Doubly Linked Lists

cs151

Starting Point

```
public class SingleLinkedList<J> implements LinkedListInterface<J>
{
    protected class Node<H>
    {
        public H data;
        public Node<H> next;
        public Node(H data)
        {
           this.data = data;
           this.next = null;
        }
    }
    protected Node<J> head = null;
    protected Node<J> tail = null;
    protected int size;
```

AddFirst

```
public void addFirst(J c) {
    Node<J> newnode = new Node<>(c);
    if (head == null) {
        head = newnode;
        tail = newnode;
        size=1;
        return;
    }
    newnode.next = head;
    head = newnode;
    size++;
}
```

Queue and Stack

```
public interface QueueInterface<E> {
   int size();
   boolean isEmpty();
   E getFront();
   boolean enqueue(E e);
   E dequeue();
   void clear();
}
```

```
public interface StackIntf<E> {
   public boolean isEmpty();
   public E push(E e);
   public E peek();
   public E pop();
   public int size();
   public void clear();
}
```

Linked Lists and Queue

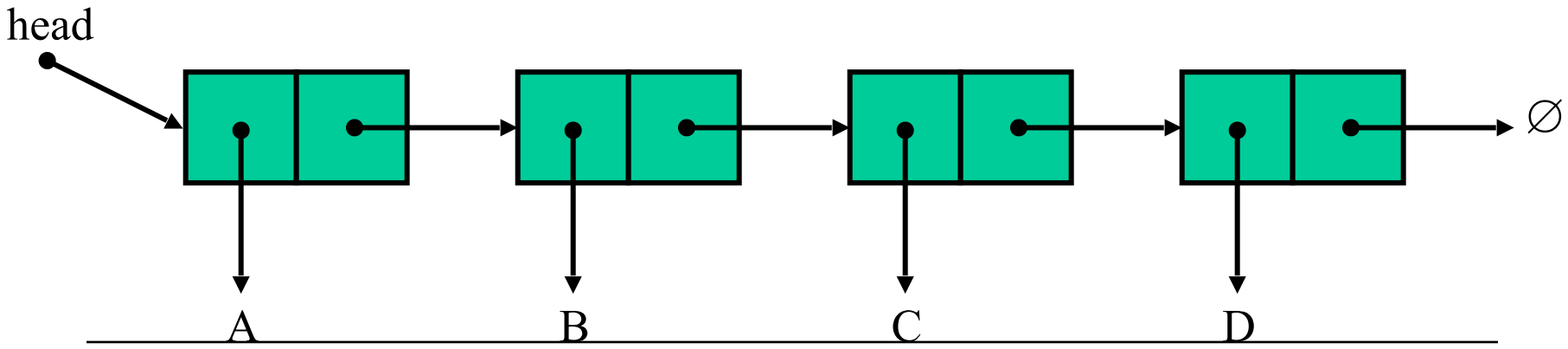
```
public class QueueLL<Q> implements QueueInterface<Q>{
    private SingleLinkedList<Q> underlyingLL;
    public QueueLL() {
        underlyingLL = new SingleLinkedList<Q>();
    }
    @Override
    public int size() {
        return underlyingLL.size();
    }
    @Override
    public boolean isEmpty() {
        return underlyingLL.isEmpty();
    }
public Q getFront() {
public boolean enqueue(Q e) {
public Q dequeue() {
public void clear() {
```

LinkedList and Stack

```
public class StackLL<S> implements StackIntf<S> {
    private SingleLinkedList<S> underlyingLL;
    public StackLL() {
        underlyingLL = new SingleLinkedList<S>();
    }
    @Override
    public boolean isEmpty() {
        return underlyingLL.isEmpty();
    }
    @Override
    public int size() {
        return underlyingLL.size();
    }
    @Override
    public void clear() {
        underlyingLL = new SingleLinkedList<S>();
    }
      @Override
      public S push(S e) {
          underlyingLL.addFirst(e);
          return e;
      }

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

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

}
```

removeLast()

- Problem
  - How do you remove the last
  - Problem, knowing the last node is not enough?
    - 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
- Happily, I already had a function to do that



next to last

```
private Node<J> nextToLastNode() {
    Node<J> n = head;
    if (n==null || n.next == null)
        return null;
    while (n.next.next != null) {
        n = n.next;
    }
    return n;
}
```

Shortcut boolean evaluation

!!!!

Remove Last

```
public J removeLast() {
    if (head == null)
        return null;
    if (head.next == null) {
        J tmp = head.data;
        head = null;
        tail=null;
        size=0;
        return tmp;
    }
    Node<J> n2last = nextToLastNode();
    J tmp = n2last.next.data;
    n2last.next = null;
    tail = n2last;
    size -= 1;
    return tmp;
}
```

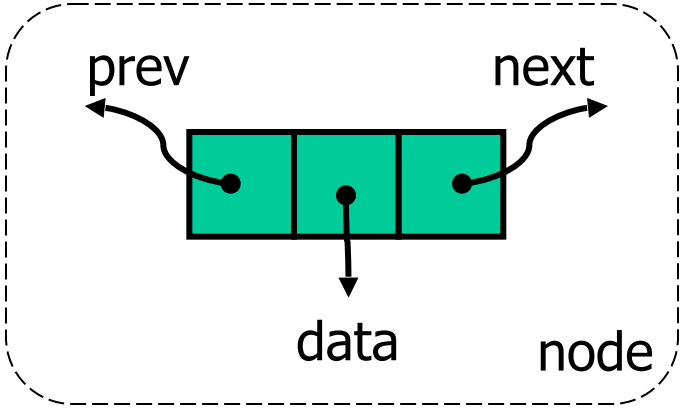
Musings on singly linked lists

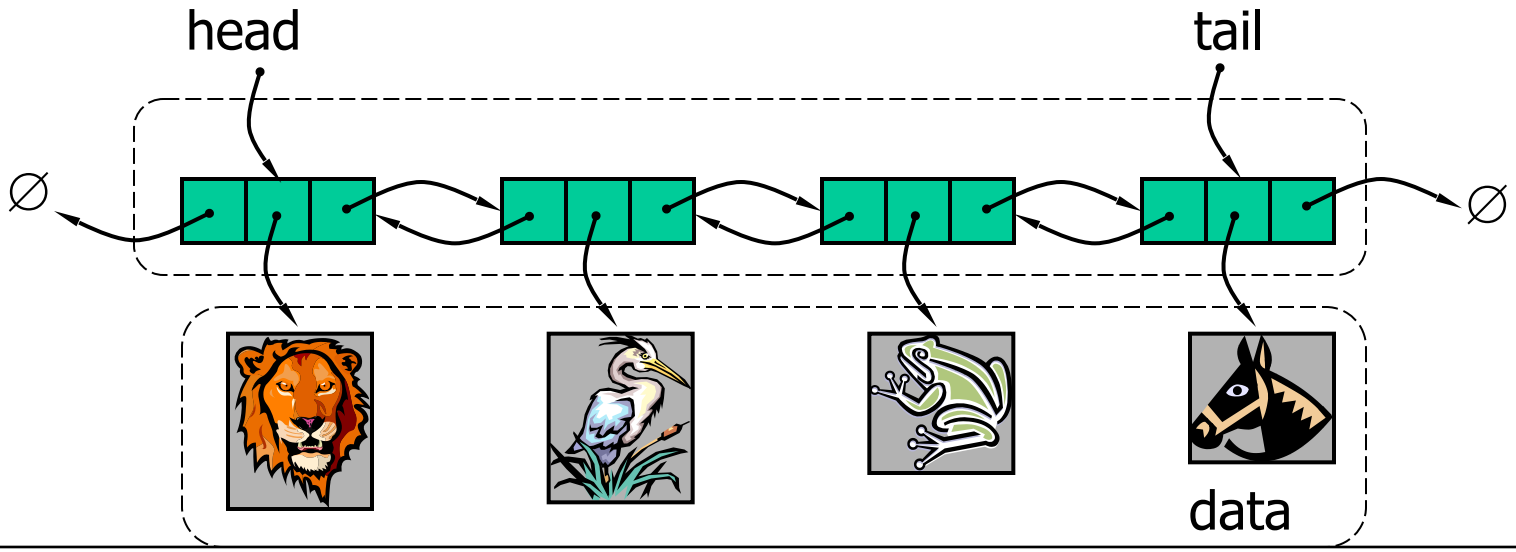
- The whole remove last method is a pain
  - and it is slow O(n)
- Not being able to go backward is a pain
  - Linked lists are a pain

- 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;
```

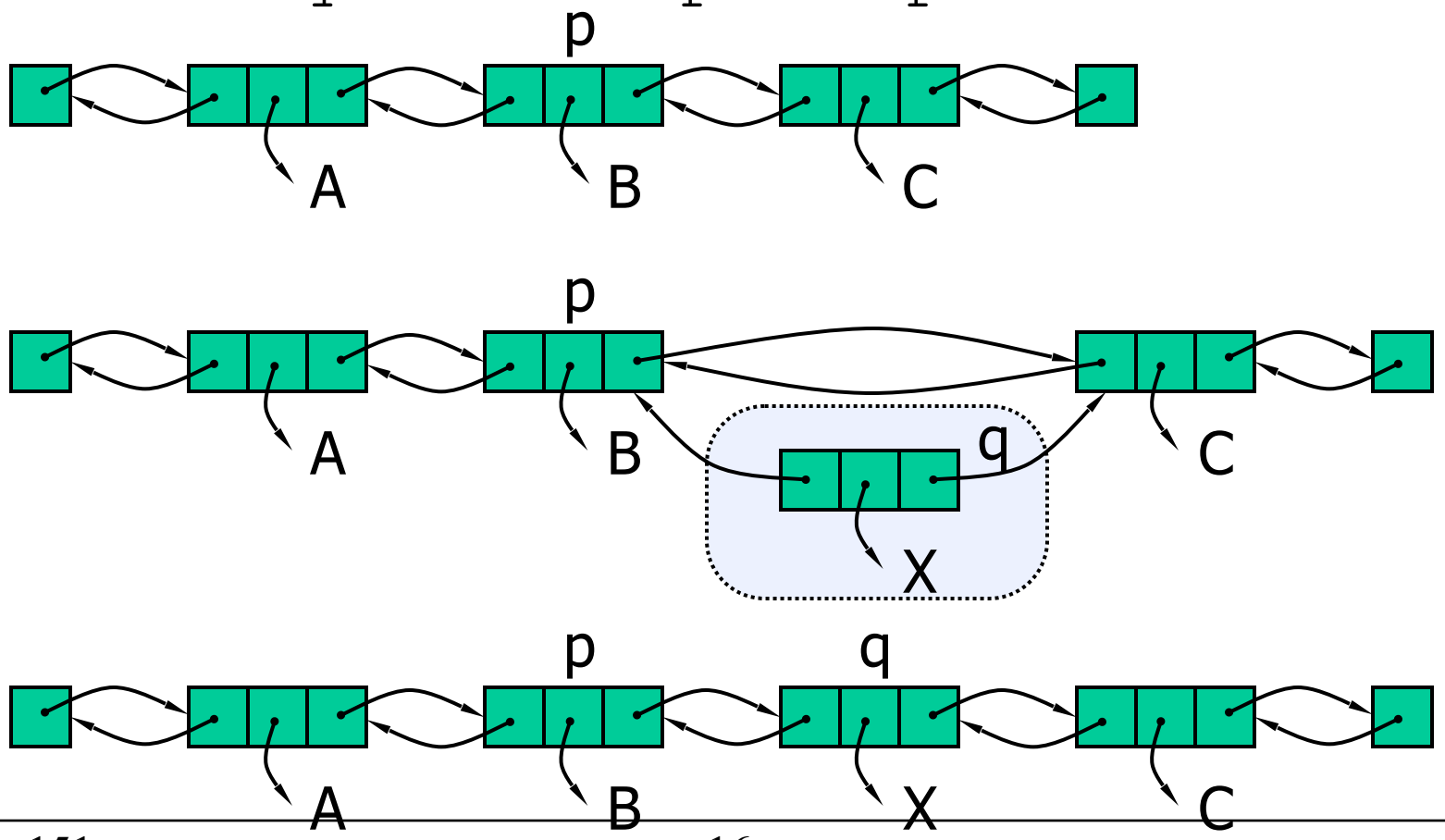
Basics

```
@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

```
public 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++;
}
```

Problems??

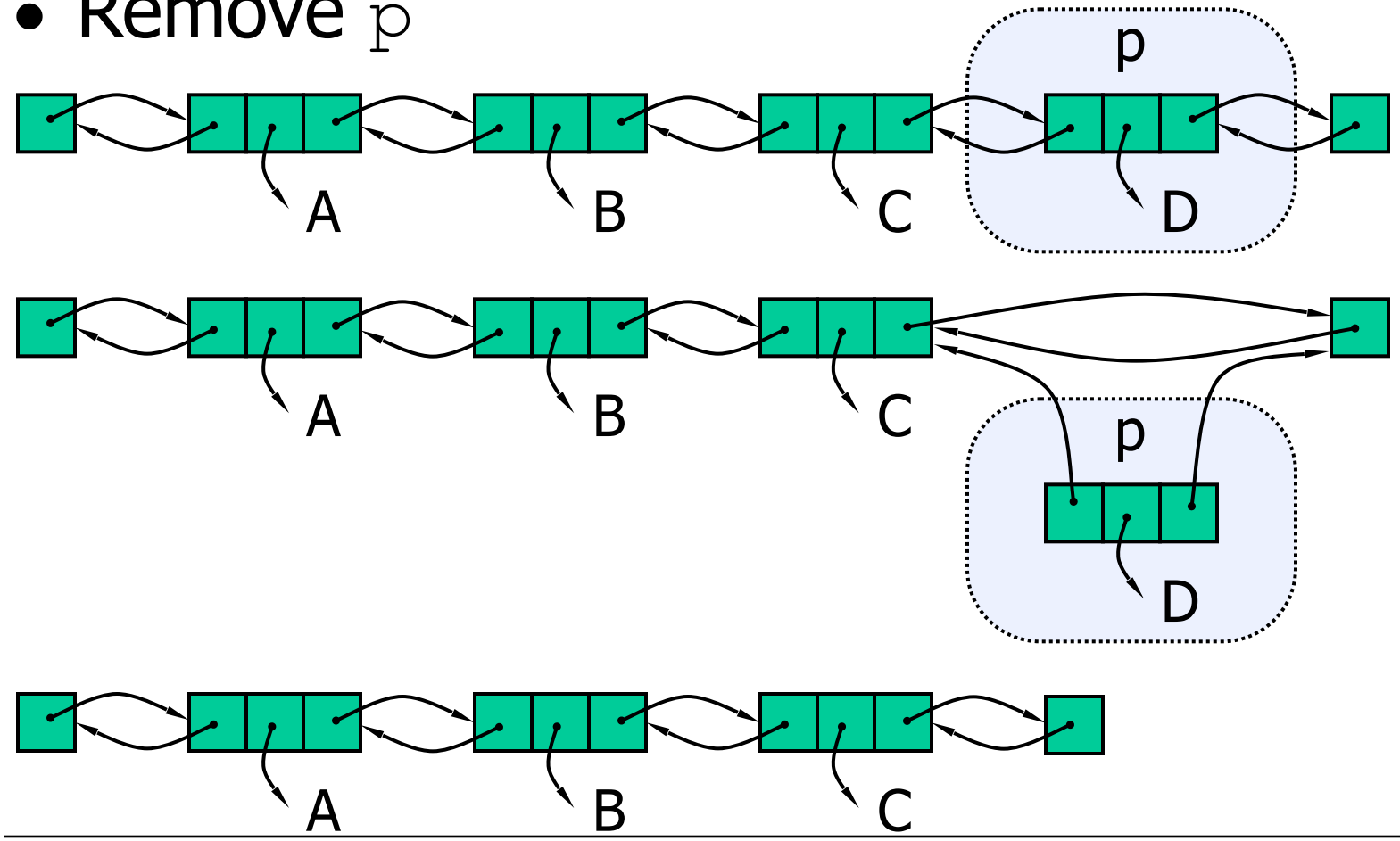
Deletion — last element

```
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

Deletion

- Remove `p`



Deletion and contains

```
    Comparable<E> remove(Comparable<E> r);
    boolean contains(Comparable<E> iD);
```

First write a private utility function

```
    private Node<T> find(Comparable<E> look4)
```

that returns a node containing look4 (or null)
then use this function in remove and contains