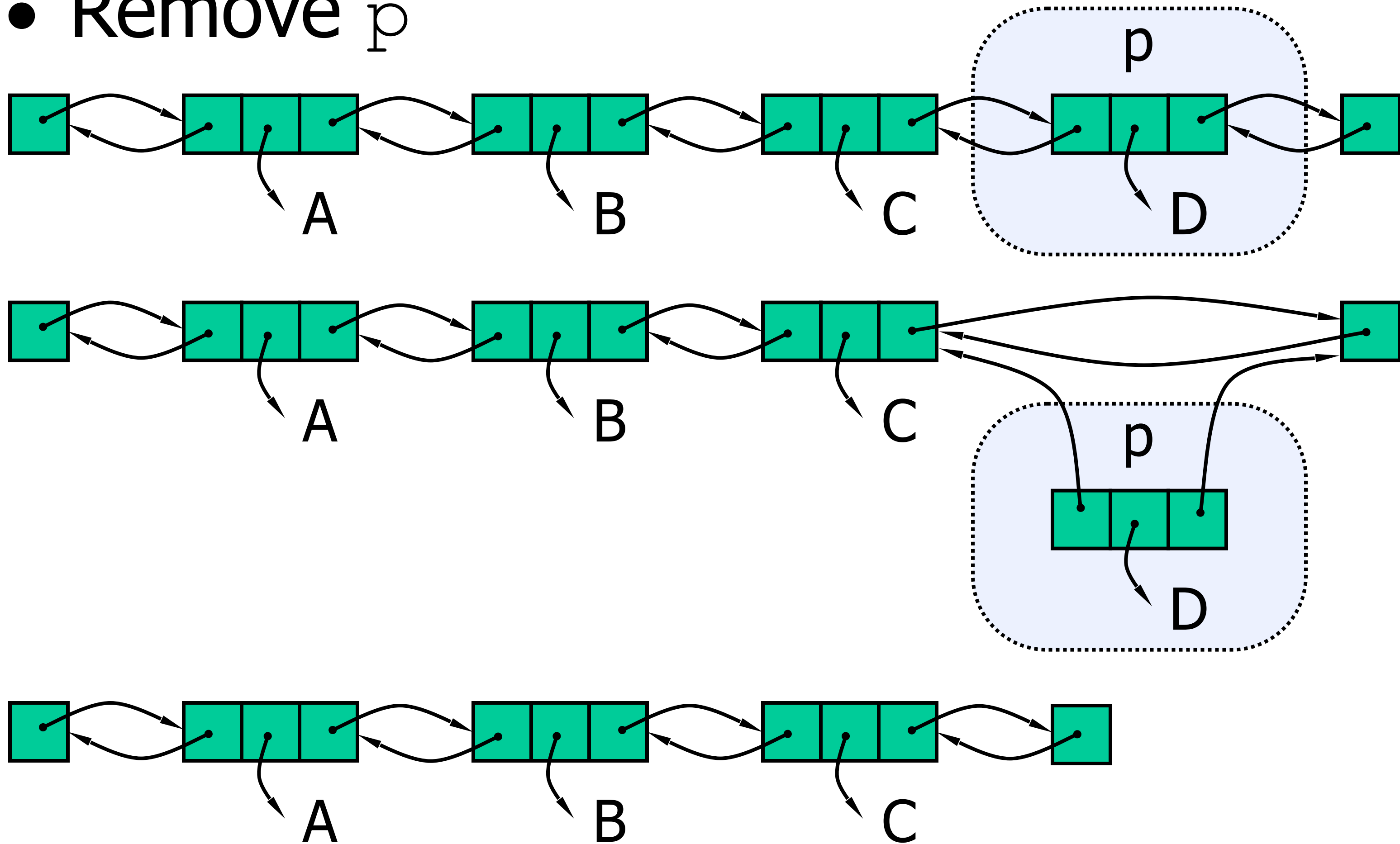
Deleting the first element is very similar

removeNth(int n)

- remove the item at the given position in the doubly linked list
  - start counting at 0
- If the linked list if shorter than n, do nothing
  - What other special cases need handling?

Deletion

- Remove `p`



Writing tests

- Goal is to identify places that code could break
- Figure out a set of inputs and/or outputs that distinguishes success from failure at that point.
- Each test should be independent of previous tests
- Ideally each test evaluates automatically ... ie final statement should be "pass" or "fail"
- Put each test in this own method
  - returns true / false

Tests -- pt 2

```
public T remove(T r) {
    DNode<T> curr = find(r);
    if (curr == null) {
        return null;
    }
    size--;
    if (curr.prev != null)
        curr.prev.next = curr.next;
    if (curr.next != null)
        curr.next.prev = curr.prev;
    if (curr == tail)
        tail = curr.prev;
    return r;
}
```

if r is not in list:
List before/after should not change
size should not change

if r is in list:
first instance of r should be removed from list. Otherwise list should be unchanged
size reduced by one

if r is in list more than once:
Only the first instance of r should be removed

if r is head ... FAILS
if r is tail ...
if list has exactly one item ...
if list has 2 items ...
if list has more than 2 items ...

Second Remove Test

This one fails

```
    public boolean testu() {
        System.out.println("Test of item removal from head of list (using remove)
\n-----------------------------------------");
        DoubleLinkedList<Integer> s = new DoubleLinkedList<>();
        DoubleLinkedList<Integer> t = new DoubleLinkedList<>();
        for (int i = 0; i < 5; i++) {
            s.addLast(i);
            if (i != 0)
                t.addLast(i);
        }
        System.out.format("First List : %s\n", s);
        System.out.format("Second List: %s\n", t);
        System.out.println("Now remove 0:  the lists should be identical");
        s.remove(0);
        System.out.format("First List : %s\n", s);
        System.out.format("Second List: %s\n", t);
        if (s.size() != t.size()) {
            System.out.println("Test failed: lists are not equal sizes");
            return false;
        }
        while (s.size() > 0) {
            if (s.removeFirst() != t.removeFirst()) {
                System.out.println("Remove test Failed -- items are not all the
same.");
                return false;
            }
        }
        System.out.println("Remove test Passed");
        return true;
    }
```

First Remove Test

this test assumes that addLast and removeFirst have already been tested

```
    public boolean testt() {
        System.out.println("Test of item removal from middle of 
list\n------------------------------------");
        DoubleLinkedList<Integer> s = new DoubleLinkedList<>();
        DoubleLinkedList<Integer> t = new DoubleLinkedList<>();
        for (int i = 0; i < 5; i++) {
            s.addLast(i);
            if (i != 3)
                t.addLast(i);
        }
        System.out.format("First List : %s\n", s);
        System.out.format("Second List: %s\n", t);
        System.out.println("Remove 3:  the lists should be identical");
        s.remove(3);
        System.out.format("First List : %s\n", s);
        System.out.format("Second List: %s\n", t);
        if (s.size() != t.size()) {
            System.out.println("Test failed: lists are not equal sizes");
            return false;
        }
        while (s.size() > 0) {
            if (s.removeFirst() != t.removeFirst()) {
                System.out.println("Remove test Failed -- items are not all the 
same.");
                return false;
            }
        }
        System.out.println("Remove test Passed");
        return true;
    }
```

Should check both forward and backward