CS 600.226: Data Structures

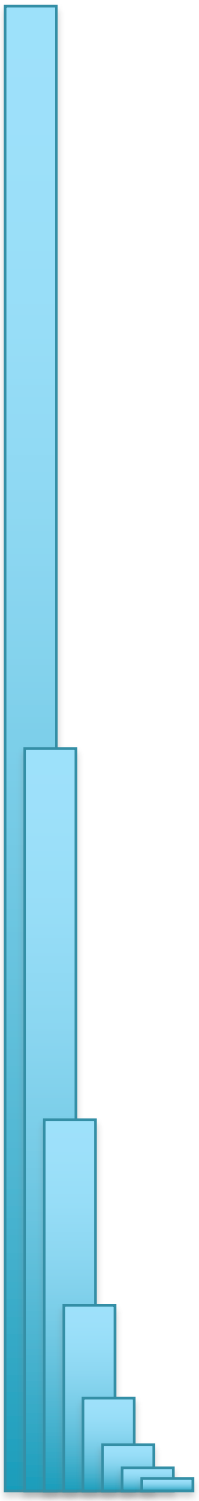
Michael Schatz

Sept 26 2018

Lecture 12. Lists

Agenda

1. *Updates on HW3*
2. *Recap on Stacks, Queues, and Deques*
3. *Lists*



Assignment 3: Due Friday Sept 28 @ 10pm

https://github.com/schatzlab/datastructures2018/blob/master/assignments/assignment03/README.md

Assignment 3: Assorted Complexities

Out on: September 21, 2018
Due by: September 28, 2018 before 10:00 pm
Collaboration: None
Grading:

Functionality 60% (where applicable)
Solution Design and README 10% (where applicable)
Style 10% (where applicable)
Testing 10% (where applicable)

Overview

The third assignment is mostly about sorting and how fast things go. You will also write yet another implementation of the Array interface to help you analyze how many array operations various sorting algorithms perform.

Note: The grading criteria now include 10% for unit testing. This refers to JUnit 4 test drivers, not some custom test program you hacked. The problems (on this and future assignments) will state whether you are expected to produce/improve test drivers or not.

Assignment 3: Due Friday Sept 28 @ 10pm

https://github.com/schatzlab/datastructures2018/blob/master/assignments/assignment03/README.md

Problem 1: Arrays with Statistics (30%)

Your first task for this assignment is to develop a new kind of `Array` implementation that keeps track of how many read and write operations have been performed on it. Check out the `Statable` interface first, reproduced here in compressed form (be sure to use and read the full interface available in github):

```
public interface Statable {
    void resetStatistics();
    int numberOfReads();
    int numberOfWrites();
}
```

This describes what we expect of an object that can collect statistics about itself. After a `Statable` object has been "in use" for a while, we can check how many read and write operations it has been asked to perform. We can also tell it to "forget" what has happened before and start counting both kinds of operations from zero again.

Make sure to update: length(), get(), and put(); skip the iterator

Assignment 3: Due Friday Sept 28 @ 10pm

https://github.com/schatzlab/datastructures2018/blob/master/assignments/assignment03/README.md

Problem 2: All Sorts of Sorts (50%)

You need to write classes implementing BubbleSort and InsertionSort for this problem. Just like our example algorithms, your classes have to implement the SortingAlgorithm interface.

All of this should be fairly straightforward once you get used to the framework. Speaking of the framework, the way you actually "run" the various algorithms is by using the PolySort.java program we've provided as well. You should be able to compile and run it without yet having written any sorting code yourself.

Here's how:

```
$ java PolySort 4000 <random.data

 Algorithm            Sorted?  Size         Reads          Writes         Seconds

 Null Sort            false    4,000        0              0              0.000007
 Gnome Sort           true     4,000        32,195,307     8,045,828      0.243852
 Selection Sort       true     4,000        24,009,991     7,992          0.252085
```

The running times can be a clue, but write up should reflect on more than time

Assignment 3: Due Friday Sept 28 @ 10pm

https://github.com/schatzlab/datastructures2018/blob/master/assignments/assignment03/README.md

Problem 3: Analysis of Selection Sort (20%)

Your final task for this assignment is to analyze the following selection sort algorithm theoretically (without running it) in detail (without using O-notation).

Here's the code, and you must analyze exactly this code (the line numbers are given so you can refer to them in your writeup for this problem):

```
 1: public static void selectionSort(int[] a) {
 2:     for (int i = 0; i < a.length - 1; i++) {
 3:         int min = i;
 4:         for (int j = i + 1; j < a.length; j++) {
 5:             if (a[j] < a[min]) {
 6:                 min = j;
 7:             }
 8:         }
 9:         int t = a[i]; a[i] = a[min]; a[min] = t;
10:     }
11: }
```

Work slowly, line-by-line analysis

TestSimpleArray.java

```
import org.junit.Test;
import org.junit.BeforeClass;
import static org.junit.Assert.assertEquals;

public class TestSimpleArray {
    static Array<String> shortArray;

    @BeforeClass
    public static void setupArray() throws LengthException {
       shortArray = new SimpleArray<String>(10, "Bla");
    }

    @Test
    public void newArrayLengthGood() throws LengthException {
        assertEquals(10, shortArray.length());
    }

    @Test
    public void newArrayInitialized() throws LengthException, IndexException {
        for (int i = 0; i < shortArray.length(); i++) {
            assertEquals("Bla", shortArray.get(i));
        }
    }

    @Test(expected=IndexException.class)
    public void IndexDetected() throws IndexException {
        shortArray.put(shortArray.length(), "Paul");
    }
}
```

@BeforeClass causes the method to be run once before any of the test methods in the class

Check the results with assertEquals, or listing the expected exception

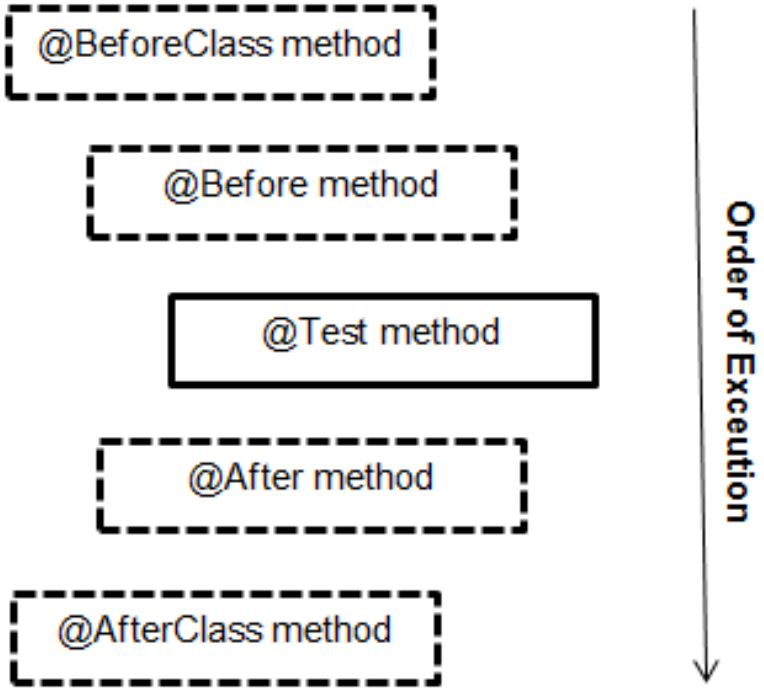
@Before vs @BeforeClass

@BeforeClass

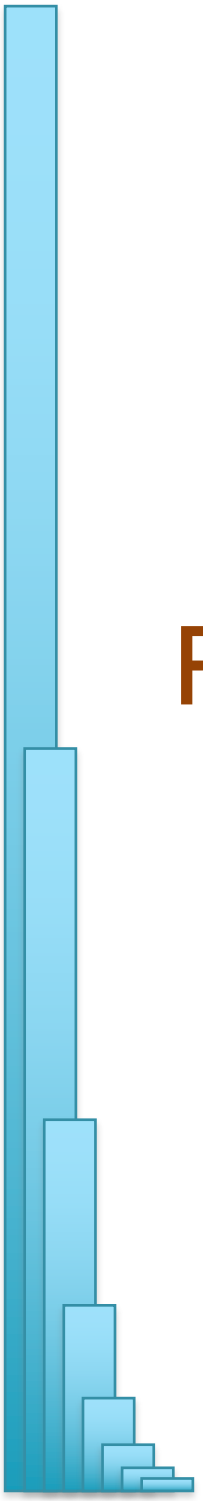
- Execute before ***all*** test methods of the class are executed.
- Used with static methods
- For example, This method could contain some initialization code

@Before

- Execute before ***each*** test method.
- Used with non-static method.
- For example, to reinitialize some class attributes used by the methods.



- @BeforeClass: Static initialization of an expensive database connection
- @Before: Non-static initialize a datastructure
- @Test: tests in arbitrary order
- @After: non-static cleanup
- @AfterClass: static cleanup



Part 1: Stacks, Queues, and Deques

Stacks versus Queues

LIFO: Last-In-First-Out
Add to top +
Remove from top

FIFO: First-In-First-Out
Add to back +
Remove from front

Stacks versus Queues

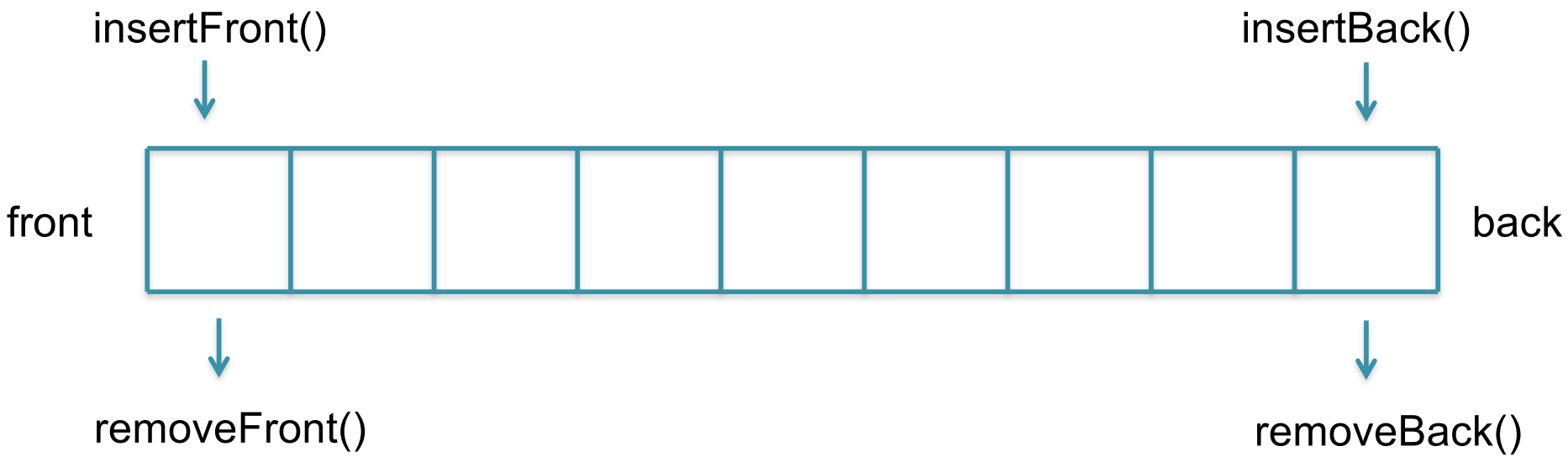
LIFO: Last-

Add to top +
Remove from top

-In-First-Out

Add to back +
Remove from front

Dequeues



Dynamic Data Structure used for storing sequences of data

- Insert/Remove at either end in O(1)
- If you exclusively add/remove at one end, then ***it becomes a stack***
- If you exclusive add to one end and remove from other, then ***it becomes a queue***
- Many other applications:
  - browser history: deque of last 100 webpages visited

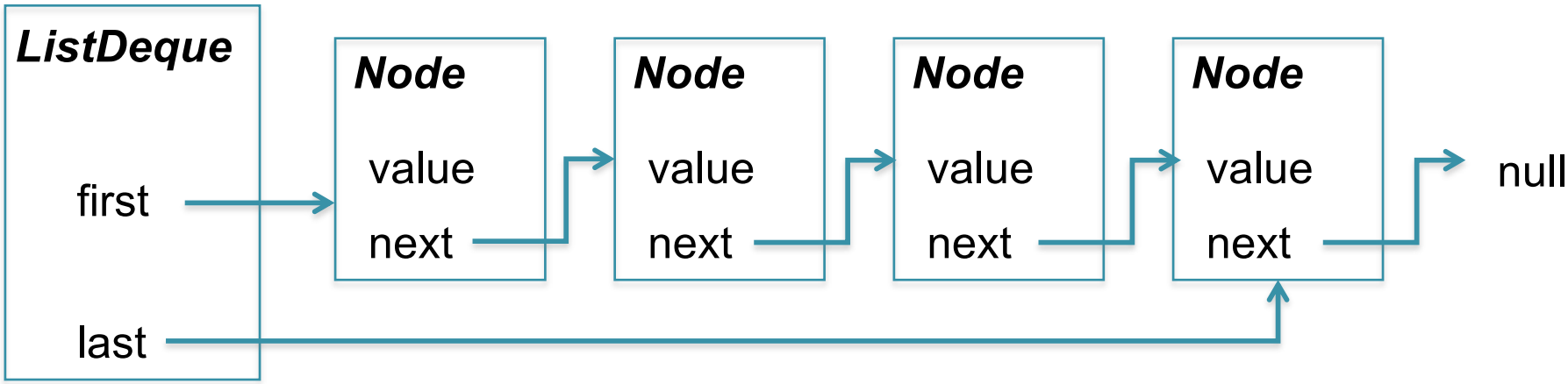
Dequeue Interface

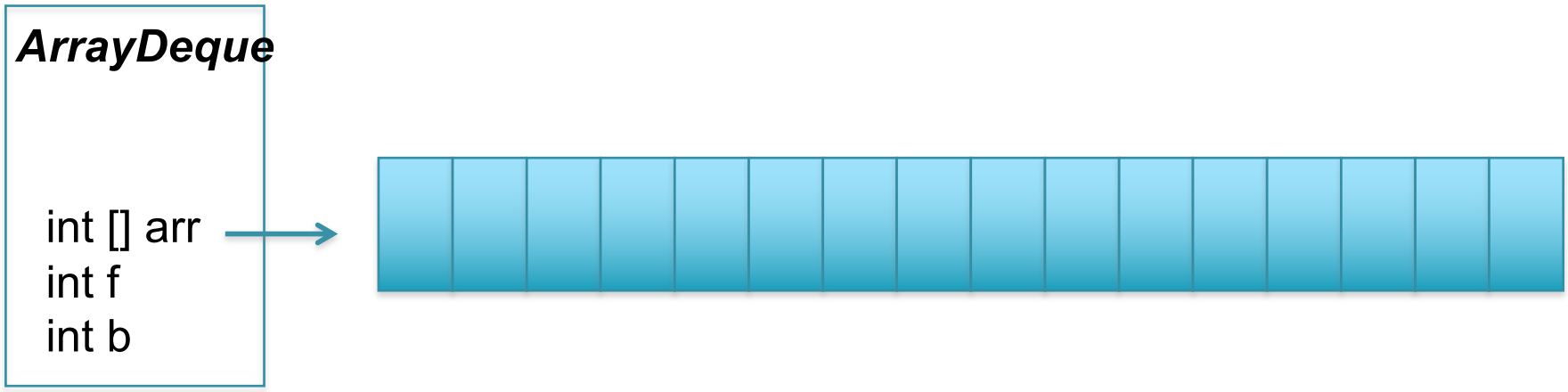
```
public interface Dequeue<T> {
    boolean empty();
    int length();

    T front() throws EmptyException;
    T back() throws EmptyException;

    void insertFront(T t);
    void insertBack(T t);

    void removeFront() throws EmptyException;
    void removeBack() throws EmptyException;
}
```

How would you implement the underlying storage?

Why?

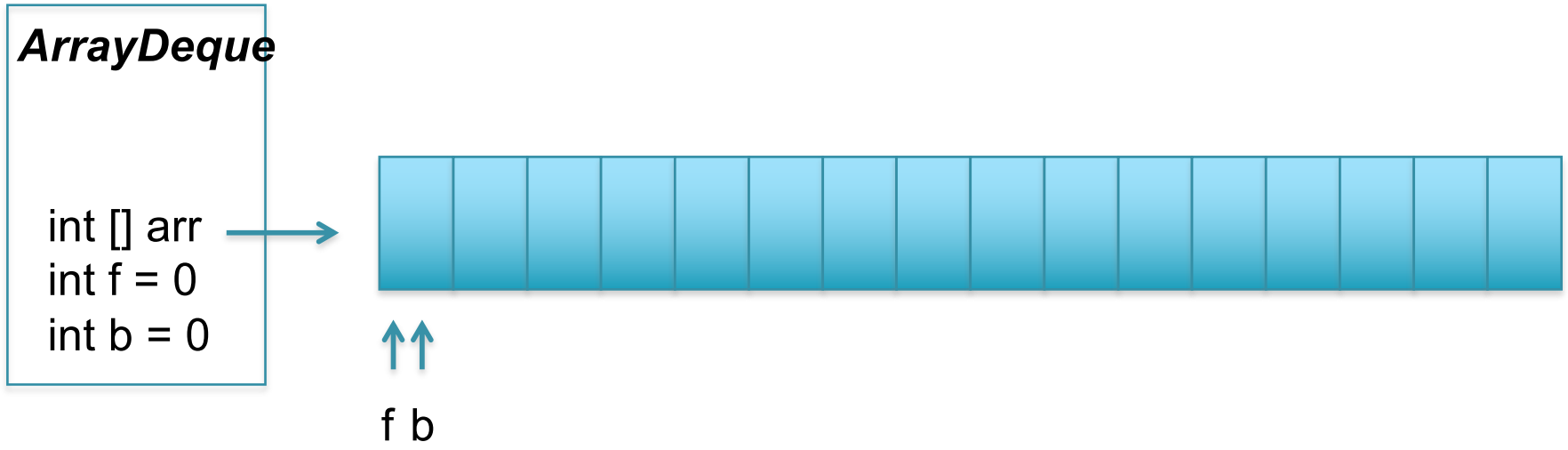
ArrayDequeue





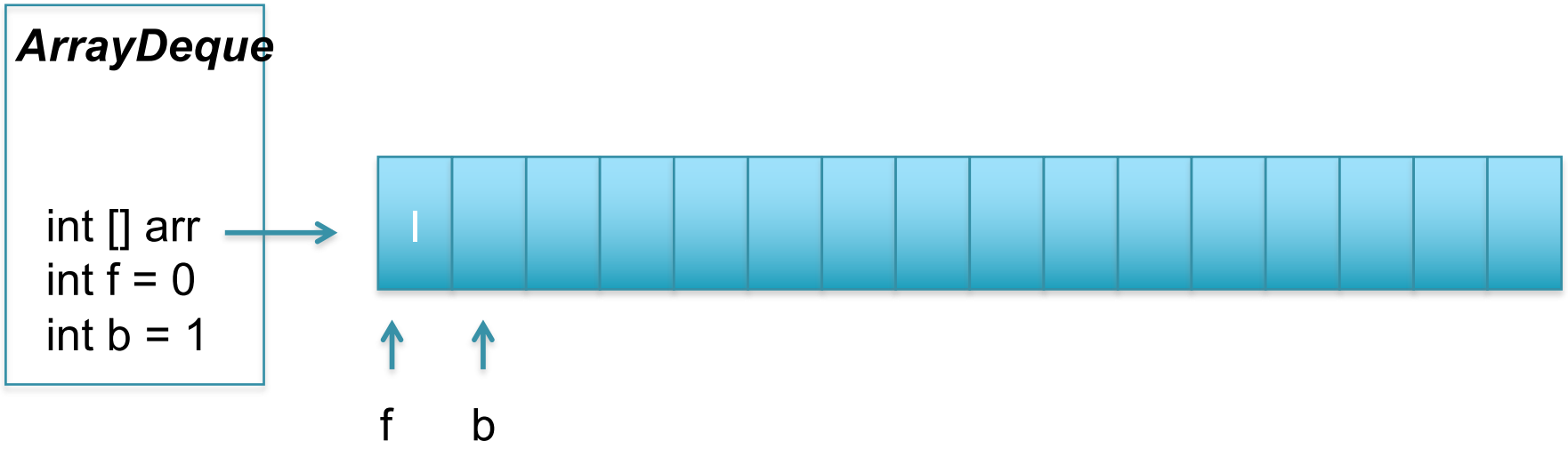
Many of the same tradeoffs as ListQueue vs ArrayQueue

ArrayDequeue



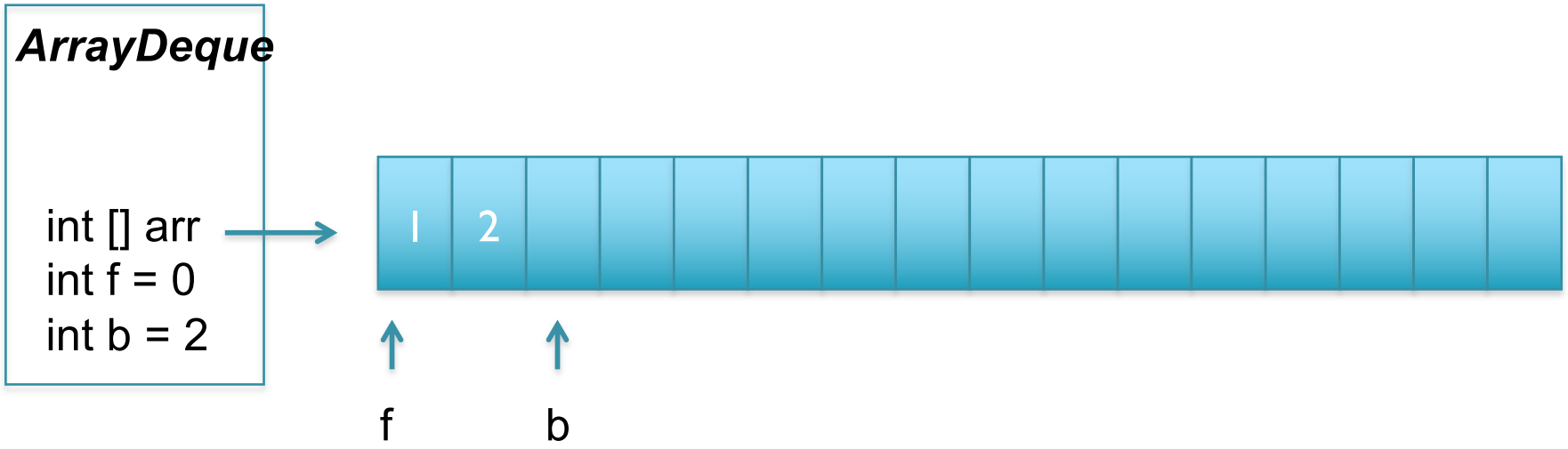
deque = new ArrayDequeue();

ArrayDequeue



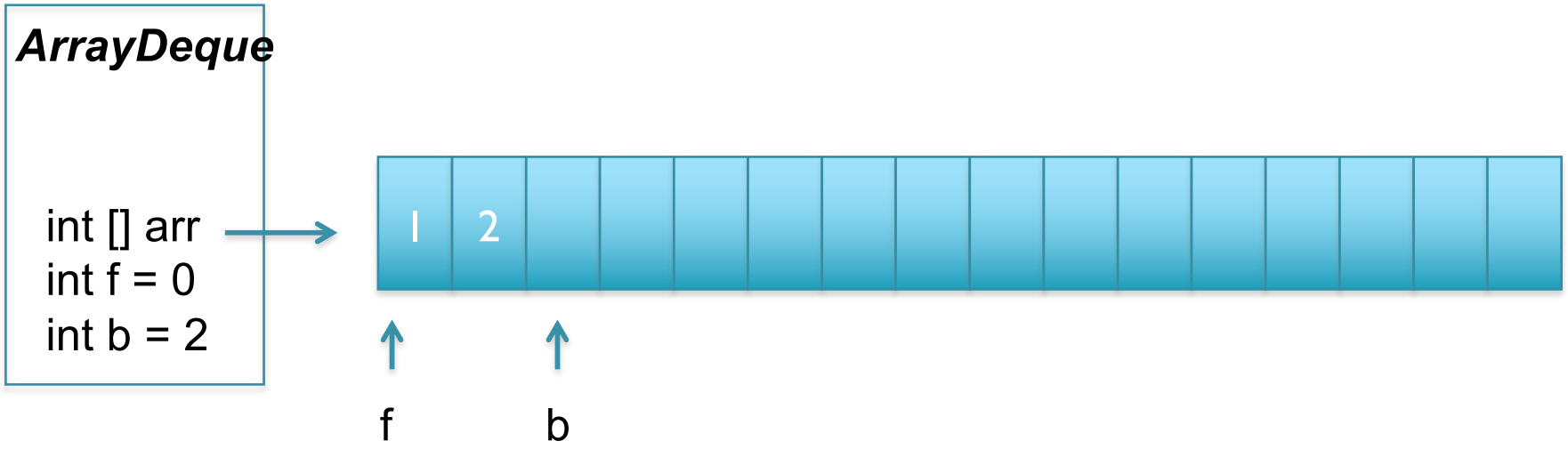

```
deque = new ArrayDequeue();
deque.insertBack(1);
```

ArrayDequeue



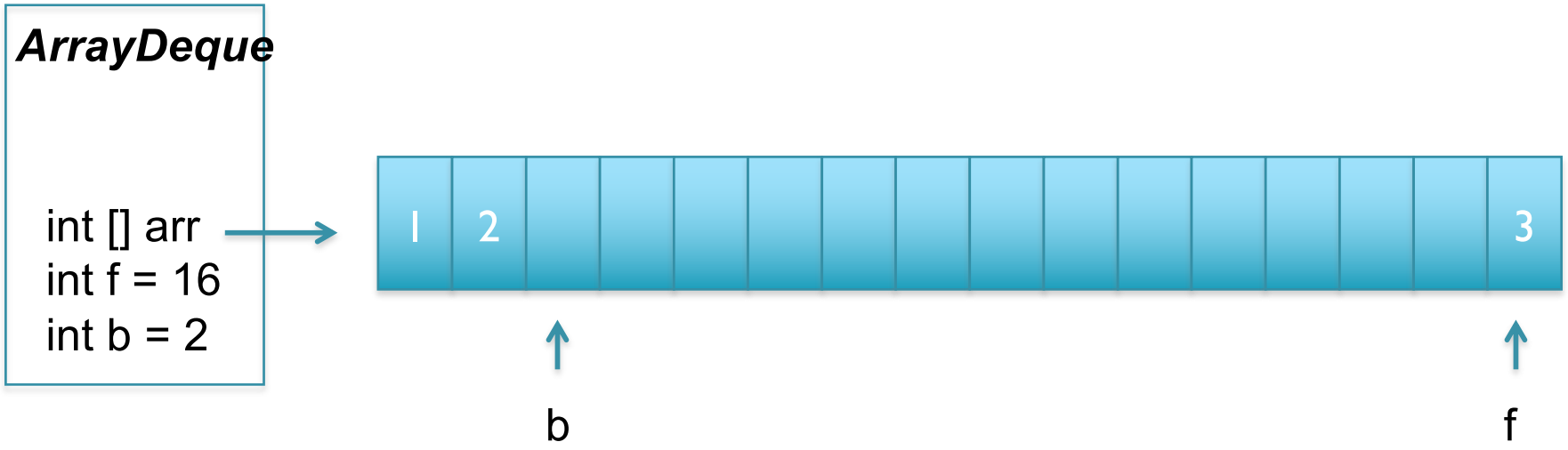

```
deque = new ArrayDequeue();
deque.insertBack(1);
deque.insertBack(2);
```

ArrayDequeue




```
deque = new ArrayDequeue();
deque.insertBack(1);
deque.insertBack(2);
deque.insertFront(3);
```

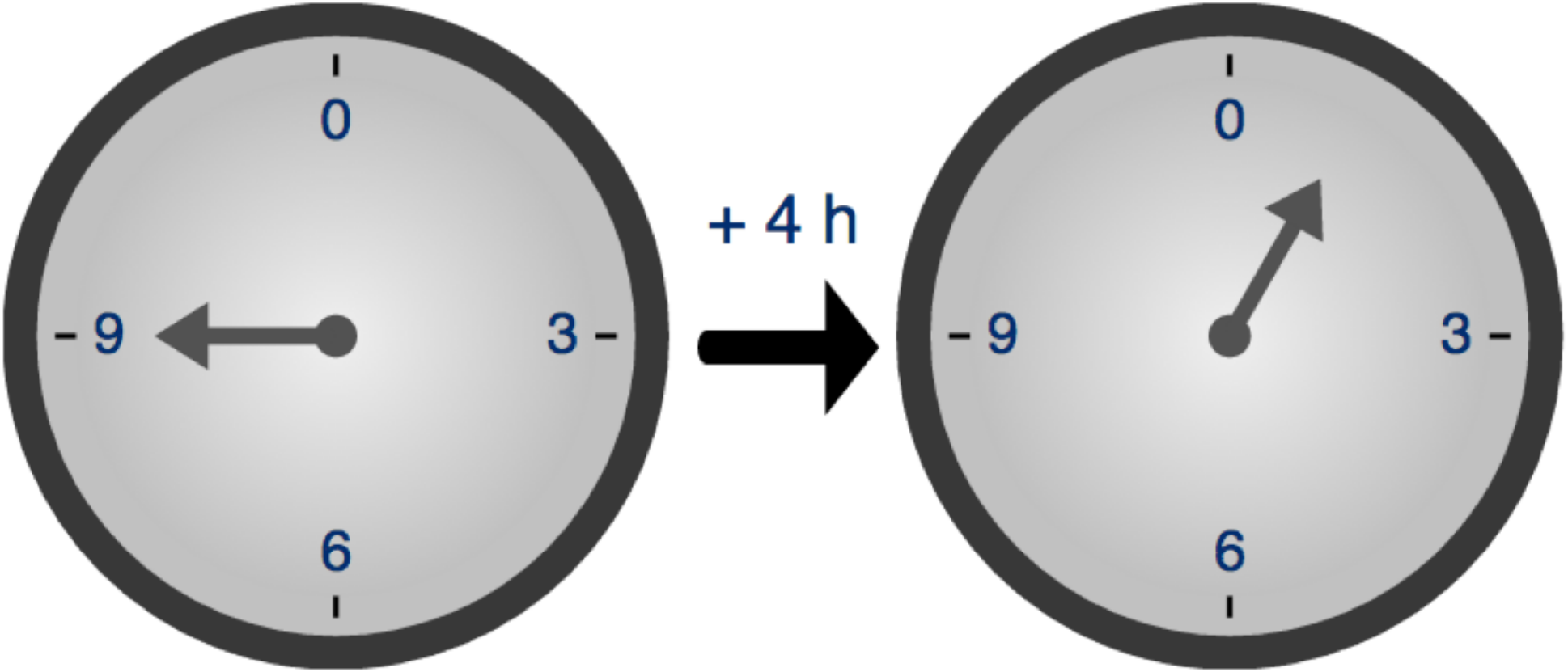
ArrayDequeue




```
deque = new ArrayDequeue();
deque.insertBack(1);
deque.insertBack(2);
deque.insertFront(3);
```

Modular Arithmetic

m = a % b means to set m to be the remainder when dividing a by b
m is guaranteed to fall between 0 and b-1



9 + 4 % 12 = ?

9 + 4 % 12 = 1
13 / 12 = 1 remainder 1

9 + 28 % 12 = ?

9 + 28 % 12 = 1
37 % 12 = 3 remainder 1

Modular Arithmetic

m = a % b means to set m to be the remainder when dividing a by b
m is guaranteed to fall between 0 and b-1

Mod.java

```
public class Mod {
  public static void main(String [] args){
    System.out.println("i\ti%2\ti%5\ti%10\ti%16");
    for (int i = 0; i < 20; i++) {
      System.out.println(i + "\t" +
                         i % 2 + "\t" +
                         i % 5 + "\t" +
                         i % 10 + "\t" +
                         i % 16);
    }
  }
}
```

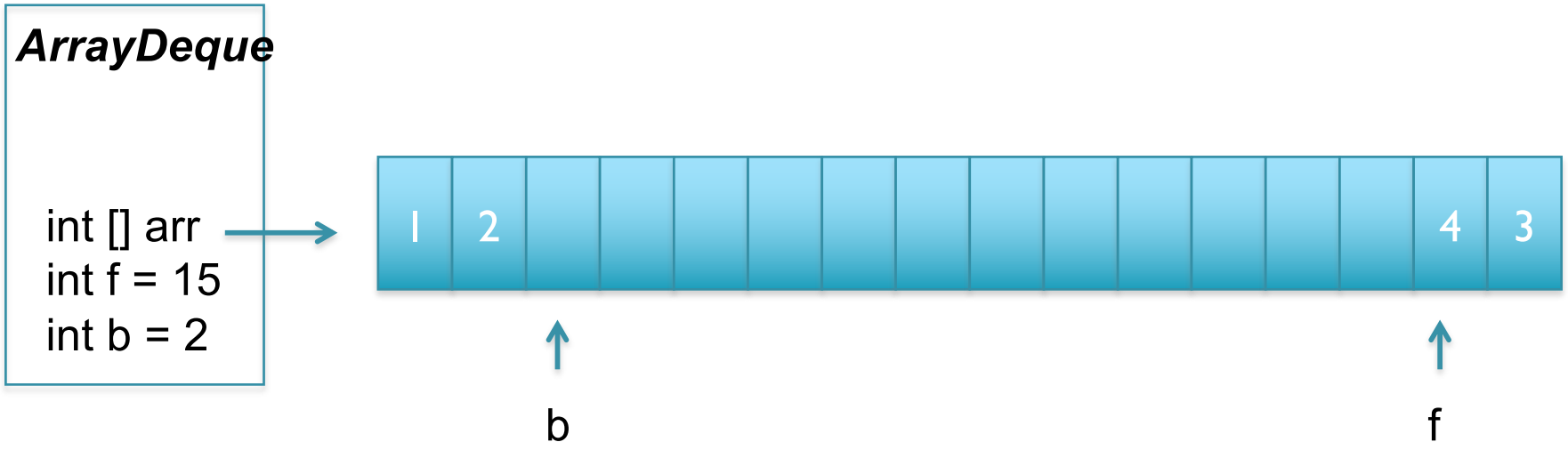
back = (back + 1) % arr.length

How do we compute length or know when it is full?

Use a separate counter. When array is totally full, double the size and copy into new array starting at 0

```
$ java Mod
i       i%2     i%5     i%10    i%16
0       0       0       0       0
1       1       1       1       1
2       0       2       2       2
3       1       3       3       3
4       0       4       4       4
5       1       0       5       5
6       0       1       6       6
7       1       2       7       7
8       0       3       8       8
9       1       4       9       9
10      0       0       0       10
11      1       1       1       11
12      0       2       2       12
13      1       3       3       13
14      0       4       4       14
15      1       0       5       15
16      0       1       6       0
17      1       2       7       1
18      0       3       8       2
19      1       4       9       3
```

ArrayDequeue



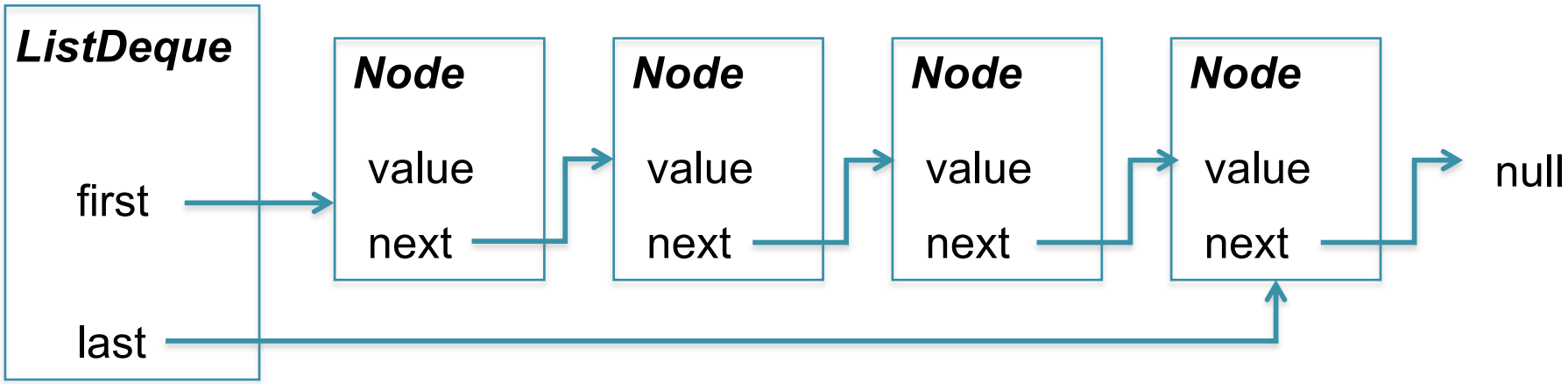
Inserting at front usually means subtract, but gets tricky when we wrap around.

f = (f-1) % arr.length; // depends on how this is implemented for negative numbers

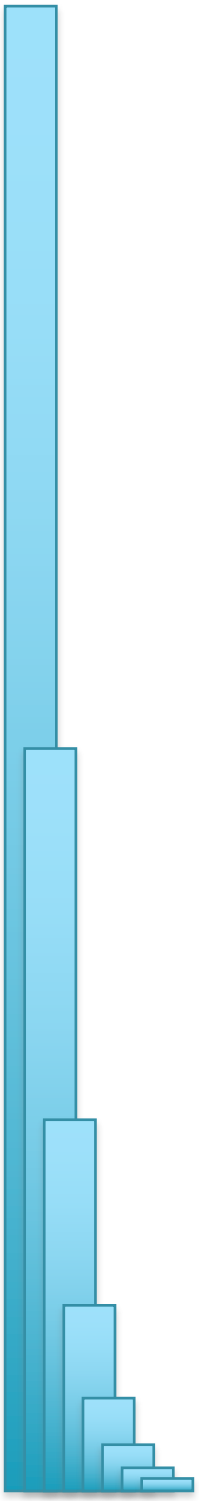
f = (f -1+arr.length) % arr.length; // does the right thing

```
deque = new ArrayDequeue();
deque.insertBack(1);
deque.insertBack(2);
deque.insertFront(3);
deque.insertFront(4);
```

ListDequeue

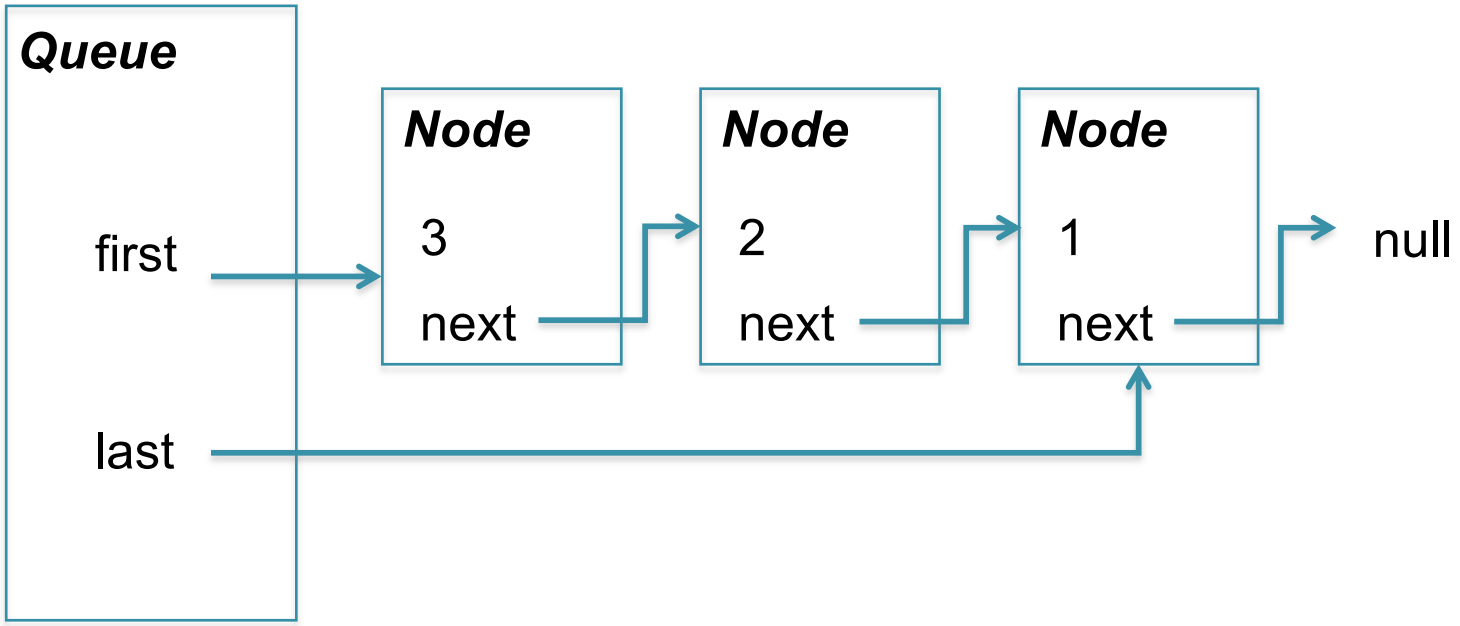


This wont work as shown



Part 2: Lists

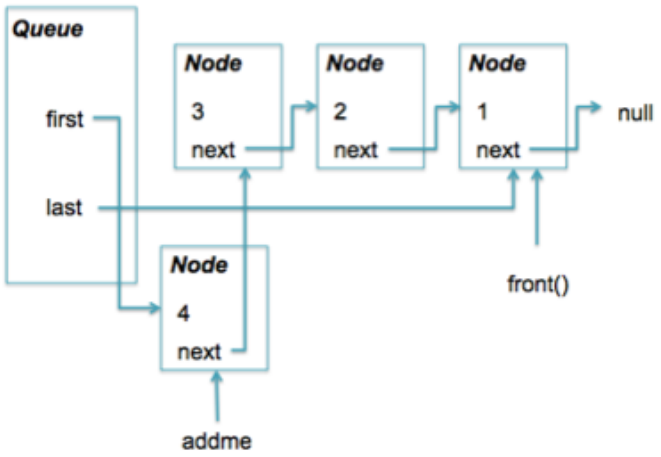
List Queue



Where should we enqueue?
Is the next node to dequeue 1 or 3?

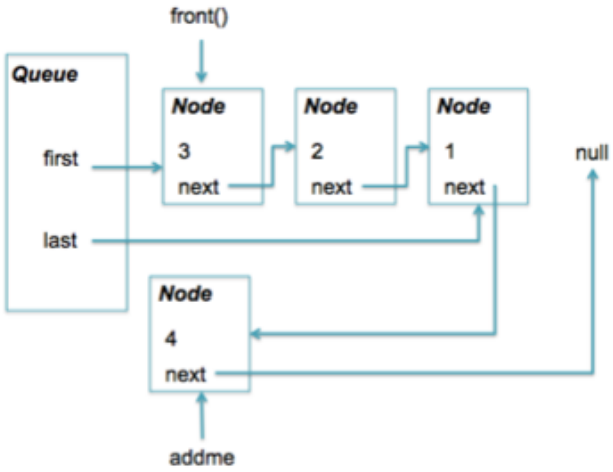
List Queue

insertFront



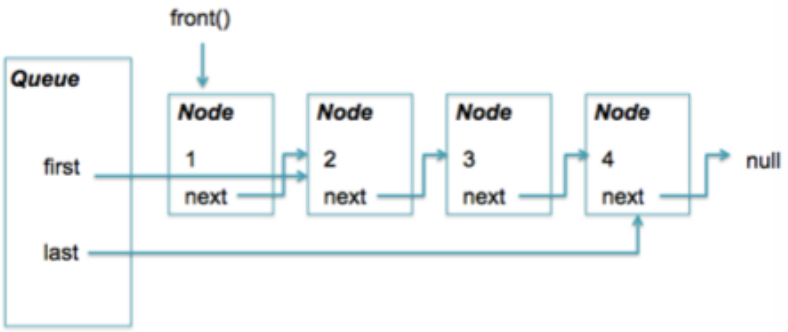
addme.next = first; first = addme;

insertBack



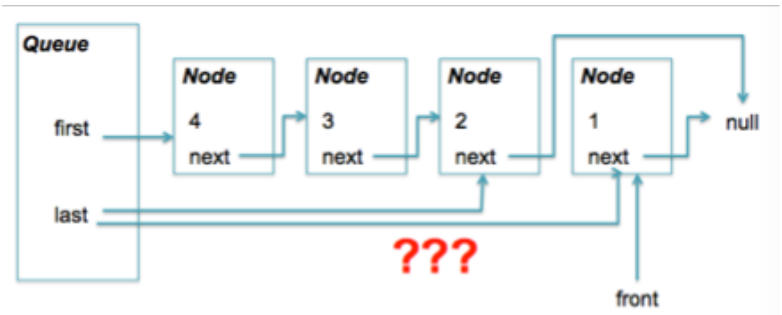
last.next = addme; addme.next = null

removeFront



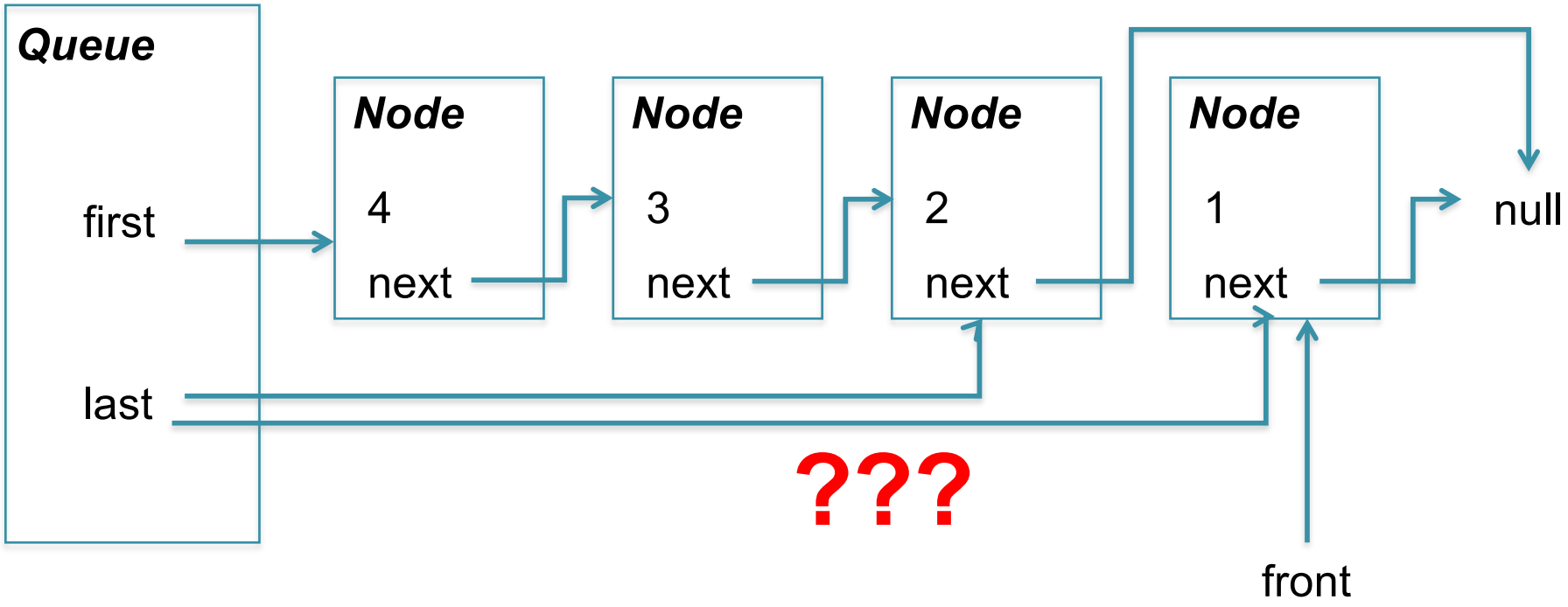
first = first.next;

removeBack



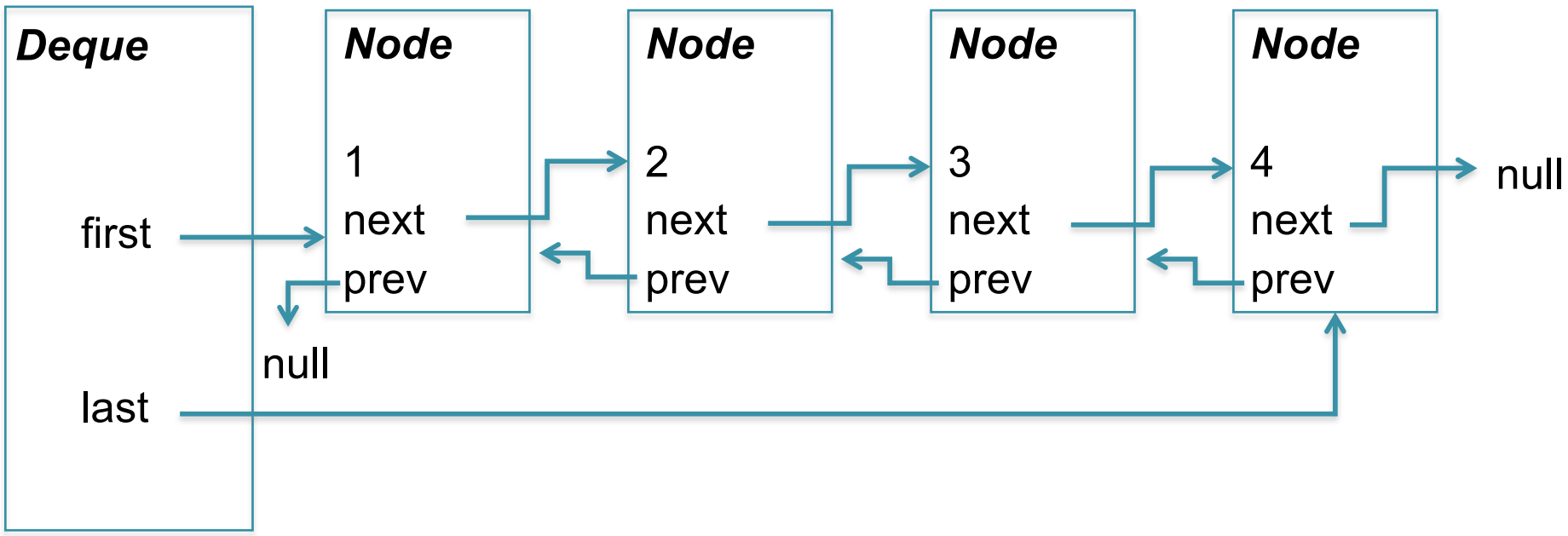
???

removeLast
(Singly Linked List)



Oops, just made remove an O(n) operation
How might you address this?

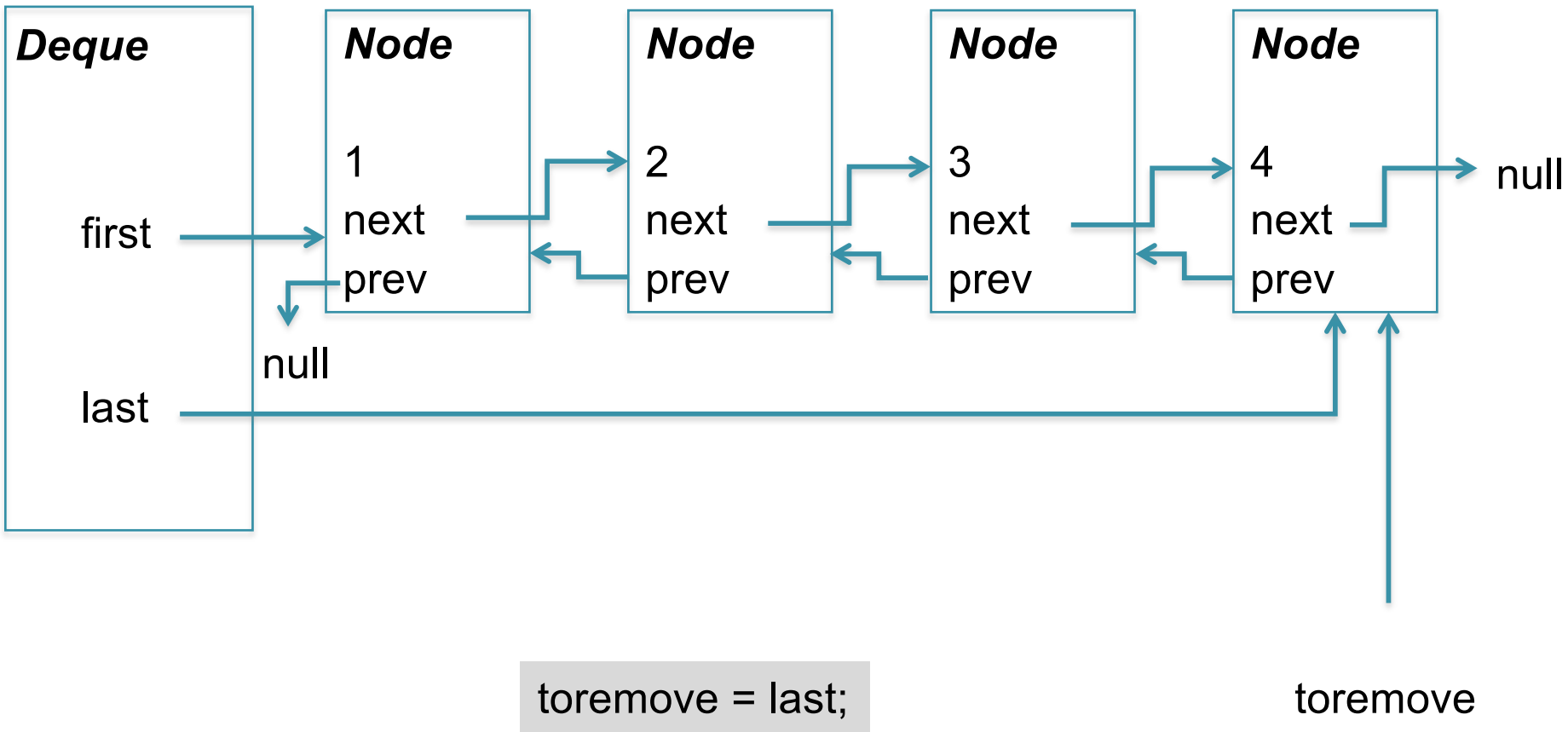
Deque with Doubly Linked List



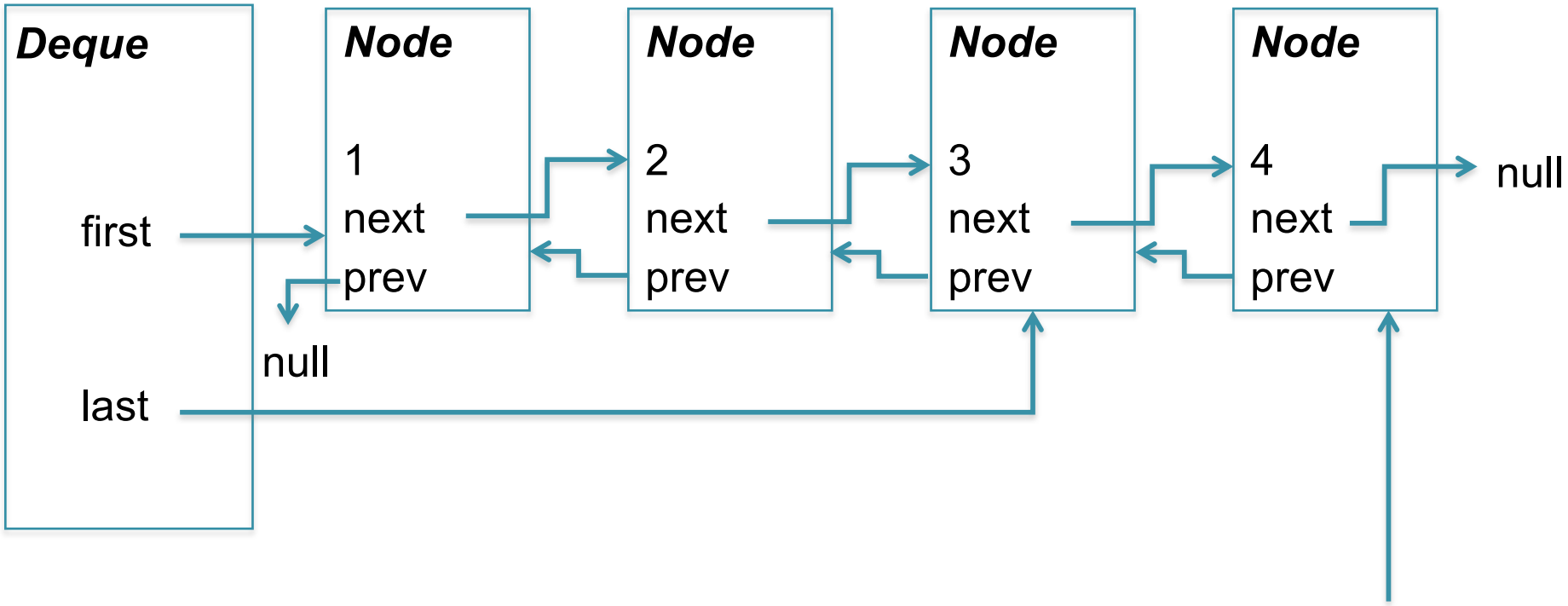
Very similar to a singly linked list, except each node has a reference to both the next and previous node in the list

A little more overhead, but significantly increased flexibility: supports insertFront(), insertBack(), removeFront(), removeBack(), insertBefore(), removeMiddle()

removeLast


Deque
first
last
Node
1
next
prev
Node
2
next
prev
Node
3
next
prev
Node
4
next
prev
null
null
toremove = last;
toremove

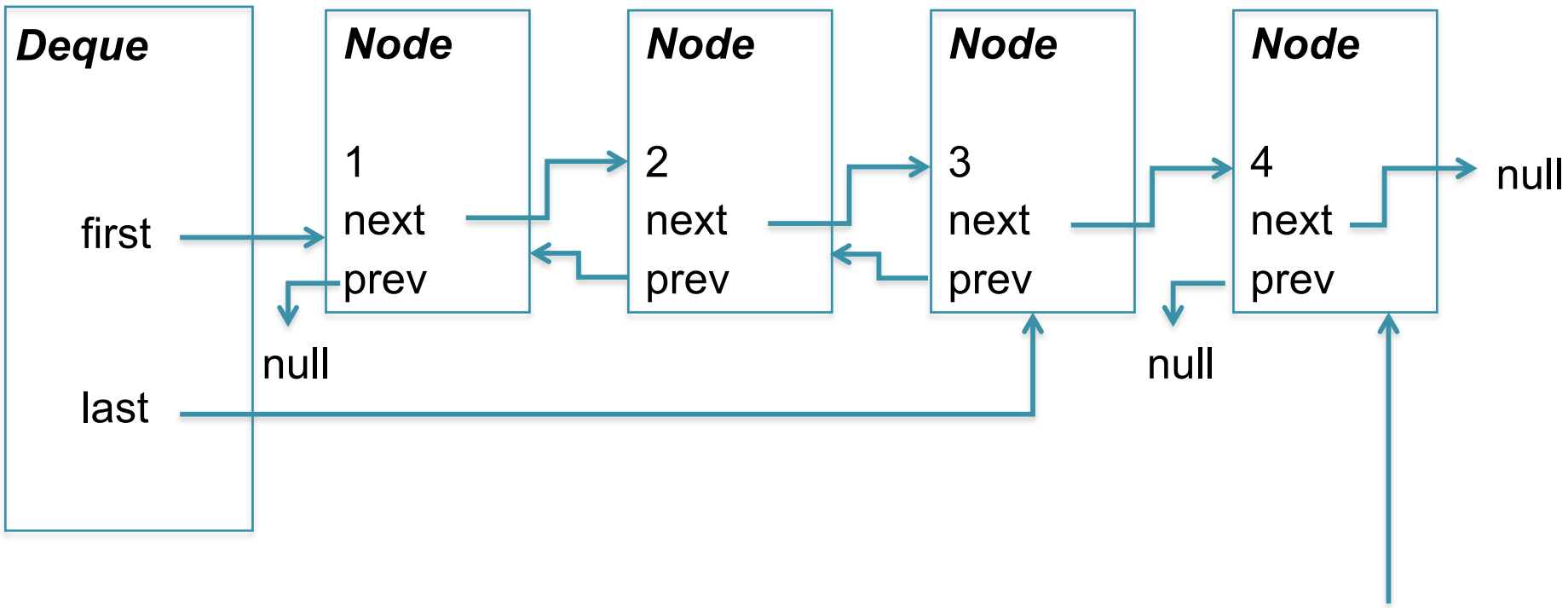
removeLast




```
toremove = last;
last = last.prev;
```

toremove

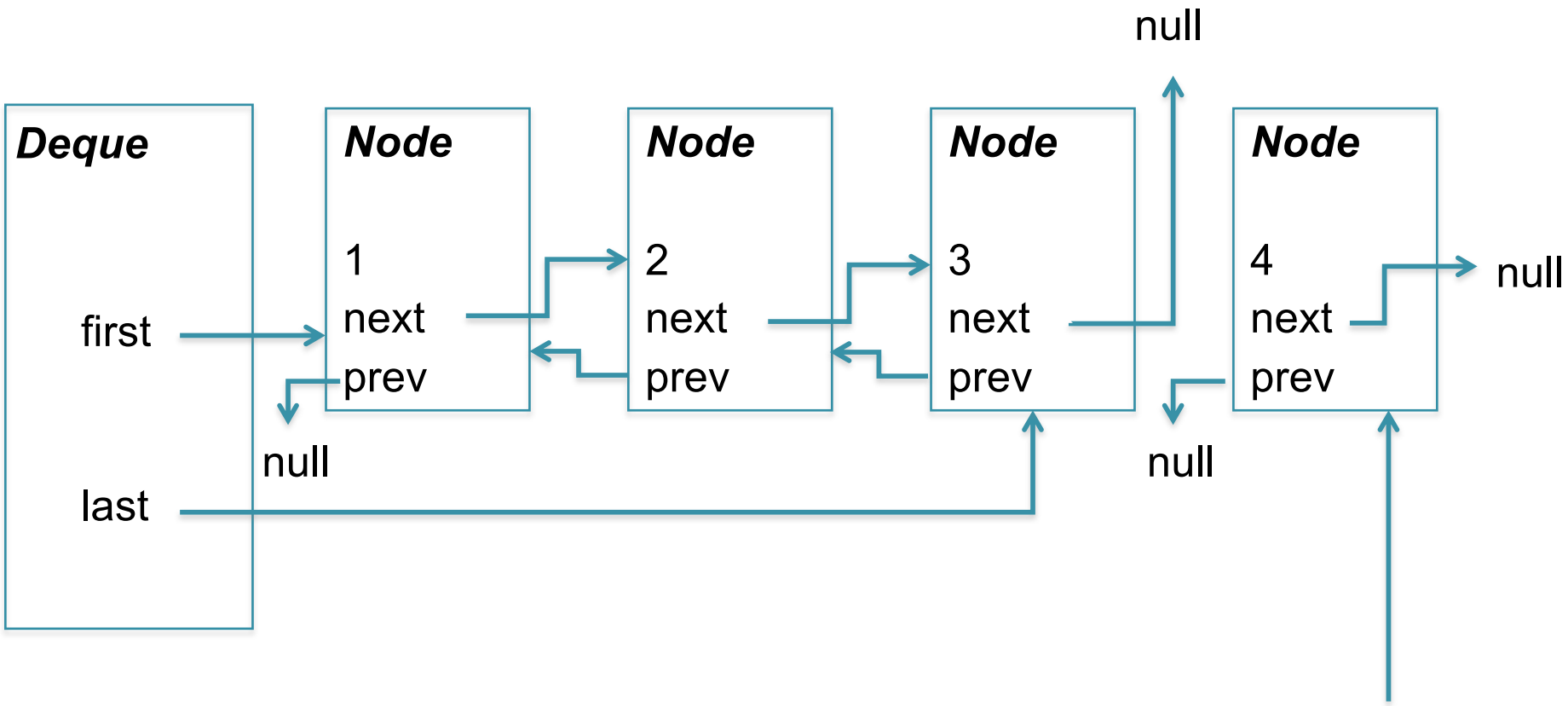
removeLast




```
toremove = last;
last = last.prev;
toremove.prev = null;
```

toremove

removeLast




```
toremove = last;
last = last.prev;
toremove.prev = null;
last.next = null;
```

O(1) removeLast() ☺

Dequeue Interface

```
public interface Dequeue<T> {
    boolean empty();
    int length();

    T front();
    T back();

    void insertFront(T t);
    void insertBack(T t);

    void removeFront();
    void removeBack();
}
```

```
public class MyDeque implements
                          Dequeue<T> {
    private static class Node<T>{
        T data
        Node<T> next;
        Node<T> prev;
    }

    private Node<T> front;
    private Node<T> back;

    T front() { return front.data; }
    T back()  { return back.data; }

     ...
}
```

How would you implement a deque with a doubly linked list?

How would you implement a general List<T> class?

List v1

```
public interface Node<T> {
     void setValue(T t);
     T getValue();

     void setNext(Node<T> n);
     void setPrev(Node<T> n);

     void getNext(Node<T> n);
     void getPrev(Node<T> n);
}

public interface List<T> {
     boolean empty();
     int length();

     Node<T> front() throws EmptyException;
     Node<T> back() throws EmptyException;

     void insertFront(Node<T> t);
     void insertBack(Node<T> t);
     void insertBefore(Node<T> t);
     void insertAfter(Node<T> t);

     void removeFront() throws EmptyException;
     void removeBack() throws EmptyException;
     void removeNode(Node<T> t);
}
```

Is this a good design?

No! We expose that Lists use Nodes for underlying storage. Worse, client programs may incorrectly edit the next/prev references

List v2

```
public class MyList<T> implements List<T> {
    private static class Node<T> {
        T data;
        Node<T> next;
        Node<T> prev;
    }

    boolean empty() { ... }
    int length() { ... }

    T front() throws EmptyException { ... }
    T back() throws EmptyException { ... }

    void insertFront(T t) { ... }
    void insertBack(T t) { ... }
    void insertBefore(???) { ... }
    void insertAfter(???) { ... }

    void removeFront() throws EmptyException { ... }
    void removeBack() throws EmptyException { ... }
    void removeNode(???) { ... }
}
```

Some implementations will have public node classes with private next/prev references. However they also have public setNext(), setPrev()
=> poor encapsulation!

Is this a good design?

Better, the nested class encapsulates the storage medium, but restricts what methods can be implemented to very first/last elements

List v3

```
public class MyList<T> implements List<T> {
    private static class Node<T> {
        T data;
        Node<T> next;
        Node<T> prev;
    }

    boolean empty() { ... }
    int length() { ... }

    T front() throws EmptyException { ... }
    T back() throws EmptyException { ... }

    void insertFront(T t) { ... }
    void insertBack(T t) { ... }
    void insertBefore(int idx) { ... }
    void insertAfter(int idx) { ... }

    void removeFront() throws EmptyException { ... }
    void removeBack() throws EmptyException { ... }
    void removeNode(int idx) { ... }
}
```

Is this a good design?

Slightly better: More flexible, but now insertBefore and removeNode() are O(n) operations

List v4

```
public interface Node<T> {
    void setValue(T t);
    T getValue();

    void setNext(Node<T> n);
    void setPrev(Node<T> n);

    void getNext(Node<T> n);
    void getPrev(Node<T> n);
}

public interface List<T> {
    boolean empty();
    int length();

    Node<T> front();
    Node<T> back();

    void insertFront(Node<T> t);
    void insertBack(Node<T> t);

    void removeFront();
    void removeBack();
}
```

List v4

```
public interface Node<T> {
    void setValue(T t);
    T getValue();

    void setNext(Node<T> n);
    void setPrev(Node<T> n);

    void getNext(Node<T> n);
    void getPrev(Node<T> n);
}

public interface List<T> {
    boolean empty();
    int length();

    Node<T> front();
    Node<T> back();

    void insertFront(Node<T> t);
    void insertBack(Node<T> t);

    void removeFront();
    void removeBack();
}
```

Nodes are useful, but is there someway to hide the internals?

List v4

```
public interface Node<T> {
    void setValue(T t);
    T getValue();

    void setNext(Node<T> n);
    void setPrev(Node<T> n);

    void getNext(Node<T> n);
    void getPrev(Node<T> n);
}

public interface List<T> {
    boolean empty();
    int length();

    Node<T> front();
    Node<T> back();

    void insertFront(Node<T> t);
    void insertBack(Node<T> t);

    void removeFront();
    void removeBack();
}
```

```
public interface Position<T> {
    // empty on purpose
}

public interface List<T> {
    // simplified interface
    int length();

    Position<T> insertFront(T t);
    Position<T> insertBack(T t);
    void insertBefore(Position<T> t);
    void insertAfter(Position<T> t);

    void removeAt(Position<T> p);
}
```

List v4

"I am a position and while you can hold on to me, you can't do anything else with me!"

```
public interface Position<T> {
    // empty on purpose
}

public interface List<T> {
    // simplified interface
    int length();

    Position<T> insertFront(T t);
    Position<T> insertBack(T t);
    void insertBefore(Position<T> t);
    void insertAfter(Position<T> t);

    void removeAt(Position<T> p);
}
```

Inserting at front or back creates the Position objects.

If you want, you could keep references to the Position objects even in the middle of the list

Pass in a Position, and it will remove it from the list

```
    void getNext(Node<T> n);
    void getPrev(Node<T> n);

    t);
    );

    void removeFront();
    void removeBack();
}
```

List v4

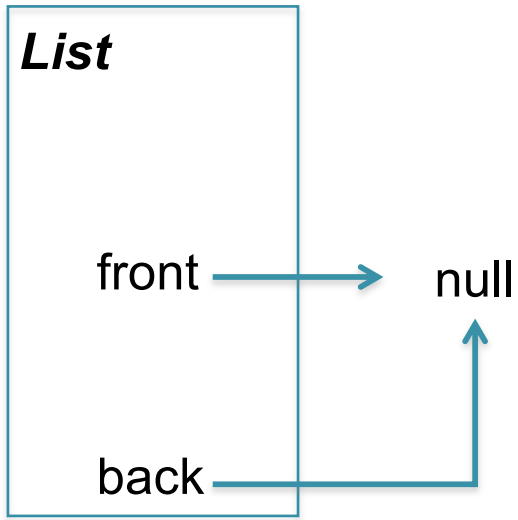
```
public class NodeList<T> implements List<T>
    private static class Node<T> implements Position<T> {
        Node<T> next;
        Node<T> prev;
        T data;
        List<T> owner; // reference back to the List it came from
    }

    private Node<T> front;
    private Node<T> back;
    private int elements;
    public int length() { return this.elements; }

    public Position<T> insertFront(T t) {
       ...
    }

    public Position<T> insertBack(T t) {
       ...
    }

    public void remoteAt(Position<T> p) {
       ...
    }
}
```

Public Position interface,
but nested (private static) Node Implementation

List v4

```
List l = new List<String>();
```

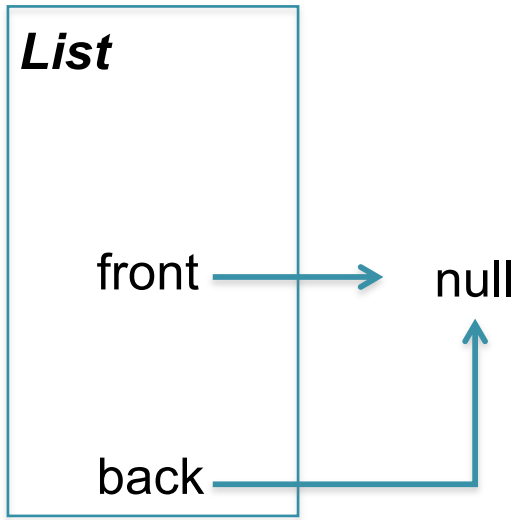
```
public interface Position<T> {
    // empty on purpose
}

public interface List<T> {
     // simplified interface
     int length();

     Position<T> insertFront(T t);
     Position<T> insertBack(T t);

     // TODO: void is temporary
     void removeAt(Position<T> p);
}
```



List v4

```
List l = new List<String>();

Position a = l.insertFront("Mike");
```

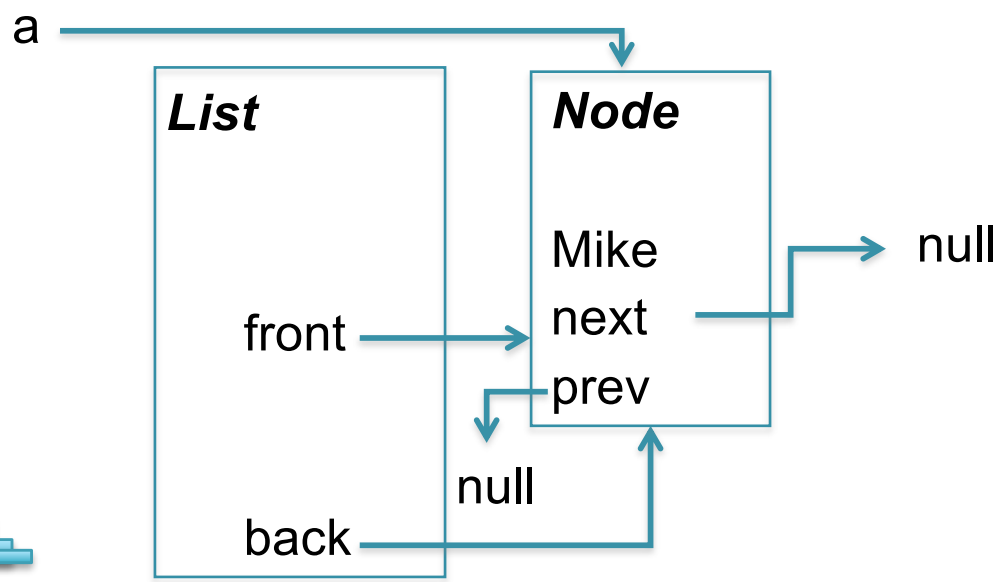
```
public interface Position<T> {
    // empty on purpose
}

public interface List<T> {
     // simplified interface
     int length();

     Position<T> insertFront(T t);
     Position<T> insertBack(T t);

     // TODO: void is temporary
     void removeAt(Position<T> p);
}
```



List v4

```
List l = new List<String>();

Position a = l.insertFront("Mike");
```

```
public interface Position<T> {
    // empty on purpose
}

public interface List<T> {
     // simplified interface
     int length();

     Position<T> insertFront(T t);
     Position<T> insertBack(T t);

     // TODO: void is temporary
     void removeAt(Position<T> p);
}
```



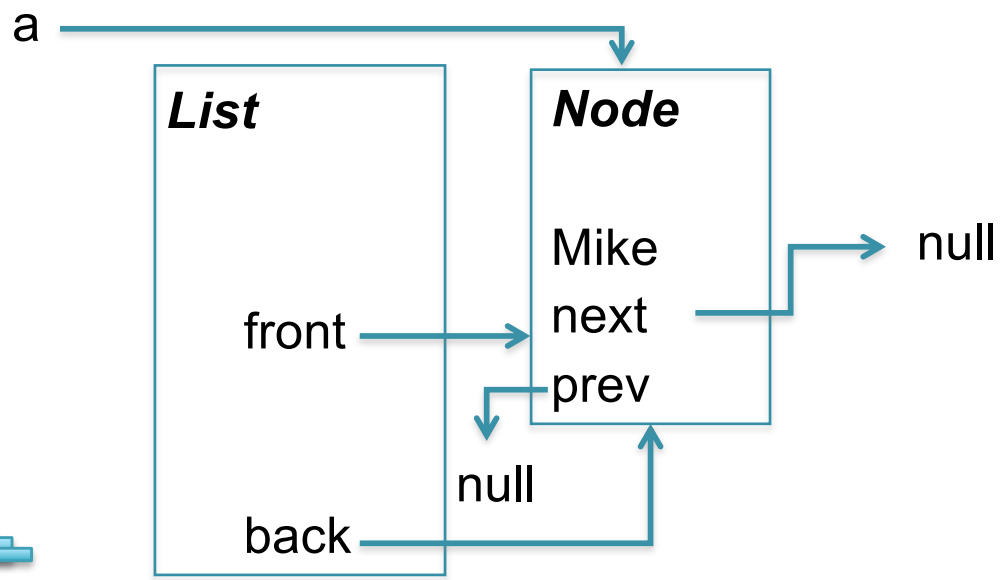
On the first insert, both front and back refer to the new node

Note the reference to Position a to returned to the client

List v4

```
List l = new List<String>();

Position a = l.insertFront("Mike");
Position b = l.insertBack("Peter");
```

```
public interface Position<T> {
    // empty on purpose
}

public interface List<T> {
     // simplified interface
     int length();

     Position<T> insertFront(T t);
     Position<T> insertBack(T t);

     // TODO: void is temporary
     void removeAt(Position<T> p);
}
```



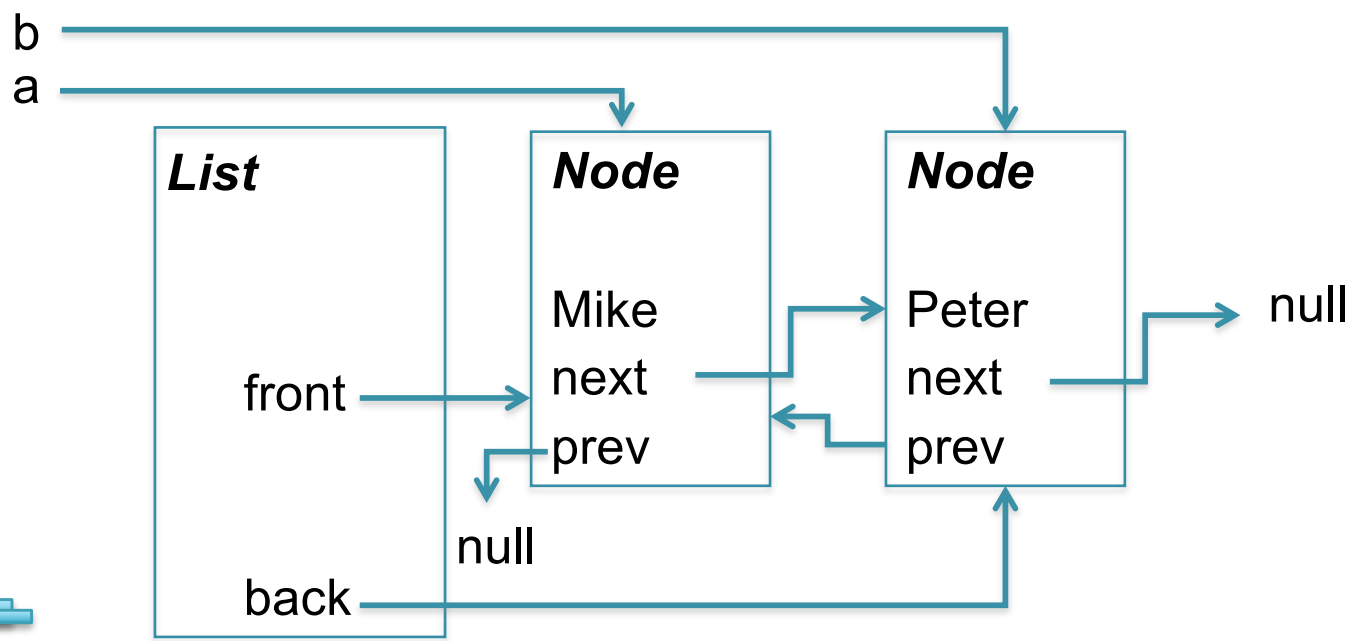
On the first insert, both front and back refer to the new node

Note the reference to Position a to returned to the client

List v4

```
List l = new List<String>();

Position a = l.insertFront("Mike");
Position b = l.insertBack("Peter");
```

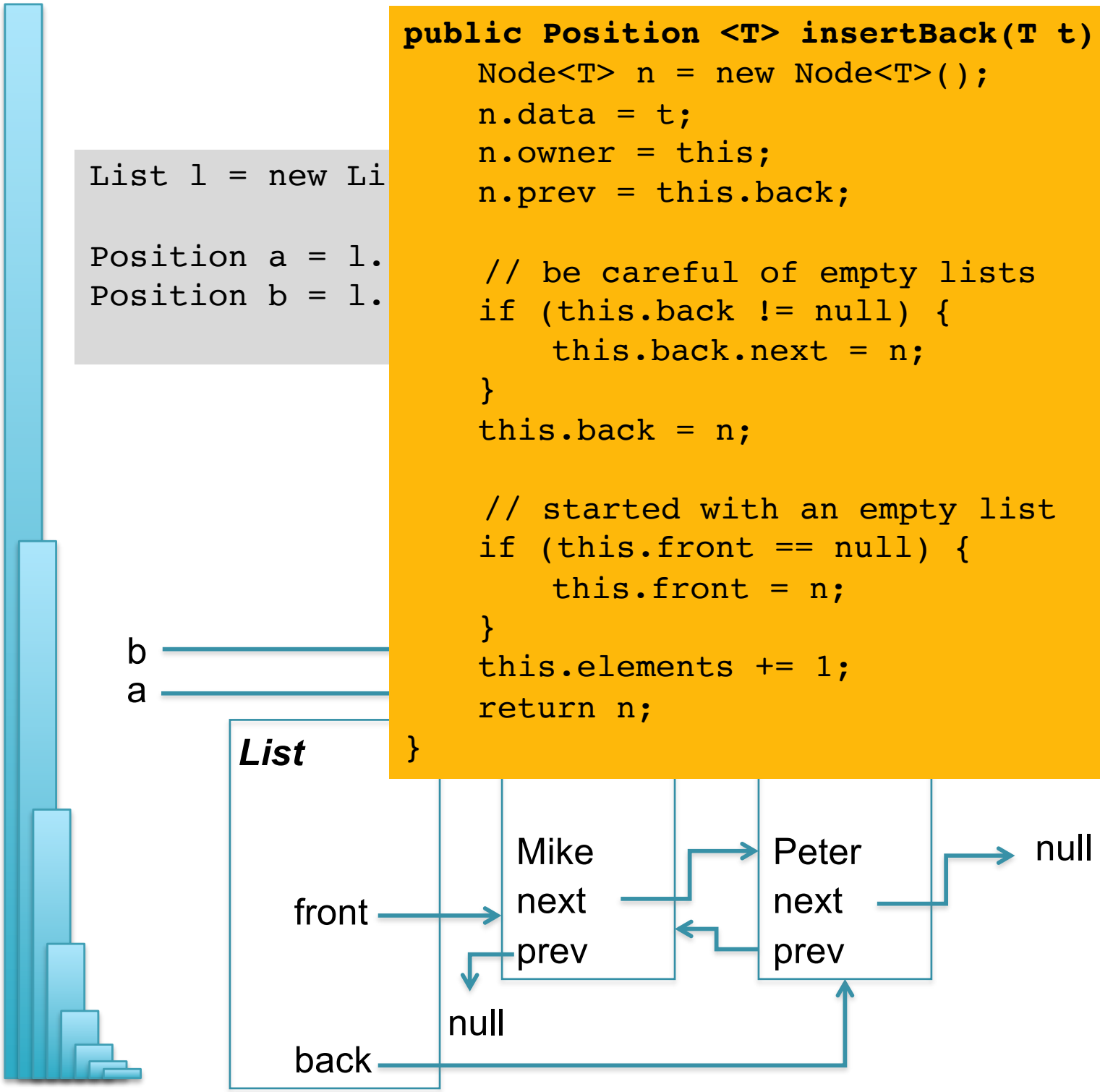
```
public interface Position<T> {
    // empty on purpose
}

public interface List<T> {
     // simplified interface
     int length();

     Position<T> insertFront(T t);
     Position<T> insertBack(T t);

     // TODO: void is temporary
     void removeAt(Position<T> p);
}
```



```
List l = new Li

Position a = l.
Position b = l.
```

```
public Position <T> insertBack(T t) {
    Node<T> n = new Node<T>();
    n.data = t;
    n.owner = this;
    n.prev = this.back;

    // be careful of empty lists
    if (this.back != null) {
        this.back.next = n;
    }
    this.back = n;

    // started with an empty list
    if (this.front == null) {
        this.front = n;
    }
    this.elements += 1;
    return n;
}
```

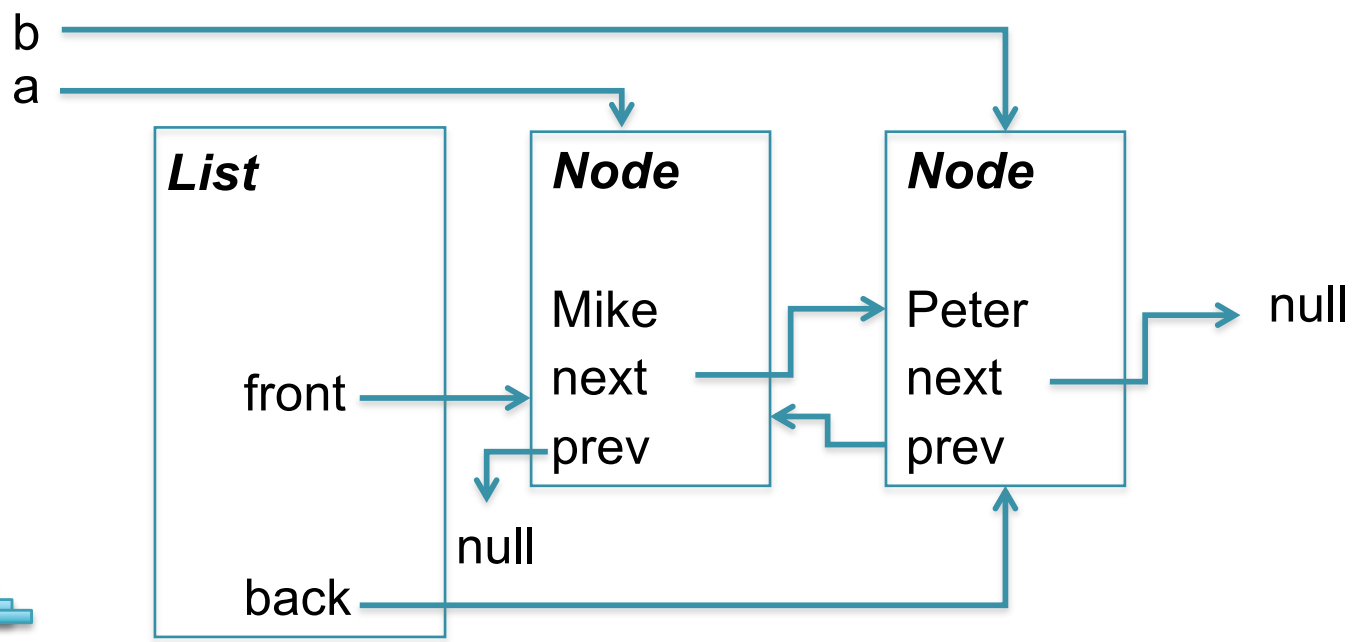
```
sition<T> {
pose


st<T> {
interface


sertFront(T t);
sertBack(T t);

is temporary
Position<T> p);
```



List v4

```
List l = new List<String>();

Position a = l.insertFront("Mike");
Position b = l.insertBack("Peter");
Position c = l.insertFront("Kelly");
```

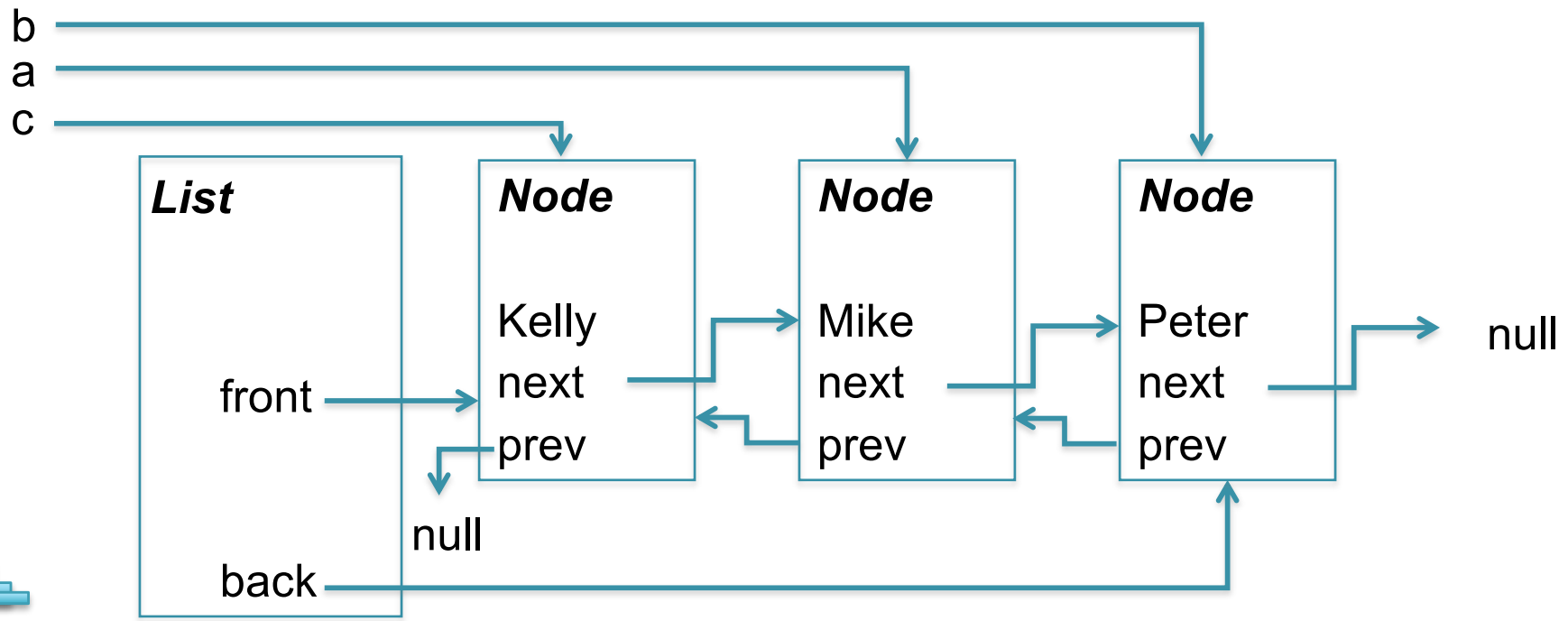
```
public interface Position<T> {
    // empty on purpose
}

public interface List<T> {
     // simplified interface
     int length();

     Position<T> insertFront(T t);
     Position<T> insertBack(T t);

     // TODO: void is temporary
     void removeAt(Position<T> p);
}
```



List v4

```
List l = new List<String>();

Position a = l.insertFront("Mike");
Position b = l.insertBack("Peter");
Position c = l.insertFront("Kelly");
```

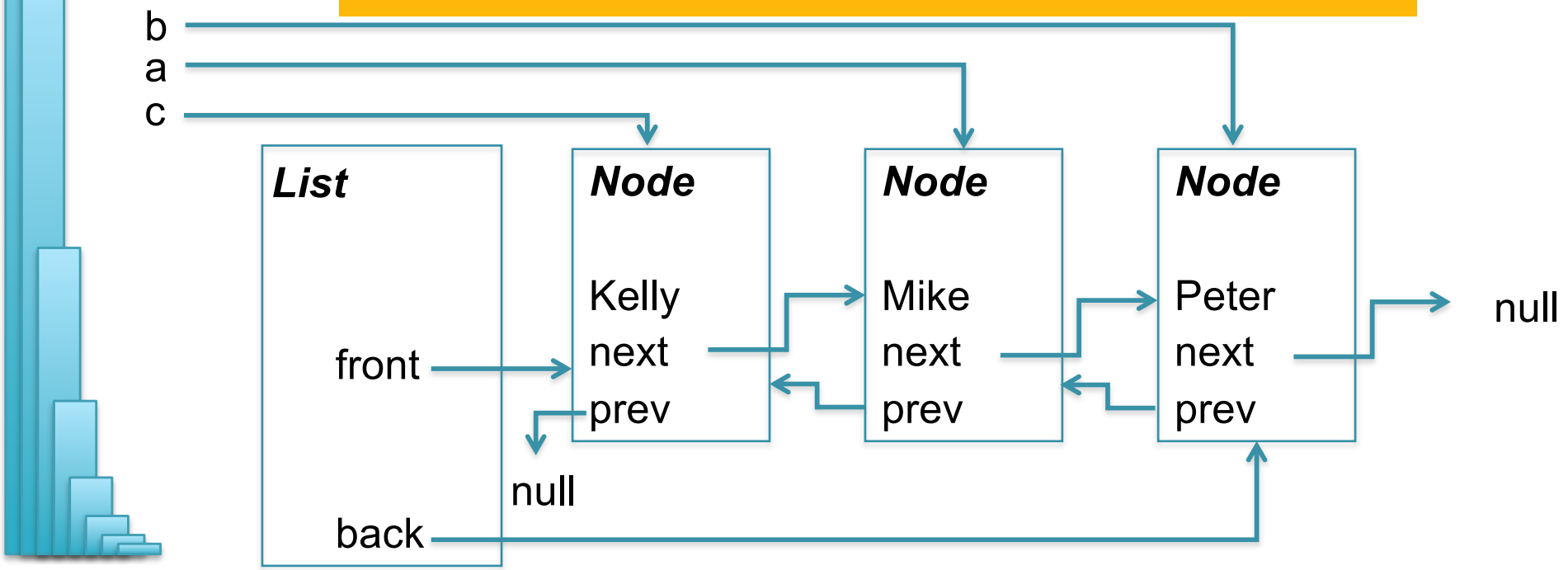
```
public interface Position<T> {
    // empty on purpose
}

public interface List<T> {
     // simplified interface
     int length();

     Position<T> insertFront(T t);
     Position<T> insertBack(T t);

     // TODO: void is temporary
     void removeAt(Position<T> p);
}
```



```
List l = ne

Position a
Position b
Position c
```

```
public Position <T> insertFront(T t ) {
    Node<T> n = new Node<T>();
    n.data = t;
    n.owner = this;
    n.next = this.front;
    if (this.front != null) {
        this.front.prev = n;
    }
    this.front =n;
    if (this.back==null) {
        this.back = n;
    }
    this.elements +=1;
    return n;
}
```

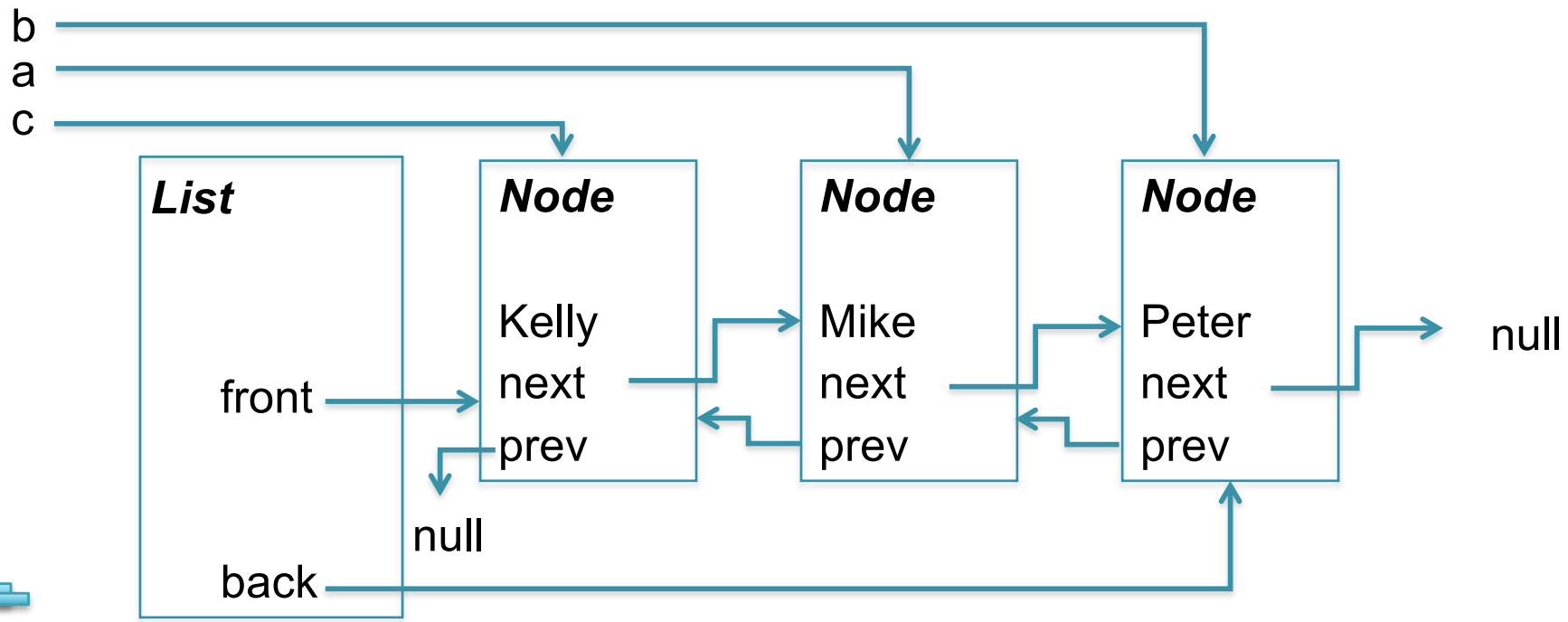
```
.on<T> {
e


T> {
erface


tFront(T t);
tBack(T t);

temporary
tion<T> p);
```



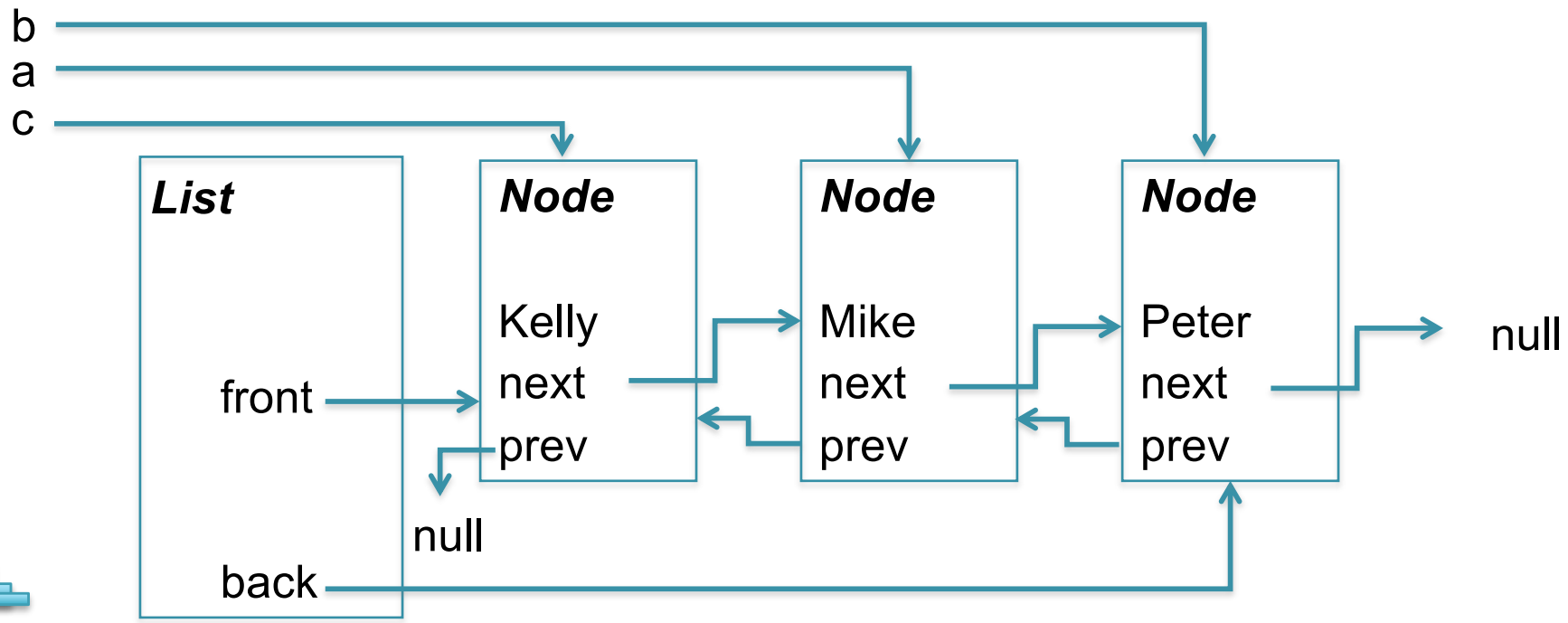
List v4

```
List l = new List<String>();

Position a = l.insertFront("Mike");
Position b = l.insertBack("Peter");
Position c = l.insertFront("Kelly");

l.removeAt(a);
```

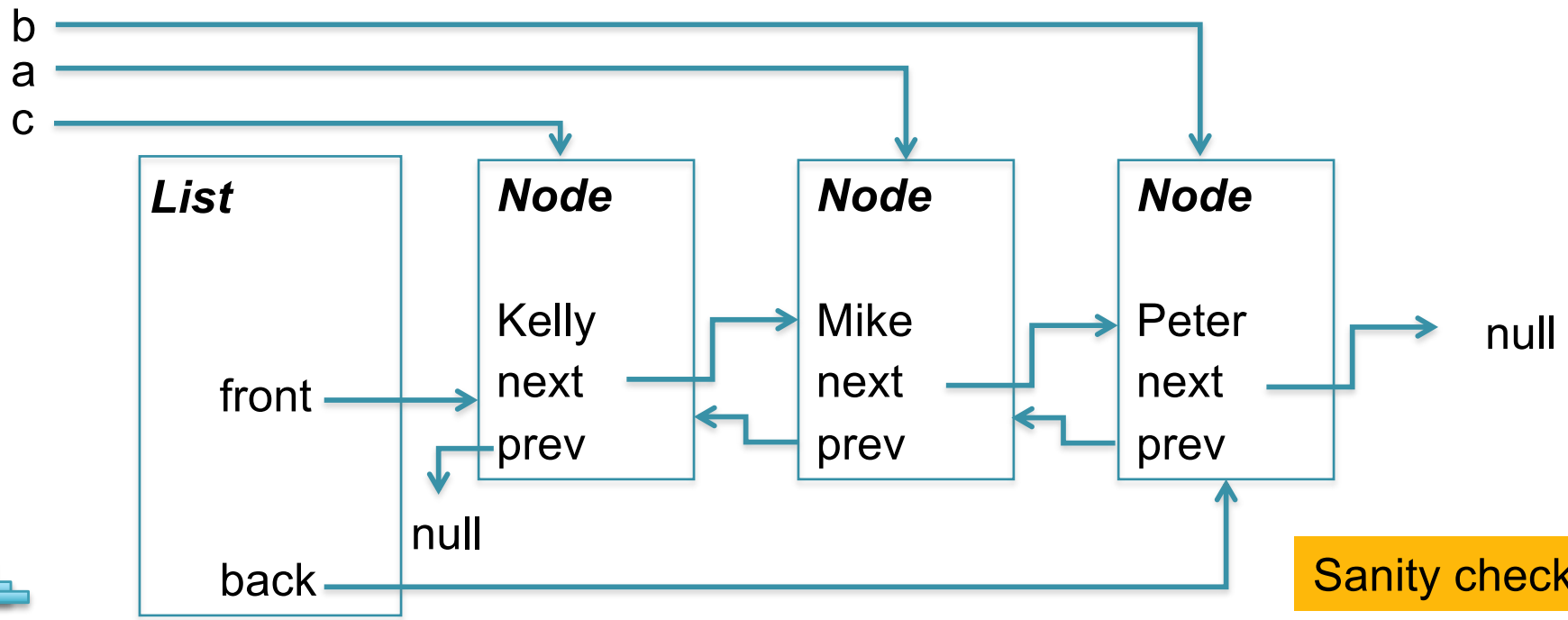
```
public interface Position<T> {
    // empty on purpose
}

public interface List<T> {
     // simplified interface
     int length();

     Position<T> insertFront(T t);
     Position<T> insertBack(T t);

     // TODO: void is temporary
     void removeAt(Position<T> p);
}
```



```
public void removeAt(Position<T> p) {
    if (!( p instanceof Node<?>) || p == null ) {
        // wildcard or nothing
        throw new InvalidPositionException ();
    }
    Node<T> n = (Node<T>) p;
    if (n.owner != this) { throw new InvalidPositionException (); }
    if (n.next != null)  { n.next.prev = n.prev; }
    if (n.prev != null)  { n.prev.next = n.next; }
    if (n == this.front) { this.front  = n.next; }
    if (n == this.back)  { this.back   = n.prev; }
    n.owner = null;
    this.elements -= 1;
}
```

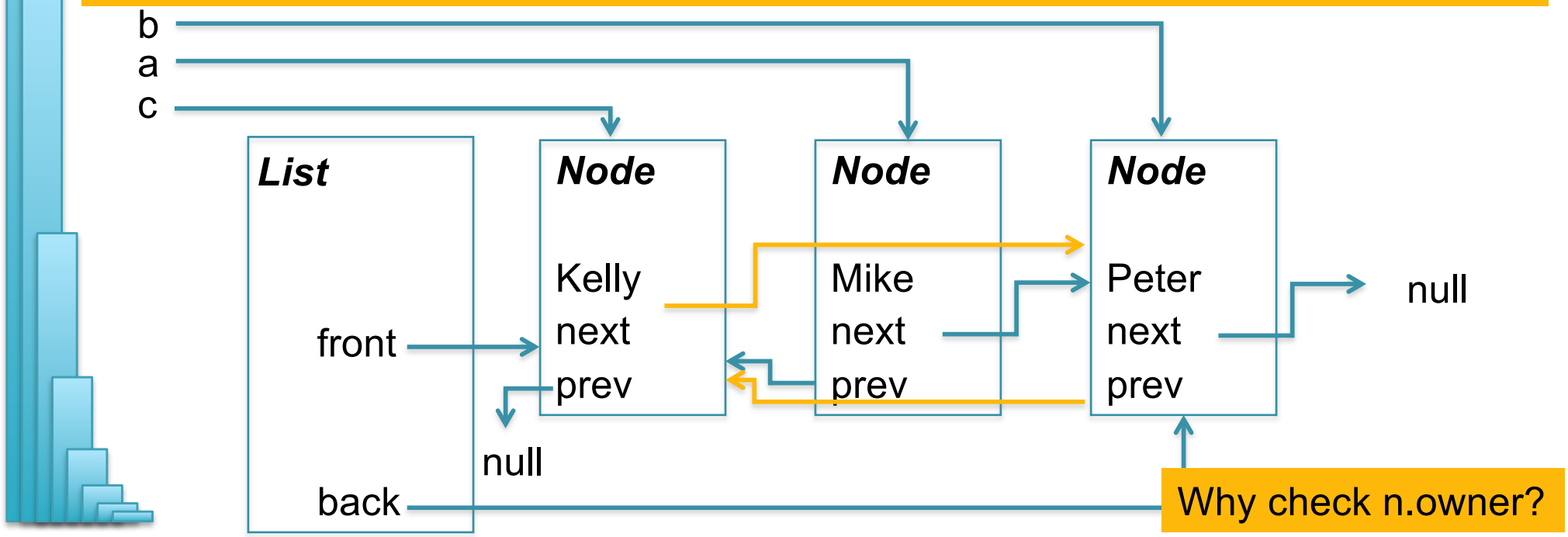


```
public void removeAt(Position<T> p) {
    if (!( p instanceof Node<?>) || p == null ) {
        // wildcard or nothing
        throw new InvalidPositionException ();
    }
    Node<T> n = (Node<T>) p;
    if (n.owner != this) { throw new InvalidPositionException (); }
    if (n.next != null)  { n.next.prev = n.prev; }
    if (n.prev != null)  { n.prev.next = n.next; }
    if (n == this.front) { this.front  = n.next; }
    if (n == this.back)  { this.back   = n.prev; }
    n.owner = null;
    this.elements -= 1;
}
```

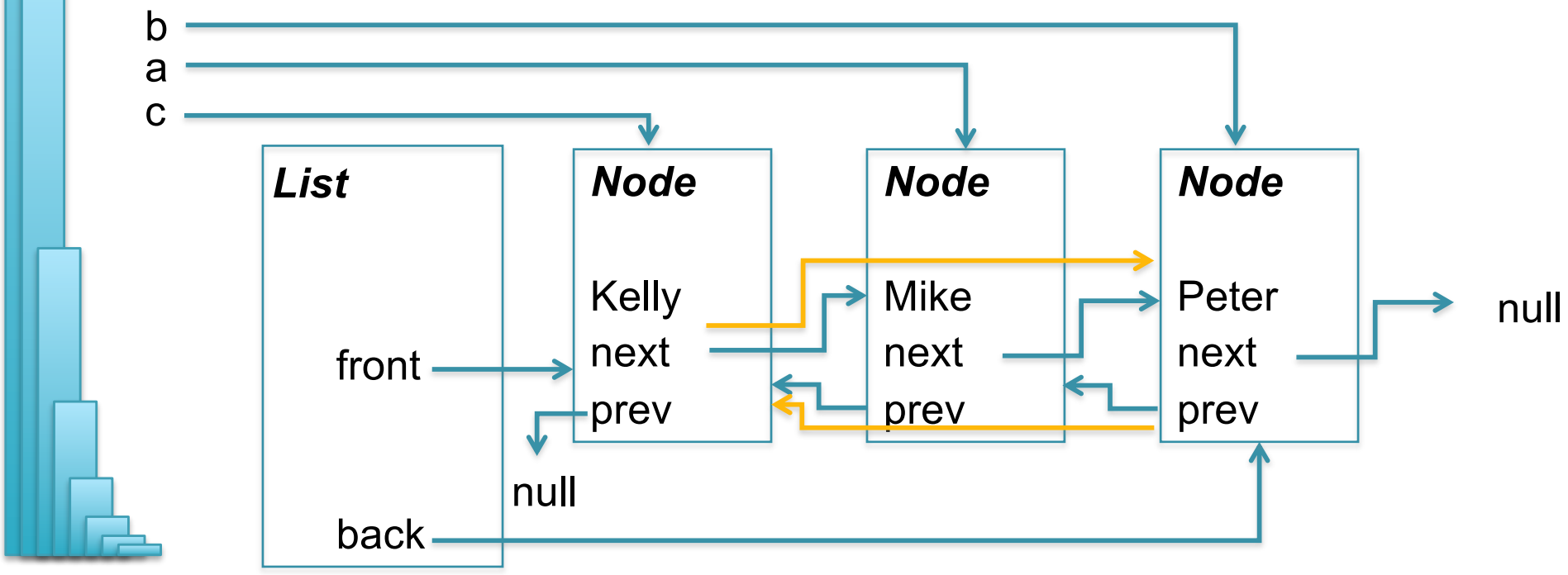


Sanity check p

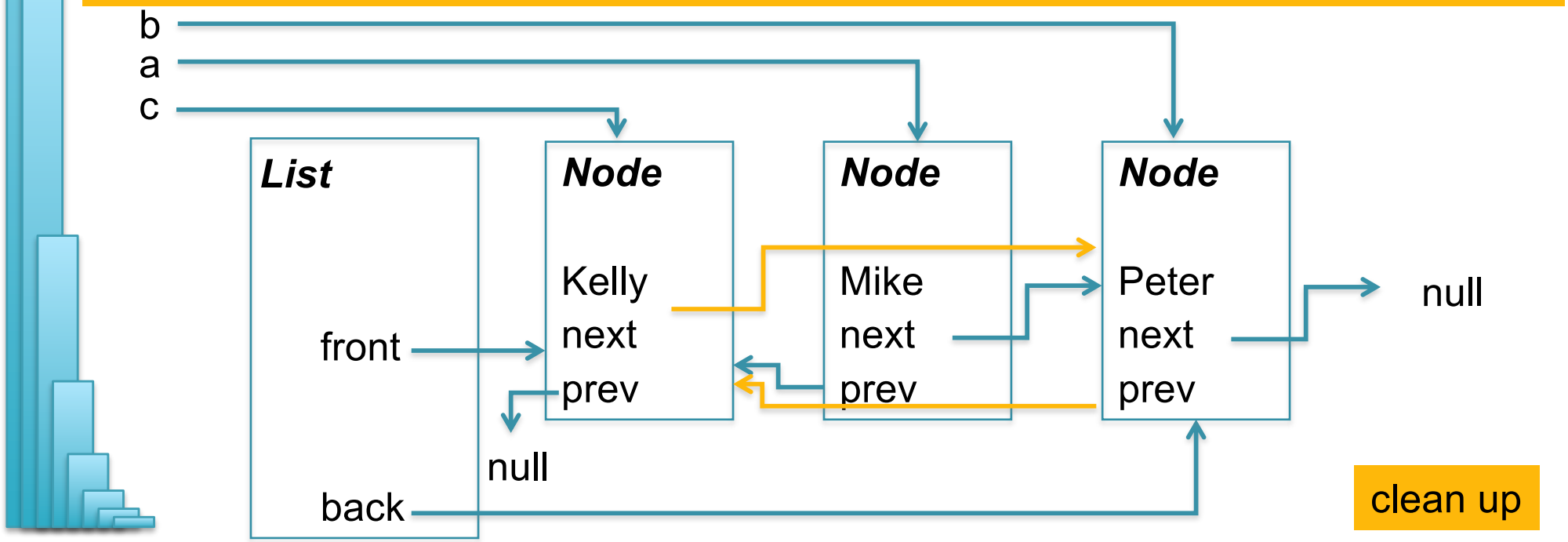
```
public void removeAt(Position<T> p) {
    if (!( p instanceof Node<?>) || p == null ) {
        // wildcard or nothing
        throw new InvalidPositionException ();
    }
    Node<T> n = (Node<T>) p;
    if (n.owner != this) { throw new InvalidPositionException (); }
    if (n.next != null)  { n.next.prev = n.prev; }
    if (n.prev != null)  { n.prev.next = n.next; }
    if (n == this.front) { this.front  = n.next; }
    if (n == this.back)  { this.back   = n.prev; }
    n.owner = null;
    this.elements -= 1;
}
```



```
public void removeAt(Position<T> p) {
    if (!( p instanceof Node<?>) || p == null ) {
        // wildcard or nothing
        throw new InvalidPositionException ();
    }
    Node<T> n = (Node<T>) p;
    if (n.owner != this) { throw new InvalidPositionException (); }
    if (n.next != null)  { n.next.prev = n.prev; }
    if (n.prev != null)  { n.prev.next = n.next; }
    if (n == this.front) { this.front  = n.next; }
    if (n == this.back)  { this.back   = n.prev; }
    n.owner = null;
    this.elements -= 1;
}
```



```
public void removeAt(Position<T> p) {
    if (!( p instanceof Node<?>) || p == null ) {
        // wildcard or nothing
        throw new InvalidPositionException ();
    }
    Node<T> n = (Node<T>) p;
    if (n.owner != this) { throw new InvalidPositionException (); }
    if (n.next != null)  { n.next.prev = n.prev; }
    if (n.prev != null)  { n.prev.next = n.next; }
    if (n == this.front) { this.front  = n.next; }
    if (n == this.back)  { this.back   = n.prev; }
    n.owner = null;
    this.elements -= 1;
}
```



clean up

List v4

```
public String toString () {
  String s ="[";
  Node<T> n = this.front;
  while (n != null) {
    s += n.data.toString();
    if (n.next != null) {
      s +="";
    }
    n = n.next;
  }
  s +="]";
  return s;
}
```

```
public static void main(String[] args) {
  List l = new NodeList();
  Position a = l.insertFront("Mike");
  System.out.println(l);
  Position b = l.insertBack("Peter");
  System.out.println(l);
  Position c = l.insertFront("Kelly");
  System.out.println(l);

  l.remove(a);
  System.out.println(l);
}
```

```
$ java NodeList
[Mike]
[Mike, Peter]
[Kelly, Mike, Peter]
[Kelly, Peter]
```

How to test the code?

```
public class NodeList<T> implements List<T> {
    private static final class Node<T> implements Position<T> {
        Node<T> next;
        Node<T> prev;
        T data;
        List<T> owner;
        public T get() { return this.data;  }
        public void put(T t) { this.data = t; }
    }
    ...
    public Position<T> front() throws EmptyException { … }
    public Position<T> back() throws EmptyException { … }
    public Position<T> insertFront(T t) { … }
    public Position<T> insertBack(T t) { … }
    public void removeFront() throws EmptyException { … }
    public void removeBack() throws EmptyException { … }
    public Position<T> insertBefore(Position<T> p, T t)
            throws PositionException { … }
    public Position<T> insertAfter(Position<T> p, T t)
        throws PositionException { … }
    public void remove(Position<T> p) throws PositionException { … }
    public String toString() { … }
}
```

How to test the code?

```
public class NodeList<T> implements List<T> {
    private static final class Node<T> implements Position<T> {
        Node<T> next;
        Node<T> prev;
        T data;
        List<T> owner;
        public T get() { return this.data;  }
        public void put(T t) { this.data = t; }
    }
    ...
    public Position<T> front() throws EmptyException { … }
    public Position<T> back() throws EmptyException { … }
    public Position<T> insertFront(T t) { … }
    public Position<T> insertBack(T t) { … }
    public void removeFront() throws EmptyException { … }
    public void removeBack() throws EmptyException { … }
    public Position<T> insertBefore(Position<T> p, T t)
            throws PositionException { … }
    public Position<T> insertAfter(Position<T> p, T t)
        throws PositionException { … }
    public void remove(Position<T> p) throws PositionException { … }
    public String toString() { … }
}
```

TestList.java (1)

```
import org.junit.Test;
import org.junit.Before;
import static org.junit.Assert.assertEquals;

public class TestList {
    List<String> list ;

    @Before
    public void setupList () {
        list = new NodeList<String>();
    }

    @Test
    public void testEmptyList () {
        assertEquals ("[]", list.toString());
        assertEquals (0, list.length());
    }

    @Test
    public void testInsertFront () {
        list.insertFront("Peter");
        list.insertFront("Paul");
        assertEquals("[Paul, Peter]", list.toString());
        assertEquals(2, list.length());
    }
```

TestList.java (2)

```
@Test
public void testInsertBack () {
    list.insertBack("Peter");
    list.insertBack("Paul");
    assertEquals ("[Peter, Paul]", list.toString ());
    assertEquals (2, list.length ());
}
```

What tests are missing?

As we add functionality,
testing code will become significantly longer
than the implementation
☺

Next Steps

1. Work on HW3

2. Check on Piazza for tips & corrections!